FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

A LOS ASISTENTES A LOS CURSOS

Las autoridades de la Facultad de Ingeniería, por conducto del jefe de la División de Educación Continua, otorgan una constancia de asistencia a quienes cumplan con los requisitos establecidos para cada curso.

El control de asistencia se llevará a cabo a través de la persona que le entregó las notas. Las inasistencias serán computadas por las autoridades de la División, con el fin de entregarle constancia solamente a los alumnos que tengan un mínimo de 80% de asistencias.

Pedimos a los asistentes recoger su constancia el día de la clausura. Estas se retendrán por el periodo de un año, pasado este tiempo la DECFI no se hará responsable de este documento.

Se recomienda a los asistentes participar activamente con sus ideas y experiencias, pues los cursos que ofrece la División están planeados para que los profesores expongan una tesis, pero sobre todo, para que coordinen las opiniones de todos los interesados, constituyendo verdaderos seminarios.

Es muy importante que todos los asistentes llenen y entreguen su hoja de inscripción al inicio del curso, información que servirá para integrar un directorio de asistentes, que se entregará oportunamente.

Con el objeto de mejorar los servicios que la División de Educación Continua ofrece, al final del curso deberán entregar la evaluación a través de un cuestionario diseñado para emitir juicios anónimos.

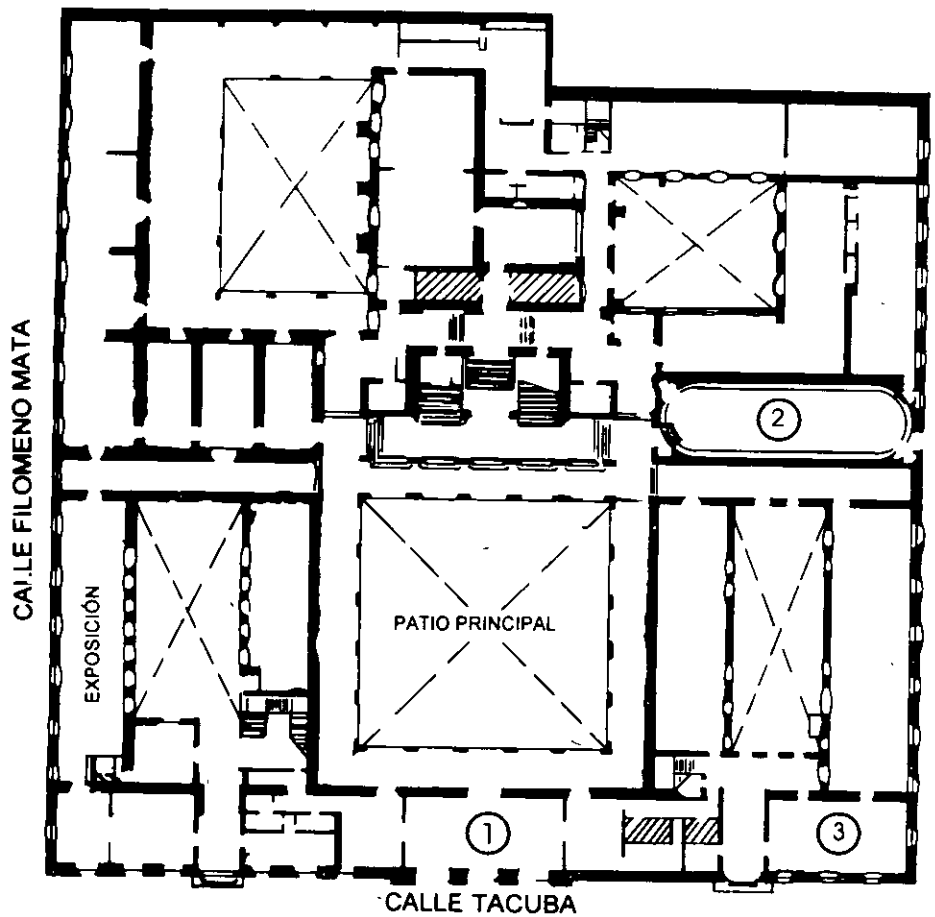
Se recomienda llenar dicha evaluación conforme los profesores impartan sus clases, a efecto de no llenar en la última sesión las evaluaciones y con esto sean más fehacientes sus apreciaciones.

Atentamente

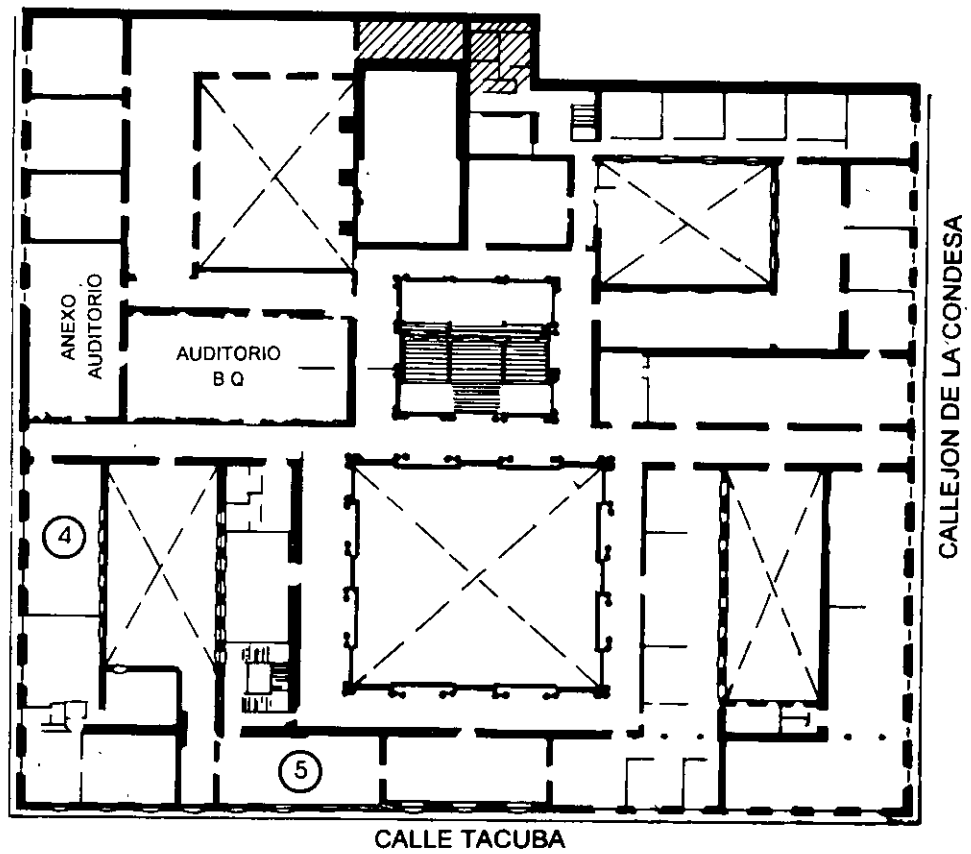
División de Educación Continua.

Palacio de Mineria Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. APDO. Postal M-2285
Teléfonos: 512-8955 512-5121 521-7335 521-1987 Fax 510-0573 521-4020 AL 26

PALACIO DE MINERIA

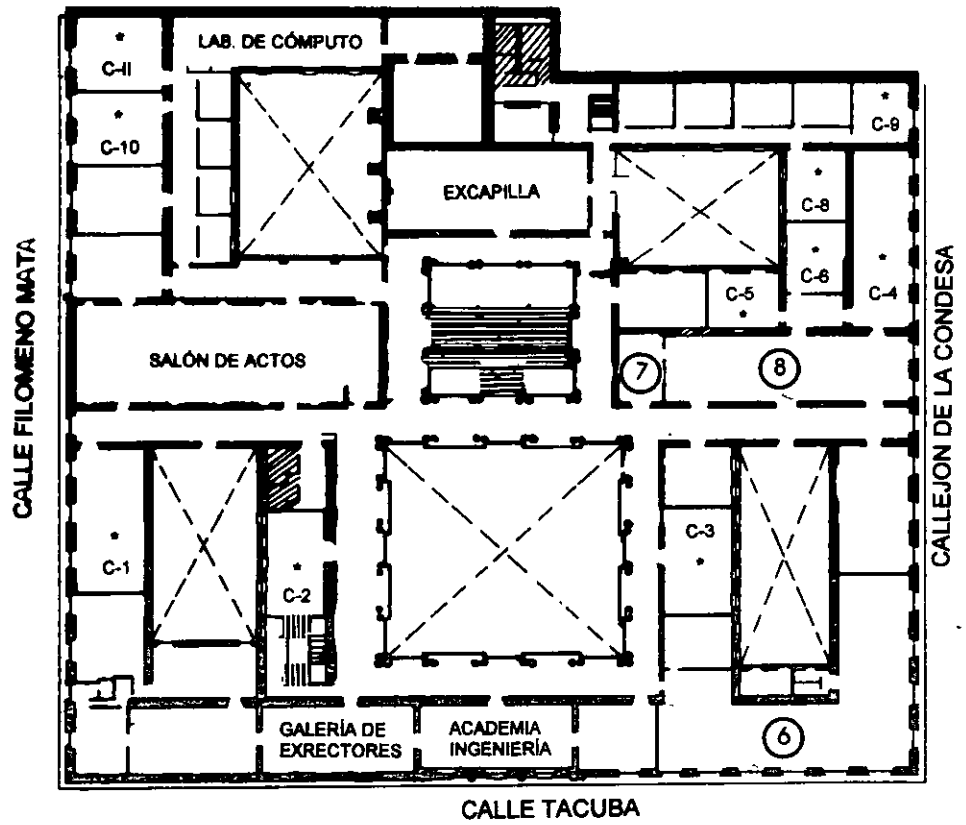


PLANTA BAJA



MEZZANINNE

PALACIO DE MINERIA



Ier. PISO

GUÍA DE LOCALIZACIÓN

1. ACCESO
2. BIBLIOTECA HISTÓRICA
3. LIBRERÍA UNAM
4. CENTRO DE INFORMACIÓN Y DOCUMENTACIÓN "ING. BRUNO MASCANZONI"
5. PROGRAMA DE APOYO A LA TITULACIÓN
6. OFICINAS GENERALES
7. ENTREGA DE MATERIAL Y CONTROL DE ASISTENCIA
8. SALA DE DESCANSO

SANITARIOS

* AULAS

DIVISIÓN DE EDUCACIÓN CONTINUA
FACULTAD DE INGENIERÍA U.N.A.M.
CURSOS ABIERTOS

VNIVERSIDAD NACIONAL
AVTÓNOMA DE
MEXICO

Procesamiento Paralelo en Sistemas de Tiempo Real

División de Educación Continua de la
Facultad de Ingeniería

en colaboración con el

Instituto de Investigaciones en Matemáticas
Aplicadas y en Sistemas

Dr. Fabián García Nocetti
Dr. Julio Solano González

5. Lenguaje Occam
    5.1 Primitivas
    5.2 Constructores y Replicadores
    5.3 Operadores y Arreglos
    5.4 Funciones y Procedimientos
    5.5 Protocolos de Comunicación
    5.6 Timers y Puertos

6. Programación de Procesos Paralelos
    6.1 Metodología
    6.2 Programación en un procesador
    6.3 Programación en múltiples procesadores
    6.4 Configuración de programas
    6.5 Mapeo de procesos en procesadores
    6.6 Casos de estudio

7. Aplicaciones
    7.1 Control
    7.2 Robótica
    7.3 Procesamiento de Señales e Imágenes

Apéndice. Occam 2 Toolset

TECNICAS DE ENSEÑANZA

Las sesiones teóricas estarán complementadas con lecturas relativas a los temas tratados y con ejercicios prácticos de programación paralela. Para desarrollar el trabajo práctico se utilizará una plataforma multiusuario de procesamiento paralelo, accesible via red local. Este sistema está integrado por 16 transputers y cuenta con compiladores paralelos y herramientas de software.

BIBLIOGRAFIA

Stone, H.S., "High Performance Computer Architectures", Addison Wesley, 1987.

Krishnamurthy, E.V., "Parallel Processing Principles and Practice", Addison Wesley, 1989.

Bertsekas, D.P., "Parallel and Distributed Computation-Numerical Methods", Prentice Hall, 1989.

Pountain, D., May, D.; "A Tutorial Introduction to Occam Programming", Blackwell Publications, 1987.

Jones, G., Goldsmith, M., "Programming in Occam 2", Prentice Hall, 1988.

Harp, G., "Transputer Applications", Computer Systems Series, Pitman, 1989.

May, M.D., "Networks, Routers and Transputers -Function, Performance and Applications, IOS Press, 1993.

Thoeni, U., "Programming Real-Time Multicomputers for Signal Processing", Prentice Hall, 1994.

Russ, J.C., "The Image Processing Handbook -2nd. Edition", CRC Press , 1995.

Webber, H.C., "Image Processing and Transputers", IOS Press, 1992.

1

Procesamiento Paralelo

PROCESAMIENTO PARALELO, TRANSPUTERS Y OCCAM

DR. FABIAN GARCIA NOCETTI
DR. JULIO SOLANO GONZALEZ

Laboratorio de Procesamiento Paralelo
Departamento de Electrónica y Automatización
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Universidad Nacional Autónoma de México
Apdo. Postal 20-726 Admon. 20, Delegación Alvaro Obregón
01000 México D.F.

1. INTRODUCCION.

Los sistemas convencionales de cómputo operan en una forma secuencial, en donde las instrucciones de un programa son ejecutadas una a la vez. Esta característica ha sido forzada por la arquitectura secuencial de computadoras convencionales (arquitectura von Neumann), en la cual un procesador central esta conectado a un banco de memoria por medio de un bus. Desde entonces la mayoría de sucesivas generaciones de computadoras disponibles han seguido este diseño [1]. Sin embargo una gran variedad de problemas asociados a las áreas de control, visión, simulación, procesamiento de voz, imágenes y procesamiento digital de señales poseen un paralelismo intrínseco. De hecho el resolver este tipo de problemas en forma secuencial ha sido computacionalmente intensivo y restrictivo, sobre todo cuando se trata con aplicaciones en tiempo real, en donde se manejan intervalos de muestreo muy cortos del orden de milisegundos.

Un caso típico en ingeniería de control es la implementación de un controlador digital en tiempo real, en donde el cálculo de las variables controladas debe ser realizado dentro de un intervalo de muestreo típico de 5-20 ms [2]. Aun considerando el poder de cómputo de modernos procesadores secuenciales convencionales, esto puede ser difícil de alcanzar, sobre todo cuando se manejan múltiples variables. Ya que este tipo de sistemas puede envolver típicamente, un número de algoritmos de control, simulación, optimización, filtrado e identificación, junto con otras actividades como adquisición y chequeo de información. Claramente, a mayor complejidad de los algoritmos corresponde una mayor dificultad en el problema de realizar los cálculos necesarios en tiempo-real.

Aplicaciones diferentes imponen demandas variables en el controlador. Por ejemplo, los intervalos de muestreo para el control de motores eléctricos deben ser cortos debido a las pequeñas constantes de tiempo manejadas. Sistemas multivariables, envuelven un mayor nivel de complejidad en el cálculo de las señales controladas ya que estas deben ser calculadas simultáneamente [3]. En sistemas de seguridad, donde la confiabilidad es fundamental, la incorporación de características adicionales tales como tolerancia de fallas y redundancia, se pueden traducir en un incremento considerable en la carga computacional.

Como se puede observar, ciertas aplicaciones reales están sobrepasando los límites de desempeño de arquitecturas convencionales [4]. Aunque el uso de la tecnología de integración ha traído como resultado un incremento en la velocidad de los procesadores y en consecuencia de los sistemas de cómputo, la tecnología no ha logrado incrementar la velocidad en la misma proporción que el nivel de integración. La velocidad máxima a la que las componentes electrónicas operan ha marcado un límite en el diseño de procesadores más veloces. La alternativa ha sido entonces modificar la arquitectura típica de los sistemas de cómputo. Esto ha estimulado la investigación y el uso de arquitecturas alternativas que satisfagan las nuevas demandas computacionales de una manera efectiva y práctica. Procesamiento paralelo ha sido una de las mas viables alternativas.

La disponibilidad actual de arquitecturas de procesamiento paralelo, que permiten distribuir tanto algoritmos como información sobre un número de procesadores, ha creado nuevas oportunidades para el diseño e implementación de sistemas más rápidos y complejos [5]. Procesamiento paralelo están siendo cada vez más atractivo como un medio para construir sistemas de alto desempeño y confiabilidad [6].

La manera convencional de lograr alto desempeño es mediante la partición del sistema en módulos y la distribución de éstos en un número adecuado de procesadores para lograr el menor tiempo de ejecución posible. Confiabilidad es llevada a cabo mediante el uso de tolerancia a fallas, que es la habilidad de un sistema para continuar operando en presencia de algún tipo de falla. La tecnología de cómputo paralelo de hecho ha teniendo un gran impacto en el diseño e implementación de sistemas en tiempo real asociados a problemas de control, visión, simulación, procesamiento de voz, imágenes y procesamiento digital de señales. Dicha tecnología ha expandido el dominio factible de este tipo de sistemas [7]. Especialmente en sistemas que operan en tiempo real y sistemas integrados, donde las limitaciones de procesamiento han restringido tradicionalmente sus capacidades.

En particular la introducción del *transputer* y su lenguaje asociado *occam*, para el soporte de procesamiento paralelo, ha tenido un gran impacto en un número de aplicaciones, permitiendo el diseño y construcción de sistemas más rápidos y complejos, de una forma simple y estructurada [8]. El transputer y occam, han sido diseñados específicamente para ser usados en sistemas de múltiples procesadores, y diversas topologías, ofreciendo el prospecto de un desempeño escalable a medida que más procesadores son agregados al sistema, siendo éste un factor determinante para su utilización en una creciente variedad de aplicaciones.

2. PROCESAMIENTO PARALELO

2.1 Beneficios de Procesamiento Paralelo

Un sistema de procesamiento paralelo esta compuesto por un número de elementos de procesamiento (PEs) que pueden operar en forma concurrente, comunicándose entre si cuando es necesario, Figura 1. Las diversas arquitecturas difieren entre sí, tanto por el poder de cómputo de sus PEs como por su modo de conexión.

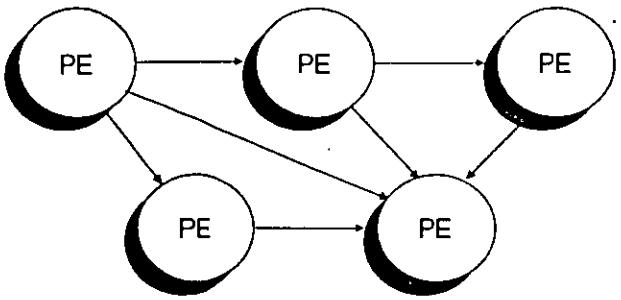


Figura 1. Sistema de Procesamiento Paralelo Generalizado.

Esto conduce a conceptos nuevos tales como granularidad de actividad, el cual es una medida del tamaño de las actividades que pueden ser ejecutadas efectivamente por los PEs de una arquitectura específica. PEs de arquitecturas de grano fino son caracterizados por tener operaciones limitadas y específicas, así como un gran ancho de banda para comunicación de información. Por otro lado, PEs de arquitecturas de grano grande poseen operaciones de propósito más general pero una banda más reducida de comunicación [9]. Ya ha sido señalado que a medida que una aplicación en tiempo real se torna más compleja, un procesador de propósito general resulta inadecuado para ejecutar las operaciones necesarias dentro del intervalo de muestreo requerido. El potencial ofrecido por un sistema de procesamiento paralelo, ofrece una solución viable tanto al problema de velocidad como complejidad. Otros beneficios incluyen, flexibilidad, conectividad y habilidad para reconfiguración, lo cual permite construir diseños de sistemas redundantes y tolerantes a fallas. Además partiendo del hecho que las operaciones concurrentes son un fenómeno natural en sistemas que operan en tiempo real, el uso de procesamiento paralelo habilita al diseñador de este tipo de sistemas a expresar claramente la variedad de actividades secuenciales y paralelas que serán realizadas por el sistema digital objetivo. Finalmente la escalabilidad de un sistema multiprocesador, permite una fácil expansión para acomodar crecientes requerimientos futuros.

2.2 Arquitecturas de Procesamiento Paralelo.

Dentro de un sistema de procesamiento paralelo los procesadores pueden ser interconectados de diferentes maneras, dando lugar a una gran variedad de arquitecturas. Similarmente, una variedad de métodos han sido desarrollados para programar estos sistemas. Frecuentemente se tiende a agrupar estas arquitecturas en dos tipos: arquitecturas de memoria compartida (procesadores pueden accesar memoria común) y arquitecturas de memoria distribuida (cada procesador tiene su propia memoria). Sin embargo, una de las clasificaciones más utilizadas es aquella conocida como taxonomía de Flynn [10]. Esta taxonomía considera la arquitectura

tradicional von Neumann como un modelo *Single Instruction-stream Single Data-stream* (SISD) y a las arquitecturas de procesamiento paralelo como:

1. *Multiple Instruction-stream Single Data-stream* (MISD). Varios procesadores ejecutan simultáneamente diferentes instrucciones en un mismo grupo de datos.
2. *Single Instruction-stream Multiple Data-stream* (SIMD). Varios procesadores ejecutan simultaneamente la misma instrucción en múltiples grupos de datos.
3. *Multiple Instruction-stream Multiple Data-stream* (SIMD). En esta arquitectura cada procesador puede realizar diferentes instrucciones en diferentes datos.

Este curso se ha centrado en arquitecturas del tipo MIMD realizadas con transputers, en donde cada uno de éstos puede ejecutar programas diferentes y se comunican entre sí solo para intercambiar resultados. Un número de topologías han sido derivadas de la implementación de este tipo de arquitecturas. La Figura 2, muestra algunas de las configuraciones mas utilizadas en la construcción de estos sistemas.

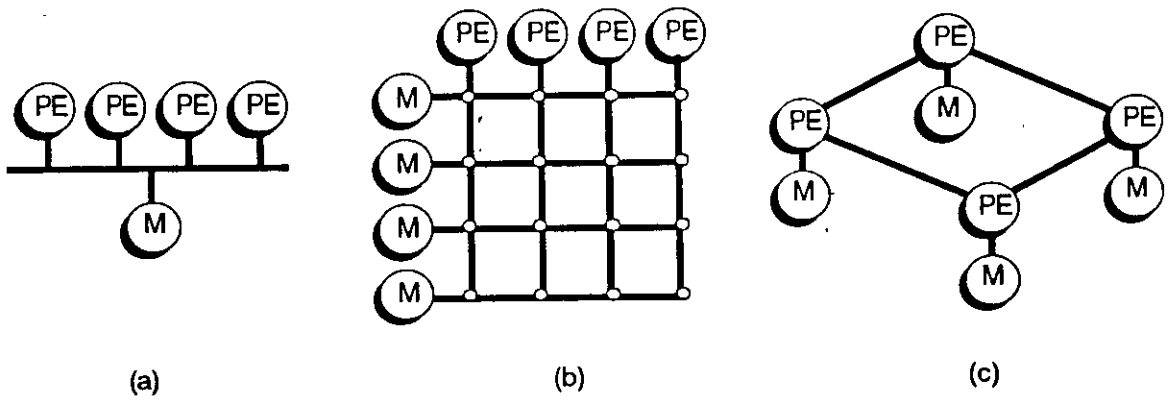


Figura 2. Configuraciones Típicas MIMD: (a)Bus Compartido; (b)Crossbar; (c)Punto
PE = Elementos de Procesamiento; M = Memoria.

Los sistemas MIMD ofrecen una mayor flexibilidad que los sistemas SIMD y MISD, en el manejo de software de propósito general y han sido por tanto adoptados en la solución de problemas en un gran número de disciplinas. Sin embargo es conveniente notar que la implementación exitosa de un determinado algoritmo dependerá del grado de paralelismo que este ofrece, aunado a una adecuada selección de granularidad. El desempeño de un sistema de procesamiento paralelo es también dependiente de la relación conocida como *R/C*, donde *R* es el tiempo de ejecución de un proceso y *C* es el tiempo que ese proceso emplea en comunicaciones con otros procesos. Claramente un valor grande en esta relación es deseable.

Sin embargo no existe una regla definitiva para establecer un valor adecuado. Ya que valores grandes de R/C pueden ignorar paralelismo potencial en el problema, mientras que el subdividir el problema hasta alcanzar el máximo paralelismo, puede conducir a un exceso de

comunicaciones entre procesos. Por lo tanto, en la tarea de distribuir o mapear una aplicación en una arreglo de PEs, el diseñador debe seleccionar cuidadosamente la granularidad del algoritmo, de tal forma que mantenga una adecuada relación entre tiempos de ejecución y comunicación de los procesos que integran el sistema.

3. TRANSPUTER Y OCCAM

El transputer es una arquitectura VLSI que soporta explícitamente concurrencia y sincronización [11]. La diferencia entre un transputer y una microcomputadora ordinaria, es que éste contiene un organizador de actividades "scheduler" integrado, capaz de distribuir su tiempo entre un número de procesos concurrentes, de esta forma el transputer, además de ejecutar procesos en modo secuencial, puede ejecutar procesos concurrentemente. Varios transputers se puede agrupar de una manera rápida y directa para formar redes y arreglos, ver Figura 3. Cada transputer trabaja en su propia actividad y usa su memoria local. Para que los transputers puedan cooperar en un sistema necesitan comunicarse entre si, esto es realizado a través de interfases seriales denominadas "links", cada uno con un canal de entrada y otro de salida.

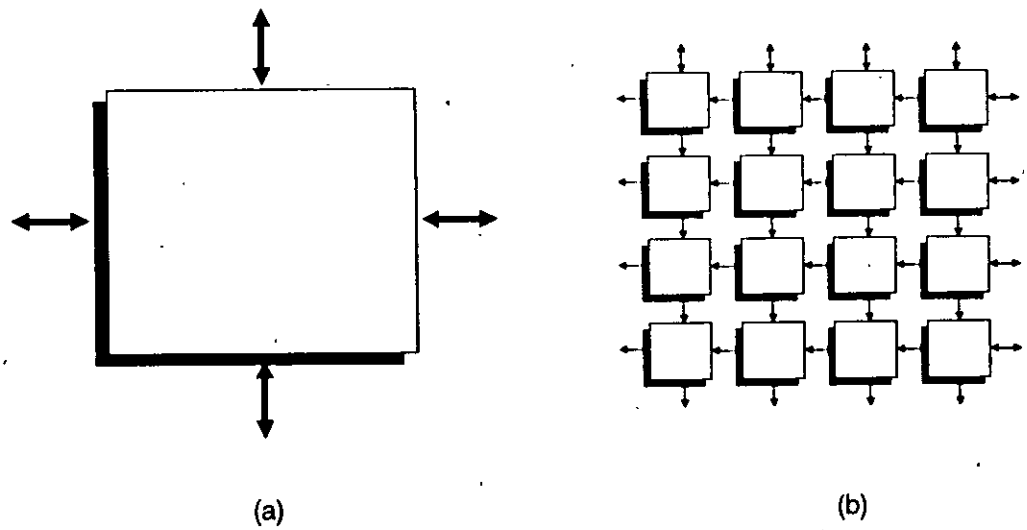


Figura 3. (a) Un Transputer. (b) Una Red de Transputers.

Por su parte el lenguaje occam esta diseñado para manejar tanto procesos secuenciales como concurrentes. Estos son modelados como procesos occam que trabajan con su propia información local. Un proceso coopera con otros procesos usando canales de comunicación. Una colección de procesos occam forma a su vez un proceso, de esta forma una jerarquía de procesos puede ser construida para modelar procesos reales.

De acuerdo con lo anterior el modelo de proceso occam puede ser mapeado eficientemente en un arreglo de transputers, cada uno con su memoria local, comunicándose a través de links, ver Figura 4. Finalmente, no obstante su profunda asociación con el transputer, occam es un lenguaje de propósito general que puede ser implementado en otras arquitecturas. El transputer es considerado aquí como un vehículo eficiente para implementar este lenguaje.

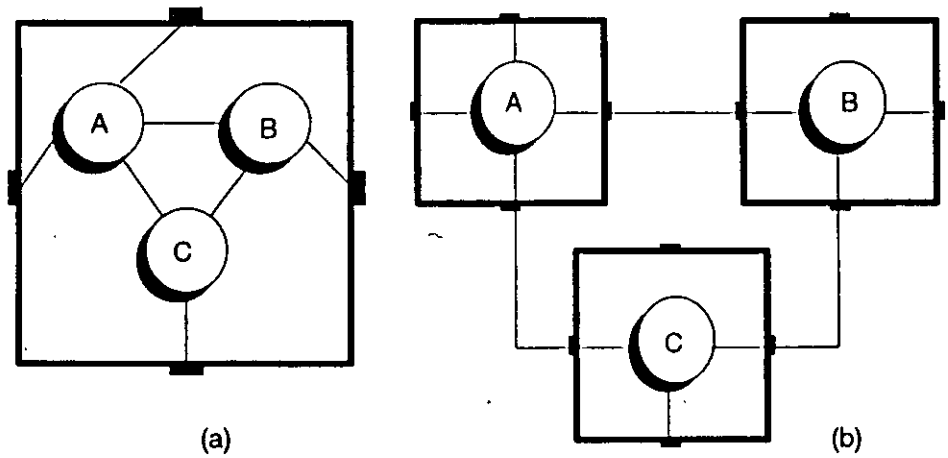


Figura 4. Procesos Occam: (a) en un Transputer; (b) en varios Transputers.

3.1 Arquitectura del Transputer

El transputer comprende una familia de microcomputadoras implementadas a nivel de VLSI, que incluyen, en un mismo circuito integrado, un procesador, memoria y links de comunicación, para interconectarse con otros transputers. La versión T800 del transputer es mostrada en la Figura 5.

El transputer T800 consta de un procesador RISC de 32 bits (15 MIPS a 30 MHz), unidad de punto flotante de 64 bits (2.25 Mflops a 30 MHz), 4 Kbytes de memoria local RAM, una interfase configurable para accesar memoria externa y cuatro links de comunicación, que operan a velocidad máxima de 20 Mbits/s [12]. Además de las características típicas del transputer, relativas a integración, velocidad de operación e interfases, otro aspecto atractivo es su soporte de concurrencia. El desarrollo y evaluación de una aplicación, como un número de procesos concurrentes, puede llevarse a cabo enteramente en un solo transputer. Después, los procesos componentes de la misma aplicación pueden ser distribuidos para operar realmente en paralelo en una red de transputers. Lo anterior requiere de instrucciones occam adicionales de configuración o mapeo que asocian cada proceso a ser ejecutado con su correspondiente transputer.

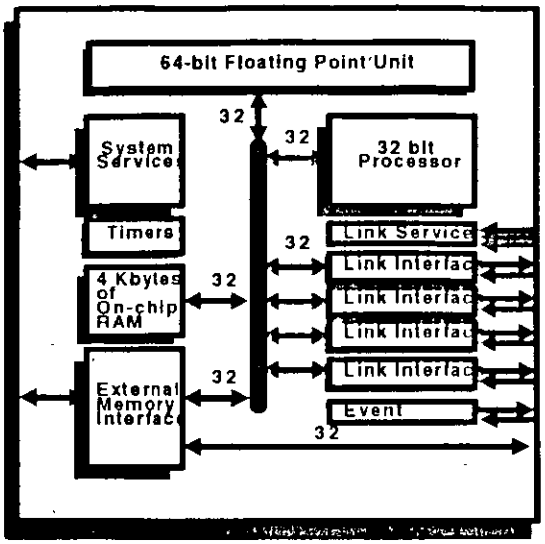


Figura 5. Arquitectura del Transputer T800.

En la implementación de aplicaciones en tiempo real, generalmente el diseñador es responsable de este mapeo físico de procesos en procesadores. Aunque existen sistemas operativos que realizan el mapeo de aplicaciones en forma transparente al usuario, estos generalmente introducen una mayor carga computacional al sistema, siendo sólo adecuados en aplicaciones fuera de línea [13]. Diferentes topologías han sido utilizadas en aplicaciones de sistemas de transputers. Estas topologías incluyen procesadores en línea, anillo, árbol, arreglos de 2 dimensiones, hipercubos, etc. La Figura 6 muestra algunos de estos arreglos.

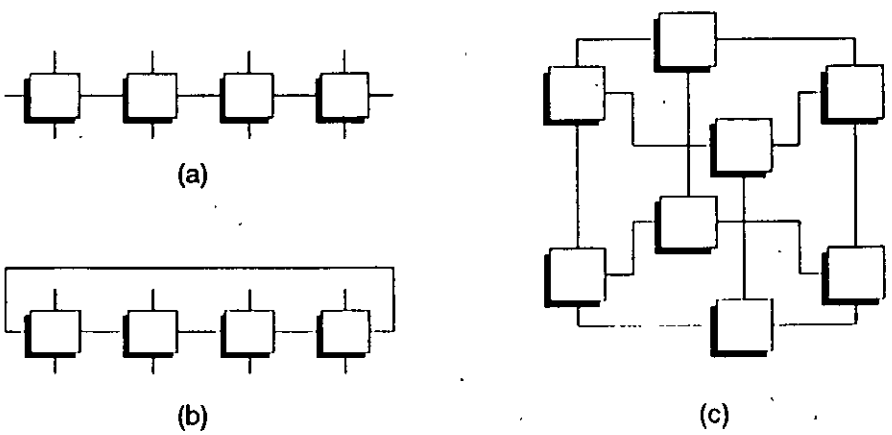


Figura 6. Topologías de Transputers: (a) Línea; (b) Anillo; (c) Hipercubo.

La nueva generación de transputers, serie T9000, actualmente en desarrollo, brindará un importante incremento en el desempeño de sistemas de procesamiento paralelo. Esta serie ofrece un arquitectura basada en procesadores "pipeline" (150 MIPS y 20 MFLOPS a 50 MHz), 16 Kbytes de memoria local y un nuevo sistema de comunicación operando a 100 Mbits/s [15]. Un aspecto también importante es que sus instrucciones son compatible con previas series, garantizando compatibilidad con sistemas de transputers actuales.

3.2 El Lenguaje Occam

Transputers pueden ser programados en lenguajes de alto nivel (C, FORTRAN ó PASCAL), sin embargo cuando es necesario explotar concurrencia y ganar mayor beneficio de su arquitectura, occam es recomendado, ya que provee muchas de las ventajas de un lenguaje de alto nivel y al mismo tiempo una alta eficiencia (comparable con la de un ensamblador). Occam es un lenguaje de programación diseñado para manejar concurrencia [16]. Esto es relevante sobre todo en aplicaciones prácticas de paralelo, ya que permite expresar un sistema en términos de procesos concurrentes que se comunican entre sí, lo cual da una estructura simple y clara. Además, ofrece la alternativa de ejecutar todos los procesos en un solo transputer o distribuirlos en un número de éstos [17].

La unidad básica de programación en occam es el "proceso", un proceso empieza, realiza una serie de acciones y se detiene o termina. Este concepto puede parecer similar a otros conceptos de programación convencional, excepto que en occam puede haber mas de un proceso ejecutándose al mismo tiempo y tratando de comunicarse con algún otro a través de sus canales. Los procesos en occam están constituidos por tres procesos primitivos:

asignación	v := e ;	asigna el valor de la variable **e** a la variable **v**,
entrada	c ? x ;	recibe un valor en el canal **c**, lo asigna a la variable **x** , y
salida	c ! e ;	transmite el valor de la expresión **e** a través del canal **c**.

Cuando un comando de salida es encontrado en un proceso, este proceso se mantiene en ese estado hasta que otro proceso ejecuta el comando de entrada correspondiente. De tal forma que la comunicación se lleve a cabo en el canal común utilizado por ambos procesos. Del mismo modo un comando de entrada no puede ser ejecutado, hasta que el correspondiente comando de salida es alcanzado en el proceso contraparte. De esta forma se asegura la sincronización. Varios procesos primitivos pueden ser combinados en uno más grande y formar un *constructor*, que es a su vez un proceso y puede ser usado como componente de otro constructor. Estos son básicamente:

SEQ -constructor secuencial
PAR -constructor paralelo, y
ALT -constructor alternativo

Constructores típicos tales como **IF** y **WHILE** son también soportados. Como occam es un lenguaje de programación concurrente, ejecución secuencial, paralela o alternativa debe ser especificada. El constructor **SEQ** indica que las declaraciones incluidas dentro del proceso serán ejecutadas en

de programación concurrente, ejecución secuencial, paralela o alternativa debe ser especificada. El constructor **SEQ** indica que las declaraciones incluidas dentro del proceso serán ejecutadas en secuencia. En el siguiente proceso, una señal medida es leída en el canal ADC, escalada y el resultado es enviado al canal DAC:

```
SEQ
  ADC ? señal.mediada
  señal.escalada := señal.escalada * factor
  DAC ! señal.escalada
```

Nótese que existe una sangría entre el constructor y sus procesos componentes. Figura 7 ilustra la relación entre este proceso y sus canales.

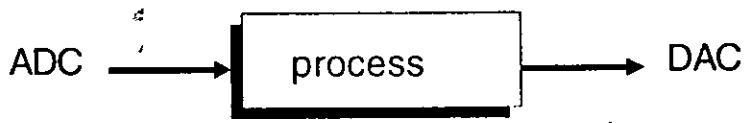


Figure 7. Ejemplo de Constructor Secuencial

El constructor **PAR** establece que los procesos incluidos serán ejecutados en paralelo, ver Figura 8. Tomando como procesos componentes dos procesos secuenciales:

```
PAR
  SEQ
    ADC1 ? señal.mediada
    señal.escalada := señal.escalada * factor
    DAC ! señal.escalada
  SEQ
    ADC2 ? señal.mediada
    señal.escalada := señal.escalada * factor
    DAC ! señal.escalada
```

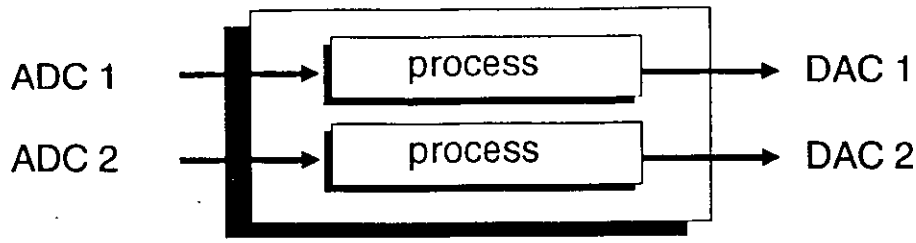


Figure 8. Ejemplo de Constructor Paralelo

El constructor **ALT** es usado para realizar la ejecución de procesos alternativos mediante la activación de su canal correspondiente. En el ejemplo: este constructor espera hasta que alguno de los canales de entrada **canal1**, **canal2** ó **canal3** es activado para realizar el proceso correspondiente. El constructor ejecuta el solamente el proceso asociado con la primera entrada que se lleve a cabo y finalmente termina. Ejemplo:

```
ALT
 canal1 ? x
  ... proceso 1
 canal2 ? x
  ... proceso 2
 canal3 ? x
  ... proceso 3
```

Nótese que hemos usado la convención **... proceso** para denotar el proceso que encierra esa declaración y expresar en forma mas clara el constructor. El constructor **ALT** ofrece un método formal de alto nivel para manejar eventos internos y externos que en procesadores convencionales son manejados como interrupciones a nivel de ensamblador.

3.3 Programación en Tiempo Real

Prioridades son fundamentales en aplicaciones en tiempo real. Aunque occam ofrece soporte para establecer y asignar diferentes niveles de prioridad en la ejecución de procesos, en su implementación en el transputer solo existen dos niveles de prioridad (alta y baja), esto ha resultado insuficiente en diversos tipos aplicaciones de tiempo real. sin embargo se han desarrollado técnicas a nivel de programación que resuelven este problema a un costo computacional mínimo [18].

Occam además ofrece una manera directa de asignar procesos lógicos a procesadores físicos. El número de procesadores puede no corresponder al número de procesos, ya que es posible asignar varios procesos a un solo procesador. Es de hecho una buena práctica el descomponer el problema de tal forma que resulten más procesos que procesadores. Ya que es generalmente mas fácil agrupar procesos, que subdividirlos una vez que el diseño del sistema se ha establecido.

Existen dos formas fundamentales de asignar actividades a procesadores: estática y dinámica. En una asignación estática la asociación de actividades a el procesador es fija y se lleva a cabo antes de ejecutar el programa, mientras que en una asignacion dinámica de actividades, éstas son distribuidas durante la ejecución del programa, de acuerdo a un número de criterios, tales como disponibilidad de procesadores, prioridades y dependencia de actividades.

La asignación dinámica de actividades ofrece un mayor potencial para una óptima utilización de procesadores, pero al mismo tiempo produce una disminución en el desempeño del sistema asociado con el incremento en el nivel de comunicaciones y el costo computacional del software de asignación y destribución. Lo cual puede ser inaceptable en cierto tipo de aplicaciones en tiempo real, en donde una asignación estática de actividades puede ofrecer un mejor desempeño.

BIBLIOGRAFIA

1. Kuck, D.J., "High Performance Computing", Oxford University Press, 1996.

2. García Nocetti, D.F. and Fleming, P.J. "Parallel Processing in Digital Control- Advances in Industrial Control", Springer-Verlag, 1992.

3. Stone, H.S., "High Performance Computer Architectures", Addison Wesley, 1987.

4 Krishnamurthy, E.V., "Parallel Processing Principles and Practice", Addison Wesley, 1989.

5. Thoeni, U., "Programming Real-Time Multicomputers for Signal Processing", Prentice Hall, 1994.

6. Lewis, T. G., "Introduction to Parallel Computing", Prentice-Hall International, 1992.

7. Metropolis, N. and Rota, G.C. "A new Era in Computation", MIT Press, 1993.

8. Harp, G., "Transputer Applications", Computer Systems Series, Pitman, 1989.

9. Tzafestas, S. "Parallel and Distributed Computing in Engineering Systems", North-Holland, 1991.

10. Flynn, M.F., "Some Computer Organisations and Their Effectiveness", IEEE Trans Comput., C-21, 1972, pp. 948-960.

11. INMOS Limited, "Transputer Overview", The Transputer Databook, Second edition, 1989.

12. INMOS Limited, "Transputer T800", The Transputer Databook, Second edition, 1989.

13. Kim, T., "Operating Systems", in Transputer Applications (Ed. Harp, G.), Computer Systems Series, Pitman 1989.

14. May, M.D., Thompson, P.W., Welch, P.H., "Networks, Routers and Transputers -Function, Performance and Applications; IOS Press, 1993.

15. Dyson C., "The Next Generation Transputer", Technical Note, Inmos Bristol, July 1990.

16. Inmos Ltd, "Occam 2 Reference Manual", Prentice Hall, 1987.

17. Jones, G., Goldsmith, M., "Programming in Occam 2", Prentice Hall, 1988.

18. Welch, P.H. "Multi-Priority Schedulers for Transputer-based Real-time Control", Real-Time Systems with Transputers, IOS Press, 1990.

2

Estructuras de Programación Paralela

Chapter 11

Approaches to writing parallel programs in occam

The previous chapters have concentrated on the various program constructs available in occam and have explained their use. In order to reap the most benefit from the use of occam and transputers, programs need to exhibit some degree of parallelism. Every application needs to be considered to see how the inherent parallelism may be best expressed and exploited. But this is not an easy exercise. The conventional mould of thinking and expressing the solution to a programming problem as a *sequence* of steps must be broken. A number of approaches for applying parallelism to problems have yielded promising results for occam and the transputer. However much more experience is needed in the art and science of parallel programming.

Parallelism is expressed within an occam program by parallel constructions. Given the fact that each parallel process may be mapped on to a separate processor for execution, the potential benefits in terms of speed and efficiency may be enormous. Nevertheless, efficiency considerations need to be taken into account so that the benefits gained from parallelism are not lost.

The granularity of the parallelism in the application and how it is distributed over a transputer network need to be carefully assessed. Granularity is a measure of parallelism - the number of parallel processes in an application. A large amount of parallelism is not necessarily an ideal situation. It is necessary to achieve a balance between keeping each processor busy with computation and maintaining inter-process communication time at a minimum. The current versions of the transputer do not provide message routing in hardware; valuable computation time can be taken up in providing extra occam processes to implement software message routing between processes. With a too fine-grained granularity, there will be many parallel processes and then the communications overhead can dominate the computation. However in coarse-grained granularity, with not so many parallel processes, the computation should dominate the communications.

The organisation of programs around the parallel paradigm has been approached in a number of different ways. These various approaches may be broadly classified into the following categories

- algorithmic - quasi-independent tasks which execute sections of the problem solution algorithm (and are therefore non-identical), data and computed results being passed among the tasks as the algorithm dictates.

- geometric - quasi-independent but identical tasks which process a portion of the data, and which interact with (or are affected by) neighbouring tasks, according to the geometry of the problem.

- process farming - fully independent but identical tasks which process the data in any order.

Analysis of the problem using data flow diagrams presents a graphical method for visualising the component processes of a program - the processes are represented by circles, the channels are represented by arcs connecting the circles. Such a method has been found useful in the design of occam programs [Kerridge]. Another graphical method which may also prove useful for designing occam programs is the Petri-net notation [Peterson]. This technique allows the graphical expression of parallel strands of computation, so that its suitability to the design of occam programs should be apparent. CSP [Hoare], the theoretical basis for occam, is an excellent tool for expressing the design of occam programs.

The following sections of the chapter discuss in more detail the three approaches to writing parallel programs in occam, giving a concrete example in each case. For simplicity of presentation, the handling of process termination has been omitted in the first two examples. Realistically, this should always be provided - see Chapter 6 for an example how this might be achieved. The third example contains code for process termination.

11.1 Algorithmic parallelism

The algorithmic approach, also known as data flow decomposition, is concerned with injecting parallelism into the algorithm being used to solve the problem at hand. The algorithm may be an already existing sequential one or totally new. Parallelism can be introduced by considering how the algorithm may be broken up into separate, quasi-independent sections. Each section can then be executed in parallel, with data flowing between the sections as necessary. Each section will perform some computation with the data, and then pass the data on to the next section.

The inherent parallelism is frequently found in a loop or iteration. Consider a linear search for example. In the sequential case, each item in the list is compared one at a time as the search sequences through the list. This comparison may however be performed more efficiently in parallel - each comparison may be performed at the same time.

The algorithmic approach is modelled by occam parallel processes, each parallel process responsible for the execution of a section of the algorithm, using the synchronisation and communication provided by occam channels to transfer data between the processes. The communication overhead between the parallel processes in such circumstances can become

quite significant.

A common example of the algorithmic approach is the pipeline - each unit of the pipeline contributes by executing a section of the algorithm. The independent units may operate on separate portions of the data as the stream of data is fed down the pipeline. The overall effect of this overlapped operation is the realisation of parallel execution.

Such organisation is not limited to one-dimensional cases. For example, a systolic array is effectively a two-dimensional pipeline that may be used to great effect for the parallel execution of matrix operations [Jones and Goldsmith].

As an example of the algorithmic approach, consider the Newton-Raphson estimate technique for evaluating square-roots. This method starts off with the value of the number whose square-root is required and an initial estimate of the square-root. The Newton-Raphson formula is then applied in an iterative manner, each time producing a better estimate of the square-root from the previous value. If the iteration is performed a large number of times, the final estimate will be a close approximation to the real value.

The Newton-Raphson formula for calculating the square root of a number x is

$$y_{i+1} = \frac{1}{2}(y_i + \frac{x}{y_i})$$

where y_i, y_{i+1} are successive estimations of the root y.

This formula may be expressed in occam as

```
Estimate := (Estimate + (Number / Estimate)) / 2.0 (REAL32)
```

where *Number* is the number whose square root is required and the initial value for *Estimate* is given. Usually, the initial value for *Estimate* is taken to be

```
Number / 2.0 (REAL32)
```

If the iteration is performed *Iterations* times, then a sequential solution may be written as

```
SEQ
  Input ? Number
  Estimate := Number / 2.0 (REAL32)
  SEQ Index = 0 FOR Iterations
    Estimate := (Estimate + (Number / Estimate)) / 2.0 (REAL32)
  Output ! Estimate
```

Written in occam, this becomes

```
PROC Pipeline (CHAN OF REAL32 InPipe, OutPipe)
  WHILE TRUE
    REAL32 Number, Estimate :
    SEQ
      -- accept number and previous estimate
      InPipe ? Number
      InPipe ? Estimate
      -- pass on number and new estimate
      OutPipe ! Number
      OutPipe ! (Estimate + (Number / Estimate)) / 2.0 (REAL32)
:
```

Finally, the *Terminate* process inputs the number and final estimate of the square root from the last pipeline process and outputs this value. Writing this in pseudo-code

```
WHILE data is available
  SEQ
    RECEIVE a number and final square root estimate from last pipeline
    process
    OUTPUT this number and final estimate as results
```

In occam, this becomes

```
PROC Terminate (CHAN OF REAL32 Extract, Output)
  WHILE TRUE
    REAL32 Number, Root :
    SEQ
      -- extract number and root (final estimate) from pipeline
      Extract ? Number
      Extract ? Root
      -- output results
      Output ! Number
      Output ! Root
:
```

The main process will comprise a **PAR** construction containing instances of *Initialise*, *Pipeline* and *Terminate*. The *Pipeline* process is replicated the desired number of times.


```
PAR
  Initialise (InChan, Pipe [0])
  PAR Index = 0 FOR Iterations
    In IS Pipe [Index] :
    Out IS Pipe [Index + 1] :
    Pipeline (In, Out)
  Terminate (Pipe [Iterations], OutChan)
```

The amounts of computation required for the sequential and parallel solutions are the same. However, the benefit derived by expressing the sequential algorithm as a parallel one accrues only when there are many numbers requiring the calculation of their square roots. The partial estimates for these numbers may all be within the pipeline at the same time (depending on the length of the pipeline) - each pipeline process can be calculating a different partial estimate. The automatic synchronisation of occam ensures the correct order of communication and, hence, the correct order of computation.

Other examples of the use of this approach are sorting [Pountain and May], prime number generating [Burns], systolic array processing [Jones and Goldsmith], compiling and solid modelling [May and Shepherd].

11.2 Geometric parallelism

With this approach, parallelism is introduced by making use of any regular spatial geometry or structure present in the problem. Rather like a large cube may be divided up into a number of smaller constituent cubes, so the spatial geometry of the problem is divided up in some symmetrical fashion, assuming a uniform distribution of data over the geometry, to allow a more tractable solution to be expressed. Each of these small units acts as a quasi-independent entity, responsible for the data in its own spatial region. The computation performed by each small unit is summed to give an overall effect. Interactions between neighbouring units may be incorporated to give a more realistic solution. This approach is also known as data structure decomposition.

Each small unit is modelled by an identical occam parallel process, each parallel process operating on the data relevant to its own domain. Interactions between nearest neighbours may be introduced with occam channels connecting the neighbouring units. The communications overhead between these communicating processes may become quite appreciable.

An example of the geometric approach is its use in the simulation of thermal conduction in a two-dimensional rectangular metal plate which is being heated by a heat source at a certain point. Simulation of thermal conduction over the whole plate is difficult. So, to simplify the problem, the geometry of the situation is utilised and the plate is subdi into a number of

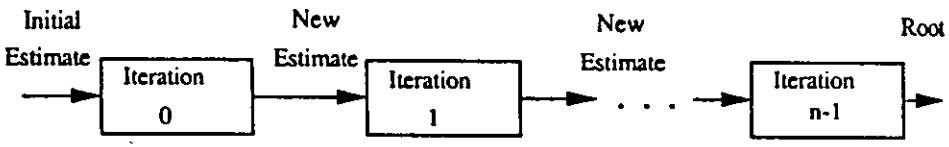


Figure 11.1 A pipeline of processes to calculate the square root of a number using the Newton-Raphson method

The Newton-Raphson procedure may be written in a concurrent form by considering each iteration as an occam parallel process. Each of these processes accepts the number and previous estimate as input, calculates the new estimate and produces the number and new estimate as output (Figure 11.1). Thus the approximation technique may be written as a pipeline of identical processes with the necessary initialisation and termination processes. The pipeline may be generated with a replicated PAR statement. The top level of the program will have the following form

```
global declarations
procedures comprising
  Initialisation process
  Pipeline processes
  Termination process
main process
```

The global declarations comprise the channels for the pipeline processes and the channels for the initialisation and termination processes. Assuming 50 iterations, this section may be written as

```
-- global declarations
VAL Iterations IS 50 :
[Iterations + 1] CHAN OF REAL32 Pipe :
CHAN OF REAL32 InChan, OutChan :
```


The *Initialise* process inputs the number whose square root is required, and outputs the value of this number and initial square root estimate to the first pipeline process. This may be expressed in pseudo-code as

```
WHILE data is available
  SEQ
    INPUT a number
    SEND this number and initial square root estimate to first pipeline
    process
```

In occam, this may be written as

```
PROC Initialise (CHAN OF REAL32 Input, Inject)
  WHILE TRUE
    REAL32 Number :
    SEQ
      Input ? Number
      -- feed number and initial estimate into pipeline
      Inject ! Number
      Inject ! Number / 2.0 (REAL32)
:
```

As indicated previously, the pipeline comprises *Iterations* identical processes which are generated with a PAR replicator. Each of these processes inputs the number and the previous estimate from the preceding pipeline process and outputs the number and new estimate to the succeeding process. Expressed in pseudo-code, this becomes

```
WHILE data is available
  SEQ
    RECEIVE a number and previous square root estimate from preceding
    pipeline process
    SEND this number and new square root estimate to next pipeline process
```

rectangular areas - these areas being the quasi-independent units which will be represented by occam processes. The heat conduction i.e. temperature of each of these areas may be estimated and summed to give an approximate effect for the heat conduction over the whole plate. The temperature of each area will depend on that of its surroundings i.e. the neighbouring areas. It is assumed that one of the areas contains the heat source.

For the example, consider a metal plate, n by m units (Figure 11.2). The program is required to monitor the temperature at the centre of each of these areas. Also, for the example, consider that two boundaries (top and left-hand side) of the plate are adjacent to an infinite heat sink and that the other two boundaries (bottom and right-hand side) of the plate are adjacent to a perfect heat insulator.

The simulation program will comprise a set of identical parallel processes, each responsible for determining the temperature of one of the areas of the metal plate. This temperature is taken to be an average of the temperatures of the four neighbouring areas. In addition areas on the boundaries of the plate will be affected by the type of adjacent boundary - the heat sink will maintain a constant (base) temperature and the heat insulator will reflect the temperature of the boundary areas. These boundary effects will be simulated by extra parallel processes.

Each plate area will have nine channels - an input and output channel for each of the neighbouring areas, up, down, left and right plus a result channel (Figure 11.3). The result channel communicates with a monitor process to display the current temperature of each area on the screen.

There will be n * m processes for calculating the areas' temperatures plus n + m processes for each of the two different boundary effects. This number of processes may be generated by suitable replication of the following processes

Sink - simulate the effect of a heat sink
Insulator - simulate the effect of a heat insulator
Source - simulate the effect of a heat source
CalcTemp - calculate the temperature of an area. Account must be taken of the fact that one of the areas will contain the heat source.

The top level of the program will have the following form

```
global declarations
procedures comprising
  Sink
  Insulator
  Source
  CalcTemp
main process
```

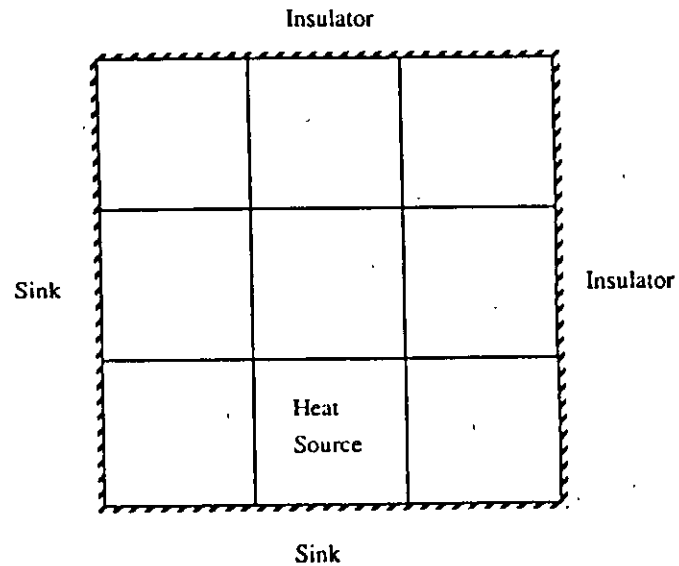


Figure 11.2 Metal plate subdivided into smaller areas

Assuming a metal plate of dimension 3 by 3 units, the global declarations are as follows

```
VAL Height IS 3 :
VAL Width IS 3 :
VAL TwiceHeight IS 2 * Height :
VAL TwiceWidth IS 2 * Width :
VAL Rectangle IS Height * Width :
VAL TwiceRectangle IS 2 * Rectangle :
VAL SourceX IS 1 :
VAL SourceY IS 2 :
VAL BaseTemp IS 50.0 (REAL32) :
[TwiceRectangle + TwiceHeight] CHAN OF REAL32 Horizontal :
[TwiceRectangle + TwiceWidth] CHAN OF REAL32 Vertical :
[Rectangle] CHAN OF REAL32 Result :
```

The initial temperature, *BaseTemp*, is assumed to be 50 degrees. The position of the area containing the heat source at its centre is given by *SourceX* and *SourceY*. The areas communicate via the channels *Horizontal*, *Vertical* and *Result*. The horizontal channels are numbered in right/left pairs down the columns of the array of areas. The vertical channels are numbered in down/up pairs along the rows of the array. The result channels are numbered along the rows (see Figure 11.4 for an example of the 3 by 3 array).

The *Sink* process is simulated by maintaining (outputting) a constant temperature, regardless of the adjacent temperature - the temperature input is disregarded. This constant temperature is taken to be the initial temperature of the plate. Expressing this in pseudo-code gives

```
WHILE simulation is required
  PAR
    RECEIVE a temperature of an adjacent area and IGNORE
    SEND a constant (base) temperature back to the adjacent area
```

Written as occam code, this gives

```
PROC Sink (CHAN OF REAL32 In, Out)
  -- left or bottom boundary
  WHILE TRUE
    REAL32 Any :
    PAR
      -- ignore adjacent temperature
      In ? Any
      -- output a steady temperature
      Out ! BaseTemp
```

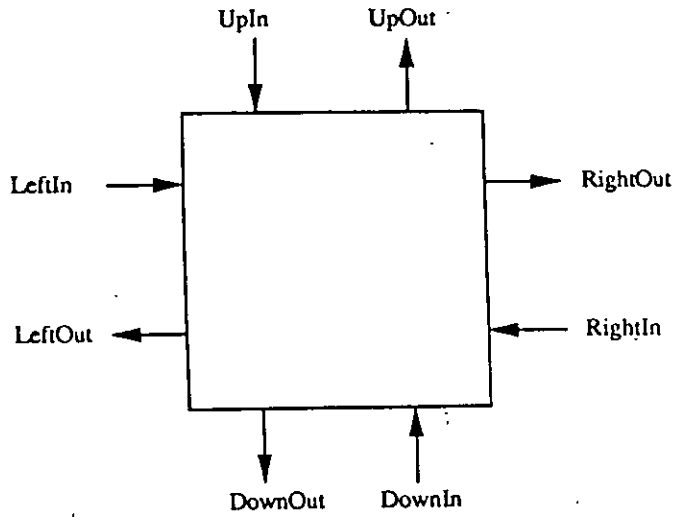


Figure 11.3 Channels of a plate area

The *Insulator* process does not allow any heat to escape - the temperature read from the adjacent area is returned. Written in pseudo-code, this is

```
WHILE simulation is required
  SEQ
    RECEIVE a temperature from an adjacent area
    SEND the temperature received back to the adjacent area
```

Rewriting in occam gives

```
PROC Insulator (CHAN OF REAL32 In, Out)
  -- top or right boundary
  WHILE TRUE
    REAL32 Temp :
    SEQ
      -- input temperature of adjacent area
      In ? Temp
      -- return last temperature read
      Out ! Temp
:
```

Process *Source* simulates a heat source by generating a temperature which increases at a steady rate - one degree higher than the previous value. This temperature is transmitted to the surrounding areas, ignoring the present temperature of these areas. The heat source will be surrounded by four neighbouring areas. Expressing these requirements in pseudo-code gives

```
WHILE simulation is required
  SEQ
    PAR
      RECEIVE a temperature from four surrounding areas and IGNORE
      SEND new temperature back to four surrounding areas
    INCREMENT temperature by one degree
```

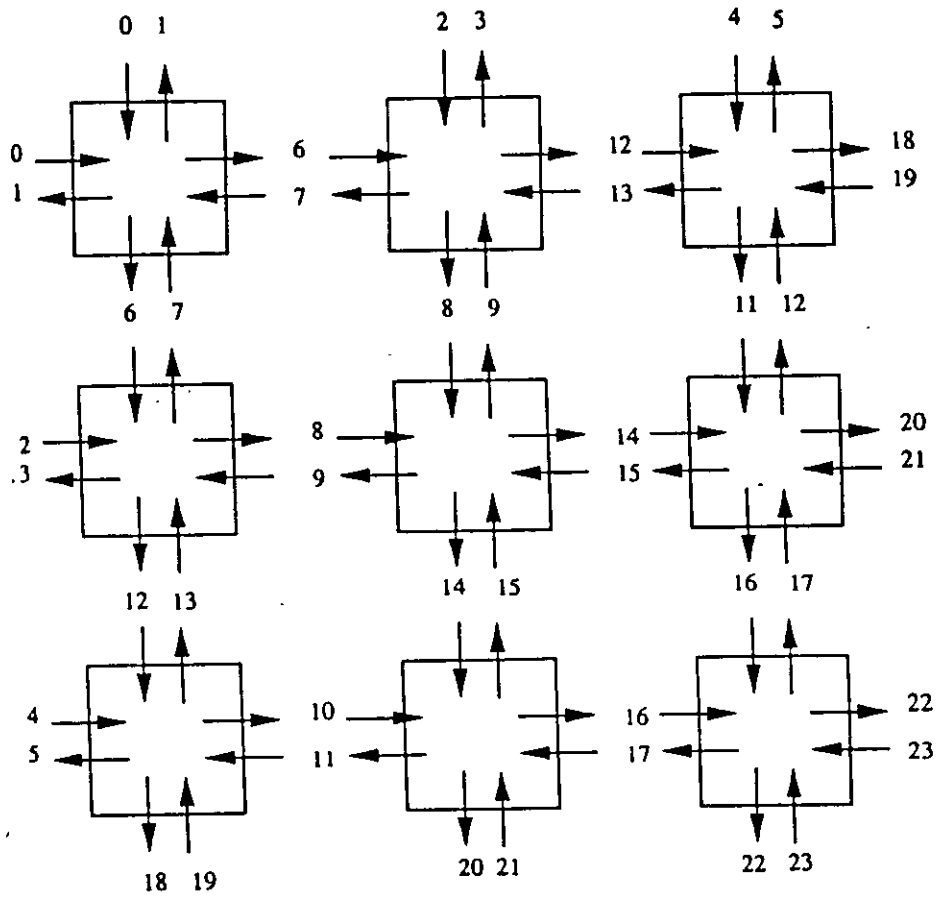


Figure 11.4 The areas and channels for a 3 by 3 array

In occam, this may be written

```
PROC Source (CHAN OF REAL32 UpIn, DownIn, LeftIn, RightIn,
                            UpOut, DownOut, LeftOut, RightOut,
                            Result)
  VAL TempIncrease IS 1.0 (REAL32) :
  REAL32 Temp :
  SEQ
    -- initial condition
    Temp := BaseTemp
    WHILE TRUE
      -- output new temperature to four surrounding areas, ignoring any inputs
      SEQ
        PAR
          REAL32 Any :
          PAR
            -- area below
            DownIn ? Any
            DownOut ! Temp
          REAL32 Any :
          PAR
            -- area to left
            LeftIn ? Any
            LeftOut ! Temp
          REAL32 Any :
          PAR
            -- area above
            UpIn ? Any
            UpOut ! Temp
          REAL32 Any :
          PAR
            -- area to right
            RightIn ? Any
            RightOut ! Temp
        Result ! Temp    -- output new temperature to Monitor
        Temp := Temp + TempIncrease    -- increase temperature of source
:
```

(The interactions with each adjacent area have been grouped in separate PAR constructions for clarity.)

Process *CalcTemp* will calculate the rise in temperature of each area due to the temperature of the neighbouring areas. Account must be taken of the fact that one area will contain the heat source. Written in pseudo-code, this is

```
IF area = heat source THEN
  GENERATE temperature rise
ELSE
  WHILE simulation is required
    SEQ
      PAR
        RECEIVE the temperature of four surrounding areas
        SEND the temperature of this area to four surrounding areas
      CALCULATE new temperature of this area, based on temperature
      rises of surrounding areas
```

Expressing this pseudo-code in occam,

```
PROC CalcTemp (BOOL HotSpot,
               CHAN OF REAL32 UpIn, DownIn, LeftIn, RightIn,
                             UpOut, DownOut, LeftOut, RightOut,
                             Result)
  IF
    -- if area contains heat source
    HotSpot
      -- generate temperature rise
      Source (UpIn, DownIn, LeftIn, RightIn,
              UpOut, DownOut, LeftOut, RightOut, Result)
    TRUE
      -- area does not contain heat source
      REAL32 Temp :
      SEQ
        -- initial conditions
        Temp := BaseTemp
```

```
WHILE TRUE
  REAL32 SumOfTemps, MeanTemp,
         UpTemp, DownTemp, LeftTemp, RightTemp,
         DeltaUp, DeltaDown, DeltaLeft, DeltaRight :
  SEQ
    -- interact with neighbouring areas
    PAR
      -- area below
      PAR
        DownIn ? DownTemp
        DownOut ! Temp
      -- area to the left
      PAR
        LeftIn ? LeftTemp
        LeftOut ! Temp
      -- area above
      PAR
        UpIn ? UpTemp
        UpOut ! Temp
      -- area to the right
      PAR
        RightIn ? RightTemp
        RightOut ! Temp
    DeltaDown := DownTemp - Temp
    DeltaLeft := LeftTemp - Temp
    DeltaUp := UpTemp - Temp
    DeltaRight := RightTemp - Temp
    -- now average these temperatures to find mean rise
    SumOfTemps := ((DeltaUp + DeltaDown)
                   + DeltaLeft) + DeltaRight
    MeanTemp := SumOfTemps / 4.0 (REAL32)
    -- increase temperature by half average temperature rise
    Temp := Temp + (MeanTemp / 2.0 (REAL32))
    -- output the result to Monitor
    Result ! Temp
```


Process *Monitor* will be responsible for keeping a record of the temperature of each area, and displaying this temperature on the screen. Expressing this in pseudo-code gives

```
WHILE simulation is required
  SEQ
    PAR
      INPUT temperature from each area
    SEQ
      IF temperature <> last temperature from each area THEN
        DISPLAY temperature
```

Writing this in occam gives

```
PROC Monitor ([ ] CHAN OF REAL32 Result)
  [Rectangle] REAL32 LastTemp :
  SEQ
    -- initialise array holding temperatures
    SEQ Index = 0 FOR Rectangle
      LastTemp [Index] := 0.0 (REAL32)
    [Rectangle] REAL32 Temp :
    WHILE TRUE
      SEQ
        -- input temperature of areas
        PAR Index = 0 FOR Rectangle
          Result [Index] ? Temp [Index]
        SEQ Index = 0 FOR Rectangle
          -- check for a temperature change
          IF
            Temp [Index] <> LastTemp [Index]
              INT Row, Col :
              SEQ
                -- display new temperature
                Row := Index / Width
                Col := Index REM Width
                Display ! Row ; Col ; Temp [Index]
                LastTemp [Index] := Temp [Index]
            TRUE
              SKIP
```

The overall structure of the main process will be an outer PAR enclosing the requisite number of instances of *CalcTemp*, *Sink* and *Insulator* processes. In addition there will be an instance of the *Monitor* process. Thus the main process may be expressed in pseudo-occam as

```
PAR
  Monitor process
  n * m  CalcTemp processes
  n + m  Sink processes
  n + m  Insulator processes
```

This process may be rewritten in terms of nested replicated PARs (assuming n rows by m columns of rectangular areas - the index *Row* moving from top to bottom, the index *Column* moving from left to right) as follows

```
PAR
  Monitor process

  PAR Row = 0 FOR n  -- left-hand side areas
    Sink process

  PAR Column = 0 FOR m
    PAR
      -- top side areas
      Insulator process

      -- middle areas
      PAR Row = 0 FOR n
        CalcTemp process

      -- bottom side areas
      Sink process

  PAR Row = 0 FOR n  -- right-hand side areas
    Insulator process
```


Using the (more meaningful) constants Height (for n) and Width (for m) as specified in the global declarations, this may be rewritten as

```
PAR
  Monitor (Result)

  PAR Row = 0 FOR Height  -- left-hand side areas
    VAL Out IS Row + Row :
    VAL In IS Out + 1 :
    Sink (Horizontal [In], Horizontal [Out])

  PAR Col = 0 FOR Width
    PAR
      -- top side areas
      VAL Out IS Col + Col :
      VAL In IS Out + 1 :
      Insulator (Vertical [In], Vertical [Out])
      -- middle areas
      PAR Row = 0 FOR Height
        VAL Up IS ((TwiceWidth * Row) + Col) + Col :
        VAL Down IS ((TwiceWidth * (Row + 1)) + Col) + Col :
        VAL Left IS ((TwiceHeight * Col) + Row) + Row :
        VAL Right IS ((TwiceHeight * (Col + 1)) + Row) + Row :
        VAL Hot IS (Row = SourceY) AND (Col = SourceX) :
        SEQ
          CalcTemp (Hot,
                    Vertical [Up],
                    Vertical [Down +1],
                    Horizontal [Left],
                    Horizontal [Right + 1],
                    Vertical [Up + 1],
                    Vertical [Down],
                    Horizontal [Left + 1],
                    Horizontal [Right],
                    Result [(Width * Row) + Col])
```

```
-- bottom side areas
VAL In IS (TwiceRectangle + Col) + Col :
VAL Out IS In + 1 :
Sink (Vertical [In], Vertical [Out])

PAR Row = 0 FOR Height   -- right-hand side areas
  VAL In IS (TwiceRectangle + Row) + Row :
  VAL Out IS In + 1 :
  Insulator (Horizontal [In], Horizontal [Out])
```

Other examples of the use of this approach are the modelling of a statistical "spin" system as may be found in liquid crystal films [Askew *et al.*].

11.3 Process farming

The farm approach is applicable to problems whose solution will succumb to a decomposition into many smaller parts and where these parts are independent of each other. As the parts are independent, each may be executed concurrently, in isolation, and the effect summed to give a solution to the whole problem. The solution is analogous to a farmer supervising the toil of many farm workers, each worker performing any given task in isolation from the other workers - hence the name.

A farm is modelled by a set of occam processes. One process is nominated the farmer The farmer process controls the organisation and allocation of work. The controlling process farms or hands out work to its subordinate worker processes. The worker processes are modelled as identical parallel occam processes. As and when each worker process completes the given task, the farmer process will issue further work for completion. Thus a farm of worker processes toil away on parts of the problem, finishing one task and starting another, until the whole problem is complete. Typically little inter-process communication is needed in such applications. However, depending on the number of worker processes and their configuration, for example whether they are organised in a linear or tree fashion, the routing of messages between the farmer and workers may well cause communications overhead problems.

An example of the process farm approach is its use in producing a graphical representation of the Mandelbrot set, or more exactly, a graphical representation of those points which lie within and without the Mandelbrot set [Barnsley *et al.*, Peitgen and Richter]. This set comprises all complex numbers, $c = a + ib$, for which the recurrence relation

$$z_{n+1} = z_n^2 + c \quad \text{for } n = 0, 1, 2, \ldots \qquad (11.1)$$

converges to a finite complex number (where z_n and z_{n+1} are complex numbers computed in

successive iterations of the recurrence relation, and $z_0 = 0$ is the initial condition). It can be demonstrated that, if for some n,

$$|z_n| > 2$$

then the iteration diverges and hence c does not belong to the Mandelbrot set.

In practice, the iteration is performed a given number of times, m, and c is considered to belong to the Mandelbrot set, M, if

$$|z_n| < 2 \quad \text{for all } n \leq m \qquad (11.2)$$

The graphical display of the members of M produces quite vivid and intriguing self-similar shapes known as *fractals*.

For display purposes, the complex number, $c = a + ib$, is taken to be a graphics screen pixel with coordinates (a,b) - the graphics screen representing the complex plane. For every screen pixel the recurrence relation is applied. If the pixel belongs to the Mandelbrot set, it is coloured black, otherwise it is allocated a colour from the graphics palette which is graded according to the speed at which the iteration diverges i.e. the smallest natural number $n < m$ for which $|z_n| \geq 2$.

Such computation is quite intensive for a suitable number of iterations and, depending on the size of the graphics screen and hence the number of pixels, needs to be performed a large number of times. The actual computational task to be performed for each pixel is the same but the amount of computation will vary depending on whether or not the sequence of recurrence values for that pixel converges or diverges.

The general form of a farm in terms of pseudo-occam is as follows

```
global declarations
PAR
  Farmer process
  PAR Index = 0 FOR NumberOfWorkers
    Worker process
```

Each worker process accepts data from the farmer process, works with this data and then sends the result back to the farmer process, becoming available to accept more data. In the current context this work will be the calculation of the recurrence relation for the given data i.e. pixel coordinates (a,b).

Assuming a graphics area of 512 by 512 pixels, with 50 workers, the global declarations section may be written as

```
VAL NumberOfWorkers IS 50 :
VAL NumberOfPixels IS 512 * 512 :
PROTOCOL RAW
  CASE
    Data ; [2] INT
    Terminate
:
PROTOCOL PROCESSED
  CASE
    Results ; [2] INT ; INT
    Quit
:
[NumberOfWorkers + 1] CHAN OF RAW ToFarm :
[NumberOfWorkers + 1] CHAN OF PROCESSED FromFarm :
```

The channnels, *ToFarm* and *FromFarm*, allow the *Farmer* process to send data to the *Worker* processes and receive results from the *Worker* processes. The protocols, *RAW* and *PROCESSED*, will be explained shortly.

In practice, to improve the efficiency trade-off between computation and communications, each worker would be given a line of pixels as data. The processor overhead setting up a transmission over a transputer link is the same for many bytes as for a few bytes. (Once a data transfer has been initiated, the transfer of data over the link is automonous of the processor.) For simplicity, this example considers the data to be a single pixel. Also in practice, it may be advantageous to have a division of labour in the farmer process, having a farmer process proper and a separate graphics process. The function of the farmer process would be to hand out pixel coordinates to the worker processes, whilst that of the graphics process would be to accept the results (pixel coordinates and colour) and display them on the graphics screen.

Rather than allowing a worker process to idle whilst the farmer issues new work, the worker process may buffer an extra unit of work so that it may proceed immediately with this new work once it has completed the previous work. This scheme keeps the workers constantly busy [Packer].

Logically, each worker process may be connected via a channel to the farmer. Practically, since the transputer has only four links and if the workers are distributed over several transputers, there may be many tiers of worker processes. Because of this, each worker process will not just be concerned with the iteration of the recurrence relation. It will also act as a message switch, passing on data to processes further down the farm. The whole farm

process becomes self-regulating, message passing being synchronised by the occam input/output primitives. In addition to forwarding work to outlying workers, the worker process will gather results from these workers for onward delivery to the farmer (or graphics) process.

Each worker process will comprise three processes : *Switch*, *Feedback* and *Mandelbrot* (Figure 11.5). This arrangement may be expressed in pseudo-occam as

```
PROC Worker
  PRI PAR
    PAR
      Switch process
      Feedback process
    Mandelbrot process
:
```

This arrangement of processes in the PRI PAR construction ensures a high throughput for communications. This is important for processes which may use the transputer links, so that messages are transmitted without delay. If a high priority process was not used, the message would not be examined until the message switch was scheduled by the low priority round-robin scheduler of the transputer [May and Shepherd].

Tagged protocols, *RAW* and *PROCESSED*, are defined for the data which is sent to and received from the worker processes. The tag *Data* of protocol *RAW* corresponds to the transfer of two integers (a pixel), whilst the tag *Results* of protocol *PROCESSED* corresponds to the transfer of three integers (a pixel and its colour). In addition, each of these protocols has a tag which is used to pass a termination nonce to the participating processes at the end of the calculation.

The *Worker* process may now be rewritten with the addition of channels.

```
PROC Worker (CHAN OF RAW FromPrevious, ToNext,
             CHAN OF PROCESSED ToPrevious, FromNext)
  CHAN OF BYTE MoreWork :
  CHAN OF RAW Work :
  CHAN OF PROCESSED WorkDone :
  PRI PAR
    PAR
      Switch (MoreWork, FromPrevious, ToNext, Work)
      Feedback (ToPrevious, FromNext, WorkDone)
    Mandelbrot (MoreWork, Work, WorkDone)
:
```

The *Switch* process is responsible for accepting pixels (work) from the *Farmer* process, buffering a pixel for its *Mandelbrot* process, and forwarding excess work to workers further

down the farm. Expressing this in pseudo-code gives

```
WHILE pixels are available
  ALT
    RECEIVE request from mandelbrot for another pixel
      IF buffer = full THEN
        SEND buffered pixel to mandelbrot
      ELSE
        SET mandelbrot = idle
    RECEIVE pixel from farmer
      IF mandelbrot = idle THEN
        SEND pixel to mandelbrot
      ELSEIF buffer = empty
        BUFFER pixel
      ELSE
        SEND pixel to next worker
```

Such a structure with more than one input to react to may be conveniently programmed using an ALT construction. The actual code will be slightly more complicated than the above pseudo-code owing to the need to watch out for and pass on the termination notice to the *Mandelbrot* process and the next worker. This is just a matter of reacting to the relevant tag of the channel protocol. The occam for the process is

```
PROC Switch (CHAN OF BYTE MoreWork,
             CHAN OF RAW FromPrevious, ToNext, Work)
  BOOL Busy, Buffered, Running :
  SEQ
    Busy := FALSE
    Buffered := FALSE
    Running := TRUE
    WHILE Running OR Busy
      [2] INT Coords, BufferedCoords :
      BYTE Any :
      ALT
        -- mandelbrot requesting more work
        Busy & MoreWork ? Any
          IF
            Buffered  -- check for buffered work
              -- pass mandelbrot the buffered work
              SEQ
                Work ! Data ; BufferedCoords
                Buffered := FALSE
```

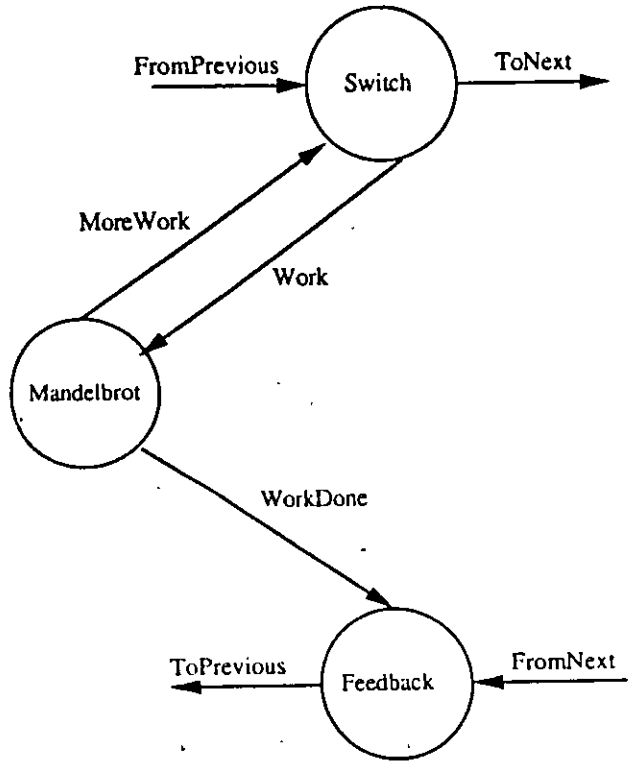


Figure 11.5 The component processes of a worker process

```
        TRUE    -- no buffered work
          Busy := FALSE
      -- a message from the farmer
      Running & FromPrevious ? CASE
        Data ; Coords -- another pixel
          IF
            NOT Busy     -- check if mandelbrot busy
              SEQ
                Work ! Data ; Coords
                Busy := TRUE
            NOT Buffered     -- check if pixel buffered
              SEQ
                BufferedCoords := Coords
                Buffered := TRUE
            TRUE -- pass on pixel
              ToNext ! Data ; Coords
        Terminate            -- termination notice
          Running := FALSE
  ToNext ! Terminate     -- pass on termination notice
  Work ! Terminate
:
```

In the above process, the boolean variable, *Running*, records whether or not a termination notice has been received from the farmer, whilst the boolean variable, *Busy*, marks whether or not the *Mandelbrot* process is processing a pixel. The *Mandelbrot* process performs the iteration procedure for a given pixel and assigns that pixel a colour dependent on the degree of convergence. With a little rearrangement the recurrence relation may be simplified for computation. Substituting $z = x + iy$, then equation (11.1) may be written as

$$x_{n+1} = x_n^2 - y_n^2 + a$$

and

$$y_{n+1} = 2x_n y_n + b$$

Since

$$| z_{n+1} | = \sqrt{x_{n+1}^2 + y_{n+1}^2}$$

the condition for Mandelbrot set occupancy - equation (11.2) - may be written as

$$x_{n+1}^2 + y_{n+1}^2 \leq 4$$

Pixels are passed from the *Switch* process each time the *Mandelbrot* process completes the previous calculation and requires more work. Completed work, in terms of the pixel colour, is

passed on to the *Feedback* process. In pseudo-code, this gives

```
WHILE pixels are available
  SEQ
    RECEIVE pixel
    loop:
      CALCULATE next iteration of recurrence relation
      IF iteration count = maximum THEN
        ASSIGN black to colour
        EXIT
      IF modulus squared > constant THEN
        ASSIGN count to colour
        EXIT
    SEND pixel and colour to feedback
```

Again the actual code will be complicated by the termination condition. This time the termination notice is passed on to the *Feedback* process. Expressed in occam (assuming a graphics palette of 256 colours, with the colour black having a value of 0, and a maximum number of iterations of 255), this gives

```
PROC Mandelbrot (CHAN OF BYTE MoreWork,
                 CHAN OF RAW Work,
                 CHAN OF PROCESSED WorkDone)
  BOOL Running :
  SEQ
    Running := TRUE
    WHILE Running
      BYTE Any :
      [2] INT Coords :
      SEQ
        -- ask for some work
        MoreWork ! Any
        Work ? CASE
          Data ; Coords      -- next set of coordinates
            VAL Constant IS 4.0 (REAL32) :
            VAL Two IS 2.0 (REAL32) :
            VAL MaxIterations IS 255 :
            VAL Black IS 0 :
            REAL32 A, B, X, Y, ZSquared :
            INT Colour, Count :
```

```
SEQ
  A := Coords [0]
  B := Coords [1]
  X := 0.0 (REAL32)
  Y := 0.0 (REAL32)
  Count := 0
  ZSquared := 0.0 (REAL32)
  -- calculate next iteration of recurrence relation
  -- and test for divergence
  WHILE (Count < MaxIterations) AND (ZSquared <= Constant)
    SEQ
      X := ((X * X) - (Y * Y)) + A
      Y := (Two * (X * Y)) + B
      ZSquared := (X * X) + (Y * Y)
      Count := Count + 1
  IF
    Count = MaxIterations -- pixel in Mandelbrot set
      Colour := Black
    ZSquared > Constant   -- pixel outside Mandelbrot set
      Colour := Count
  WorkDone ! Results ; Coords ; Colour      -- send results back
Terminate     -- termination notice
  Running := FALSE
WorkDone ! Quit       -- pass on the termination nouce
```

The *Feedback* process multiplexes the results from its *Mandelbrot* process and those received from other workers on the farm, and feeds them back to the *Farmer* process. Putting this in pseudo-code

```
WHILE pixels are available
  ALT
    RECEIVE pixel and colour from our worker
      SEND pixel and colour back to graphics process
    RECEIVE pixel and colour from other workers
      SEND pixel and colour back to graphics process
```

The code may be succinctly expressed in occam using an **ALT** construction. This time the treatment of the terminating condition needs more effort. The termination notice is only passed on when one has been received from *both* the local *Mandelbrot* process and the next worker [Jones and Goldsmith].


```
PROC Feedback (CHAN OF PROCESSED ToPrevious, FromNext, WorkDone)
  [2] INT Coords :
  INT Colour :
  BOOL Local, Other :
  SEQ
    Local := TRUE
    Other := TRUE
    WHILE Local OR Other
      ALT
        Local & WorkDone ? CASE
          Results ; Coords ; Colour -- results from our worker
            ToPrevious ! Results ; Coords ; Colour    -- pass back to farmer
          Quit    -- termination notice from our worker
            Local := FALSE
        Other & FromNext ? CASE
          Results ; Coords ; Colour        -- results from another worker
            ToPrevious ! Results ; Coords ; Colour    -- pass back to farmer
          Quit    -- termination notice from next worker
            Other := FALSE
    ToPrevious ! Quit              -- pass on termination notice
```

In the above process boolean variables, *Local* and *Other*, record whether or not a termination notice has been received from the local worker or another worker respectively. Only when a termination notice has been received from both these processes does the *Feedback* process terminate.

A simplistic farmer process, which assumes that the farm is arranged as a chain of worker processes, is presented below. The farmer sends each pixel to the first worker in the chain for redistribution. The farm is primed by issuing 2**NumberOfWorkers* pixels to the work force. This amount of data just fills up each worker and each buffer. As each pixel is reported processed, another pixel is issued to the farm, until all the pixels have been processed [Atkin]. At this point a termination notice is issued to the farm of worker processes and the farmer waits to receive this back before finally terminating itself. In pseudo-code this is

```
PRIME farm with pixels
loop:
  RECEIVE completed work from farm
  SEND another pixel to fram
UNTIL pixels exhausted
```

The termination notice will pass down the chain of worker processes via the switch processes causing each switch process to terminate on receipt. Before terminating, each switch process will inform its mandelbrot process to terminate, which in turn will inform the feedback process of a local termination. When the termination notice reaches the end of the chain of worker processes, it must be returned to the farmer process via the feedback processes, causing each feedback process to terminate (provided the local termination has also been received). Writing this in occam gives

```
PROC Farmer (CHAN OF RAW ToWorker,
             CHAN OF PROCESSED FromWorker)
  INT WorkDone, WorkWanted :
  BOOL Running, Terminating :
  SEQ
    -- prime the farm
    SEQ Index = 0 FOR 2 * NumberOfWorkers
      -- send pixel (row and column)
      ToWorker ! Data ; [Index / 512, Index REM 512]
    WorkDone := 2 * NumberOfWorkers
    WorkWanted := 0
    Running := TRUE
    Terminating := FALSE
    WHILE Running
      INT Colour :
      [2] INT ResultCoords :
      PRI ALT
        FromWorker ? CASE
          Results ; ResultCoords ; Colour        -- receive completed work
            SEQ
              .
              . -- plot result
              .
              WorkWanted := WorkWanted + 1
          Quit    -- termination notice returned
            Running := FALSE
```



```
        (WorkWanted > 0) & SKIP
          IF
            WorkDone < NumberOfPixels
              SEQ
                -- send another pixel
                ToWorker ! Data ; [Index / 512, Index REM 512]
                WorkDone := WorkDone + 1
                WorkWanted := WorkWanted - 1
            NOT Terminating
              SEQ
                Terminating := TRUE
                ToWorker ! Terminate -- issue termination notice to farm
            TRUE
              SKIP
:
```

In the above process, the variable *WorkDone* keeps a count of the number of pixels processed, while the variable *WorkWanted* keeps a count of the number of pixels required to top up the farm.

The main process will comprise a PAR construction containing instances of the *Farmer* process and a number of replicated *Worker* processes. The last worker in the chain is a special case as it has no one else further down the chain to communicate with. A dummy process, *EndStop*, is provided to match the channels of this last worker so they are not left dangling. This dummy process accepts the termination notice from the switch process of the last worker (protocol *RAW*) and and generates a new one to pass back down the chain of feedback processes to the farmer (protocol *PROCESSED*) [Jones and Goldsmith].

```
PAR
  Farmer (ToFarm [0], FromFarm [0])
  PAR Index = 0 FOR NumberOfWorkers
    Worker (ToFarm [Index], ToFarm [Index + 1],
            FromFarm [Index], FromFarm [Index + 1])
  EndStop (ToFarm [NumberOfWorkers], FromFarm [NumberOfWorkers])
```

An alternative to the dummy process approach is to treat the last worker separately [Packer]. This approach requires that the protocols *RAW* and *PROCESSED* be combined into one protocol, *COMBINED*, say.

```
PROTOCOL COMBINED
  CASE
    Data ; [2] INT
    Results ; [2] INT ; INT
    Terminate
:
```

This means, of course, that the process and channel declarations, and the tag input and outputs must be altered accordingly. For example, the *Worker* process becomes

```
PROC Worker (CHAN OF COMBINED FromPrevious, ToNext,
                 ToPrevious, FromNext)
  CHAN OF BYTE MoreWork :
  CHAN OF COMBINED Work, WorkDone :
  PRI PAR
    PAR
      Switch (MoreWork, FromPrevious, ToNext, Work)
      Feedback (ToPrevious, FromNext, WorkDone)
    Mandelbrot (MoreWork, Work, WorkDone)
:
```

and the *Feedback* process becomes

```
PROC Feedback (CHAN OF COMBINED ToPrevious, FromNext, WorkDone)
  [2] INT Coords :
  INT Colour :
  BOOL Local, Other :
  SEQ
    Local := TRUE
    Other := TRUE
    WHILE Local OR Other
      ALT
        Local & WorkDone ? CASE
          Results ; Coords ; Colour -- results from our worker
            ToPrevious ! Results ; Coords ; Colour    -- pass back to farmer
          Terminate    -- termination notice from our worker
            Local := FALSE
        Other & FromNext ? CASE
          Results ; Coords ; Colour       -- results from another worker
            ToPrevious ! Results ; Coords ; Colour    -- pass back to farmer
          Terminate    -- termination notice from next worker
            Other := FALSE
        ToPrevious ! Terminate       -- pass on termination notice
:
```

A special channel, *LoopBack* is declared as follows:

```
CHAN OF COMBINED LoopBack :
```

This channel is looped back in the last worker process to provide a return path for the termination notice.

```
PAR
  Farmer (ToFarm [0], FromFarm [0])
  PAR Index = 0 FOR NumberOfWorkers - 1
    Worker (ToFarm [Index], ToFarm [Index + 1],
            FromFarm [Index], FromFarm [Index + 1])
  Worker (ToFarm [NumberOfWorkers], LoopBack
          FromFarm [NumberOfWorkers], LoopBack)
```

Another example of process farming is its application to ray tracing to generate realistic images of scenes [Packer]. Such an application requires considerable amounts of processing power. It has been shown that the processing speed is directly proportional to the number of transputers used for this generation of images.

11.4 Efficiency factors

Even after designing a parallel algorithm, there are a number of competing factors to be taken into consideration for an efficient implementation.

- processor connectivity - the transputer has only *four* physical links. Depending on the distribution of processes on processors, communications between processes may need to pass through several intervening transputers. This routing of messages imposes an extra overhead on each transputer, and the balance between computation and communication needs to be carefully assessed.

- processor loading - the processing load of each transputer in the network must

be taken into consideration. The system is likely to run at the speed of the transputer with the highest processing load, as the other transputers in the system will probably be held up, waiting to communicate with this transputer. So the processing load should be shared as evenly as possible among the available transputers, and not left to chance or haphazard placement. The farm approach semi-dynamically balances the load for each processor, since a processor only receives more work when it becomes idle. The algorithmic approach needs especial care, as an overloaded processor may create a bottleneck in the pipeline or whatever configuration is chosen.

• processor type and memory - there are different types of transputer available with different word sizes and floating point capabilities. The choice of transputer for a particular function within a network needs to be carefully considered to match applications with suitable processors. For example, computation-bound tasks involving floating-point operations will obviously benefit from the use of a T800 processor. As regards memory, the program memory requirement must be balanced against the use of internal memory (fast but finite) and external memory.

3

Métricas de Desempeño

Measuring Parallel Processor Performance

Many metrics are used for measuring the performance of a parallel algorithm running on a parallel processor. This article introduces a new metric that has some advantages over the others. Its use is illustrated with data from the Linpack benchmark report and the winners of the Gordon Bell Award.

Alan H. Karp and Horace P. Flatt

There are many ways to measure the performance of a parallel algorithm running on a parallel processor. The most commonly used measurements are the elapsed time, price/performance, the speed-up, and the efficiency. This article defines another metric which reveals aspects of the performance that are not easily discerned from the other metrics.

The elapsed time to run a particular job on a given machine is the most important metric. A Cray Y-MP/1 solves the order 1,000 linear system in 2.17 seconds compared to 445 seconds for a Sequent Balance 21000 with 30 processors [2]. If you can afford the Cray and you spend most of your time factoring large matrices, then you should buy a Cray.

Price/performance of a parallel system is simply the elapsed time for a program divided by the cost of the machine that ran the job. It is important if there are a group of machines that are "fast enough." Given a fixed amount of money, it may be to your advantage to buy a number of slow machines rather than one fast machine. This is particularly true if you have many jobs to run and a limited budget. In the previous example, the Sequent Balance is a superior price/performer than the Cray if it costs less than 0.5 percent as much. On the other hand, if you can't wait 7 minutes for the answer, the Sequent is not a good buy even if it wins in price/performance.

These two measurements are used to help you decide what machine to buy. Once you have bought the machine, speed-up and efficiency are the measurements often used to let you know how effectively you are using it.

The speed-up is generally measured by running the same program on a varying number of processors. The speed-up is then the elapsed time needed by 1 processor divided by the time needed on p processors, $s = T(1)/T(p)$. (Of course, the correct time for the uniprocessor run would be the time for the best serial algorithm, but almost nobody bothers to write two programs.) If you are interested in studying algorithms for parallel processors, and system A gives a higher speed-up than system B for the same program, then you would say that system A provides better support for parallelizing this program than does system B.

An example of such support is the presence of more processors. A Sequent with 30 processors will almost certainly produce a higher speed-up than a Cray with only 8 processors.

The issue of efficiency is related to that of price/performance. It is usually defined as

$$e = \frac{T(1)}{pT(p)} = \frac{s}{p}. \qquad (1)$$

Efficiency close to unity means that you are using your hardware effectively; low efficiency means that you are wasting resources. As a practical matter, you may buy a system with 100 processors that each takes 100 times longer than you are willing to wait to solve your problem. If you can code your problem to run at high efficiency, you'll be happy. Of course, if you have 200 processors, you may not be unhappy with 50 percent efficiency, particularly if the 200 processors cost less than other machines that you can use.

Each of these metrics has disadvantages. In fact, there is important information that cannot be obtained even by looking at all of them. It is obvious that adding processors should reduce the elapsed time, but by how much? That is where speed-up comes in. Speed-up close to linear is good news, but how close to linear is good enough? Well, efficiency will tell you how close you are getting to the best your hardware can do, but if your efficiency is not particularly high, why? The new metric defined in the following section is intended to answer these questions.

NEW METRIC

We will now derive our new metric, the experimentally determined serial fraction, and show why it is useful. We will start with Amdahl's Law [1] which in its simplest form says that

$$T(p) = T_s + \frac{T_p}{p}. \quad (2)$$

where T_s is the time taken by the part of the program that must be run serially and T_p is the time in the parallelizable part. Obviously, $T(1) = T_s + T_p$. If we define the fraction serial, $f = T_s/T(1)$ then equation (2) can be written as

$$T(p) = T(1)f + \frac{T(1)(1-f)}{p}, \quad (3)$$

or, in terms of the speed-up s

$$\frac{1}{s} = f + \frac{1-f}{p}. \quad (4)$$

We can now solve for the serial fraction, namely

$$f = \frac{1/s - 1/p}{1 - 1/p}. \quad (5)$$

The experimentally determined serial fraction is our new metric. While this quantity is mentioned in a large percentage of papers on parallel algorithms, it is virtually never used as a diagnostic tool the way speed-up and efficiency are. It is our purpose to correct this situation.

The value of f is useful because equation (2) is incomplete. First, it assumes that all processors compute for the same amount of time, i.e., the work is perfectly load balanced. If some processors take longer than others, the measured speed-up will be reduced giving a larger measured serial fraction. Second, there is a term missing that represents the overhead of synchronizing processors.

Load-balancing effects are likely to result in an irregular change in f as p increases. For example, if you have 12 pieces of work to do that take the same amount of time, you will have perfect load balancing for 2, 3, 4, 6, and 12 processors, but less than perfect load balancing for other values of p. Since a larger load imbalance results in a larger increase in f, you can identify problems not apparent from speed-up or efficiency.

The overhead of synchronizing processors is a monotonically increasing function of p, typically assumed to increase either linearly in p or as $\log p$. Since increasing overhead decreases the speed-up, this effect results in a smooth increase in the serial fraction f as p increases. Smoothly increasing f is a warning that the granularity of the parallel tasks is too fine.

A third effect is the potential reduction of vector lengths for certain parallelizations of a particular algorithm. Vector processor performance normally increases as vector length increases except for vector lengths slightly larger than the length of the vector registers. If the parallelization breaks up long vectors into shorter vectors, the time to execute the job can increase. This effect then also leads to a smooth increase in the measured serial fraction as the number of processors increases. However, large jobs usually have very long vectors, vector processors usually have only a few processors (the Intel iPSC-VX is an exception), and there are usually parallelizations that keep the vector lengths fixed. Thus, the reduction in vector length is rarely a problem and can often be avoided entirely.

In order to see the advantage of the serial fraction over the other metrics, look at Table I which is extracted from Table 2 of the Linpack report [2]. The Cray Y-MP shows speed-ups ranging from 1.95 to 6.96. Is that good? Even if you look at the efficiency, which ranges from 0.975 to 0.870, you still don't know. Why does the efficiency fall so rapidly? Is there a lot of overhead when using 8 processors? The serial fraction, f, answers the question; the serial fraction is nearly constant for all values of p. The loss of efficiency is due to the limited parallelism of the program.

The single data point for the Sequent Balance reveals that f performs better as a metric as the number of processors grows. The efficiency of the 30 processor Sequent is only 83 percent. Is that good? Yes, it is since the serial fraction is only 0.007.

The data for the Alliant FX/40 shows something different. Here the speed-up ranges from 1.90 to 3.22 and the efficiency from 0.950 to 0.805. Although neither of these measurements tells much, the fact that f ranges from 0.053 to 0.080 does; there is some overhead that is increasing as the number of processors grows. We can't tell what this overhead is due to—synchronization cost, memory contention, or what—but at least we know it is there. The effect on the FX/80 is much smaller although there is a slight increase in f, especially for fewer than 5 processors.

Even relatively subtle effects can be seen. The IBM 3090 has a serial fraction under 0.007 for 2 and 3 processors, but over 0.011 for 4 or more. Here the reason is most likely due to the machine configuration; each set of 3 processors is in a single frame and shares a memory management unit. Overhead increases slightly when two of these units must coordinate their activities. This effect also shows up on the 3090-280S which has two processors in two frames. Its run has twice the serial fraction as does the run on the 3090-200S. None of the other metrics would have revealed this effect.

Another subtle effect shows up on the Convex. The 4 processor C-240 shows a smaller serial fraction than does the 2 processor C-220. Since the same code was presumably run on both machines, the actual serial fraction must be the same. How can the measured value decrease? This appears to be similar to the "superlinear" speed-ups reported on some machines. As in those cases, adding processors adds cache and memory bandwidth which reduces overhead. Perhaps that is the case here.

Care must be used when comparing different machines. For example, the serial fraction on the Cray is 3 times larger than on the Sequent. Is the Cray really that inefficient? The answer is no. Since almost all the parallel work can be vectorized, the Cray spends relatively less time in the parallel part of the code than does the Sequent which has no vector unit. Since the parallelizable part speeds up more than the serial part which

Table I. Summary of Linpack report Table 2

Computer	p	Time(sec)	s	e	f
Cray Y-MP/8	1	2 17	—	—	—
Cray Y-MP/8	2	1 11	1.95	0.975	0.024
Cray Y-MP/8	3	0 754	2.88	0.960	0.021
Cray Y-MP/8	4	0.577	3.76	0.940	0 021
Cray Y-MP/8	8	0.312	6.96	0 870	0.021
IBM 3090-180S VF	1	7.27	—	—	—
IBM 3090-200S VF	2	3.64	2.00	1.000	0.002
IBM 3090-280S VF	2	3 65	1.99	0 995	0.004
IBM 3090-300S VF	3	2.46	2.96	0.987	0.007
IBM 3090-400S VF	4	1.89	3 85	0 963	0 013
IBM 3090-500S VF	5	1.52	4.78	0 956	0.011
IBM 3090-600S VF	6	1.29	5.64	0.940	0.012
Alliant FX/40	1	66.1	—	—	—
Alliant FX/40	2	34 8	1.90	0.950	0.053
Alliant FX/40	3	24.9	2 65	0.883	0.066
Alliant FX/40	4	20.5	3.22	0 805	0.080
Alliant FX/80	1	57.7	—	—	—
Alliant FX/80	2	29 8	1.94	0.970	0.032
Alliant FX/80	3	20.7	2.79	0 930	0.038
Alliant FX/80	4	16.2	3.56	0.890	0.041
Alliant FX/80	5	13.6	4.24	0 848	0.045
Alliant FX/80	6	11.8	4.89	0.815	0.046
Alliant FX/80	7	10 6	5.44	0.777	0.048
Alliant FX/80	8	9.64	5.99	0.749	0.048
Sequent	1	11111	—	—	—
Sequent	30	445	25.0	0.833	0.007
Convex C-210	1	15	—	—	—
Convex C-220	2	7 98	1.88	0.940	0.064
Convex C-240	4	4.03	3.72	0.930	0 025

Note: p=#processors, s=speed-up, e=efficiency, f=serial fraction

has less vector content, the fraction of the time spent in serial code is increased.

The Linpack benchmark report measures the performance of a computational kernel running on machines with no more than 30 processors. The results in Table II are taken from the work of the winners of the Gordon Bell Award [5]. Three applications are shown with maximum speed-ups of 639, 519, and 351 and efficiencies ranging from 0.9965 to 0.3430. We know this is good work since they won the award, but how good a job did they do? The serial fraction ranges from 0.00051 to 0.0019 indicating that they did a very good job, indeed.

The serial fraction reveals an interesting point. On all three problems, there is a significant reduction in the serial fraction in going from 4 to 16 processors (from 16 to 64 for the wave motion problem). As with the Convex results, these numbers indicate something akin to "superlinear" speed-up. Perhaps the 4-processor run sends longer messages than does the 16-processor run, and these longer messages are too long for the system to handle efficiently. At any rate, the serial fraction has pointed up an inconsistency that needs further study.

SCALED SPEED-UP

All the analysis presented so far refers to problems of fixed size. Gustafson [4] has argued that this is not how parallel processors are used. He argues that users will increase their problem size to keep the elapsed time of the run more or less constant. As the problem size grows, we should find the fraction of the time spent executing serial code decreases, leading us to predict a decrease in the measured serial fraction.

If we assume that the serial time and overhead are independent of problem size, neither of which is fully justified, [3]

$$T(p, k) = T_s + \frac{kT_p}{p}, \qquad (6)$$

where $T(p, k)$ is the time to run the program on p processors for a problem needing k times more arithmetic. Here k is the scaling factor and $k = 1$ when $p = 1$. Flatt [3] points out that the scaling factor k must count arithmetic, not some more convenient measure such as memory size.

Our definition of speed-up must now account for the additional arithmetic that must be done to solve our

Table II. Summary of Bell Award winning performance

p	s	e	f
	Wave Motion		
4	3.986	0 9965	0.0012
16	15.86	0.9913	0 00097
64	62.01	0 9689	0 00051
256	226.2	0 8836	0 00052
1024	639.0	0 6240	0.00059
	Fluid Dynamics		
4	3.959	0 9898	0.0035
16	15.47	0 9669	0.0023
64	58.53	0 9145	0.0015
256	201.6	0 7875	0 0011
1024	519.1	0 5069	0.00095
	Beam Stress		
4	3 954	0 9885	0.0039
16	15 46	0 9663	0.0023
64	57 46	0 8978	0 0018
256	177.5	0 6934	0 0017
1024	351.2	0 3430	0.0019

Note: p=#processors, s=speed-up, e=efficiency, f=serial fraction

larger problem. We can use

$$s_k = \frac{kT(1, 1)}{T(p, k)}. \quad (7)$$

The scaled efficiency is then $e_k = s_k/p$, and the scaled serial fraction becomes

$$f_k = \frac{1/s_k - 1/p}{1 - 1/p}. \quad (8)$$

By our previous definitions we see that $f = kf_k$ which under ideal circumstances would remain constant as p increases.

The scaled results are shown in Table III. Although these runs take constant time as the problem size grows, the larger problems were run with shorter time steps. A better scaling would be one in which the time integration is continued to a specific value.

If the run time were still held constant, the problems would scale more slowly than linearly in p. In thes examples, the Courant condition limits the step size which means that the correct scaling would be $k = \sqrt{p}$. The scaling chosen for the problems of Table III is $k = p$.

As predicted [3], the efficiency decreases, barely, as p increases even for scaled problems. The scaled serial fraction, on the other hand, decreases smoothly. This fact tells us that the decreasing efficiency is not caused by a growing serial fraction. Instead it tells us the problem size is not growing fast enough to completely counter the loss of efficiency as more processors are added.

The variation in f_k as the problem size grows makes it difficult to interpret. If we allow for the fact that ideally the increase in the problem size does not affect the serial work, we again have a metric that should remain constant as the problem size grows. Table III shows how kf_k varies with problem size. We see that this quantity grows slowly for the wave motion problem, but that there is virtually no increase for the fluid dynamics problem. These results indicate that the serial work increases for the wave motion problem as the problem size grows but not for the fluid dynamics problem. The irregular behavior of kf_k for the beam stress problem warrants further study.

SUMMARY

We have shown that the measured serial fraction, f, provides information not revealed by other commonly used metrics. The metric, properly defined, may also be useful if the problem size is allowed to increase with the number of processors.

What makes the experimentally determined serial fraction such a good diagnostic tool of potential per-

Table III. Bell Award scaled problems. $k=p$

p	s_k	e_k	f_k	kf_k
		Wave Motion		
4	3.998	0 9995	0.00013	0.00053
16	15.95	0 9969	0.00020	0.0032
64	63 61	0.9939	0 000097	0.0062
256	254.1	0.9926	0.000029	0.0074
1024	1014	0.9902	0.0000098	0.010
		Fluid Dynamics		
4	3.992	0.9980	0.00067	0.0027
16	15.96	0 9975	0.00015	0.0024
64	63.82	0 9972	0.000046	0.0029
256	255.2	0.9969	0.000013	0.0033
1024	1020	0.9961	0.0000033	0.0034
		Beam Stress		
4	4 001	1 000	0.0	0.0
16	16 00	1.000	0.000021	0.00034
64	63.96	0 9994	0.000015	0.00098
256	255.8	0 9992	0.0000038	0.00096
1024	1023	0 9990	0.0000012	0.001

Note: p=#processors, s_k=scaled speed-up, e_k=scaled efficiency, f_k=scaled serial fraction

formance problems? While elapsed time, speed-up, and efficiency vary as the number of processors increases, the serial fraction would remain constant in an ideal system. Small variations from perfect behavior are much easier to detect from something that should be constant than from something that varies. Since kf_k for scaled problems shares this property, it, too, is a useful tool.

Ignoring the fact that p takes on only integer values makes it easy to show that

$$\frac{d}{dp}\frac{1}{e} = f \tag{9}$$

if we ignore the overhead and assume that the serial fraction is independent of p. Thus, we see that the serial fraction is a measure of the rate of change of the efficiency. Even in the ideal case, $1/e$ increases linearly as the number of processors increases. Any deviation of $1/e$ from linearity is a sign of lost parallelism. The fraction serial is a particularly convenient measure of this deviation from linearity.

The case of problem sizes that grow as the number of processors is increased is only slightly more complicated. In this case

$$\frac{d}{dp}\frac{1}{e_k} = f_k + (p-1)\frac{df_k}{dp} = \frac{f}{k}\left(1 - \frac{p-1}{k}\frac{dk}{dp}\right). \tag{10}$$

If $k = 1$, *i.e.*, there is no scaling, equation (10) reduces to equation (9). If $k = p$, then $1/e$ has a slope of f/p^2 which is a very slow loss of efficiency. Other scalings lie between these two curves.

It is easy to read too much into the numerical values. When the efficiency is near unity, $1/s$ is close to $1/p$ which leads to a loss of precision when subtracting in the numerator of equation (5). If care is not taken, variations may appear that are mere round-off noise. All entries in the tables were computed from

$$f = 1 - \frac{1 - 1/s}{1 - 1/p}. \tag{11}$$

Since both s and p are considerably greater than unity and neither $1/s$ nor $1/p$ is near the precision of the floating point arithmetic, there is only one place where precision can be lost. Rounding the result to the significance of the reported times guarantees that the results are accurate. Similarly,

$$f_k = 1 - \frac{1 - 1/s_k}{1 - 1/p} \tag{12}$$

is used for the scaled serial fraction.

Although our numerical examples come from rather simple cases, one of us (Alan Karp) has successfully used this metric to find a performance bug in one of his applications. Noting an irregularity in the behavior of f led him to examine the way the IBM Parallel Fortran prototype compiler was splitting up the work in the loops. Due to an oversight, the compiler truncated the result of dividing the number of loop iterations by the number of processors. This error meant that one processor had to do one extra pass to finish the work in the loop. For some values of p this remainder was small; for others, it was large. The solution was to increase his problem size from 350, which gives perfect load balancing on his six-processor IBM 3090-600S only for 2 and 5 processors, to 360 which always balances perfectly. (This error was reported to the IBM Parallel Fortran compiler group.)

REFERENCES

1. Amdahl, G.M. Validity of the single processor approach to achieving large scale computer capabilities. In *Proceedings of AFIPS Spring Joint Computer Conference*, 30. Atlantic City, NJ, 1967.
2. Dongarra, J.J. Performance of various computers using standard linear equation software. Report CS-89-85, Computer Science Department, Univ. Tennessee, Knoxville, October 12, 1989.
3. Flatt, H.P., and Kennedy, K. Performance of parallel processors. *Parallel Comput.* 12, (Oct. 1989), 1–20
4. Gustafson, J.L. Reevaluating Amdahl's Law. *Commun. ACM* 31, 5 (May 1988), 532–533
5. Gustafson, J.L., and Montry, G.R., and Brenner, R.E. Development of parallel methods for a 1024-processor hypercube. *SIAM Sci. Stat. Comp.* 9, (July 1988), 609–638.

CR Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)—*parallel processors*; C.4 [Computer Systems Organization]: Performance of Systems—*measurement techniques*

General Terms: Measurement, Performance

Additional Key Words and Phrases: Parallel performance

ABOUT THE AUTHORS:

ALAN KARP is a staff member at IBM's Palo Alto Scientific Center. He has worked on problems of radiative transfer in moving stellar matter and in planetary atmospheres, hydrodynamics problems in pulsating stars and in enhanced oil recovery, and numerical methods for parallel processors. He is currently studying the interface between programmers and parallel processors with special attention to debugging parallel algorithms.

HORACE P. FLATT is manager of IBM's Palo Alto Scientific Center. He received a Ph.D. in mathematics from Rice University in 1958, subsequently becoming manager of the applied mathematics group of Atomics International, Inc. He joined IBM in 1961 and has primarily worked in management assignments in applied research in computer systems and applications. Authors' Present Address: IBM Scientific Center, 1530 Page Mill Road, Palo Alto, CA 94304.

4

Transputer

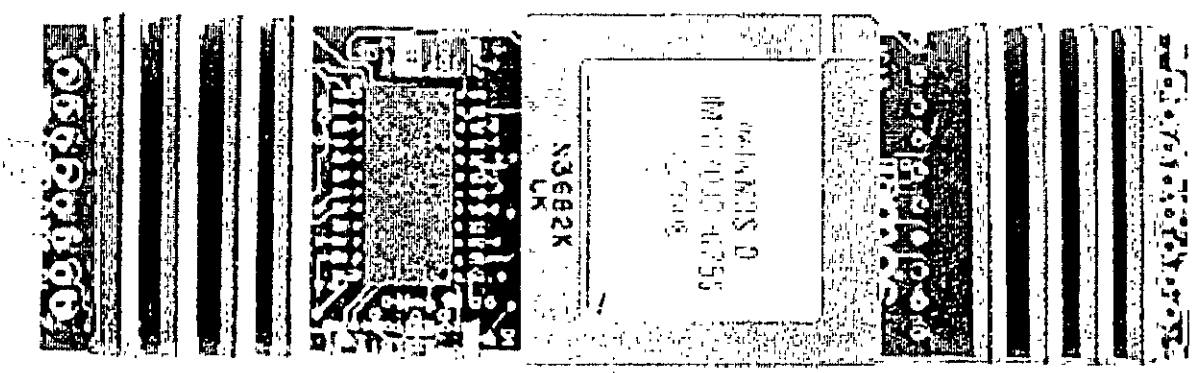
Transputers

design and use as a building block

Jacco de Leeuw Arjan de Mes

FWI University of Amsterdam

October 1992



This TRAM contains a IMST800 transputer and 4Mb of DRAM. It is sold by Inmos as an 'off-the-shelf' component for use as a building block.

1. Introduction

The first idea for the transputer stems from 1975, when Ian Barron had the idea that it should be possible to build a processor on a single chip which could be used as a building block for larger parallel computers, or even supercomputers. Along with this processor, a high level language had to be provided to use all of its possibilities.

In 1984 the technology had advanced far enough for the first transputer to be built. It became possible to place a processor with memory (not much) and communication facilities on a single chip. The first transputer was built by the firm Inmos and was for testing purposes only. In 1985 the first commercial transputer was made by the same firm. This was the T414a.

The T414a became successful when Inmos produced an adapter for the IBM's Personal Computer and released the language OCCAM 1, to program the transputer. Unfortunately, there were a number of flaws to the T414a and OCCAM 1. The T414a did not have a floating point unit and OCCAM 1 provided no alternative. OCCAM 1 did not support functions, nor the possibility to send more than one variable through a communication channel.

The T414b and OCCAM 2 solved most of those problems. The transputer was ready for serious applications, such as real-time systems and as a building block for supercomputers. During that same period Inmos built a smaller and cheaper variation of the T414, the T212.

The T800, which was introduced in 1987, was the first transputer with a coprocessor. This coprocessor could be placed on-chip due to new advances in the VLSI technology. The latest news [15NETN92] suggests Inmos is planning to release a new transputer with an on-chip router, the T9000. It is said to be 10 times faster than the T800. It is however feared by experts, that the speed of the T9000 will be outperformed by its (non-transputer) competitors before its official release.

We will not discuss OCCAM in our paper, this has been done many times before and we do not consider it essential to our statement.

As VLSI technology progresses, the requirements for computer systems keep increasing. These requirements increase so fast, the VLSI technology can't keep the pace. To meet these new demands, the Von Neumann architecture has become insufficient. Therefore research in this field concentrates on other architectures.
Decisions have to be made concerning the grain size (i.e. what is the smallest unit that can be executed in parallel). The transputer uses 'fine grain parallelism' internally (distribution of instructions), but when communicating with other transputers, it uses 'medium grain parallelism' (distribution of procedures). It is hard to classify the transputer into one particular class as defined by Flynn [01FLYN66] (i.e. SIMD, MIMD etc.). This depends on the environment in which the transputer is used. It can be used stand-alone (SISD) or in a wavefront approach (SPrMD: Single Program Multiple Data), etc. Although the transputer has a reduced instruction set, it can not be considered as a RISC processor; it has a microprogram.

2. Transputer Internal Architecture

Every transputer has the same basic architecture. The internal bus connects the processor to local memory and to an external memory interface. The communication links are connected to the bus via an interface. This makes it possible for the processor to work independent of the links. Depending on the type of transputer, the floating point unit is also connected to this bus.

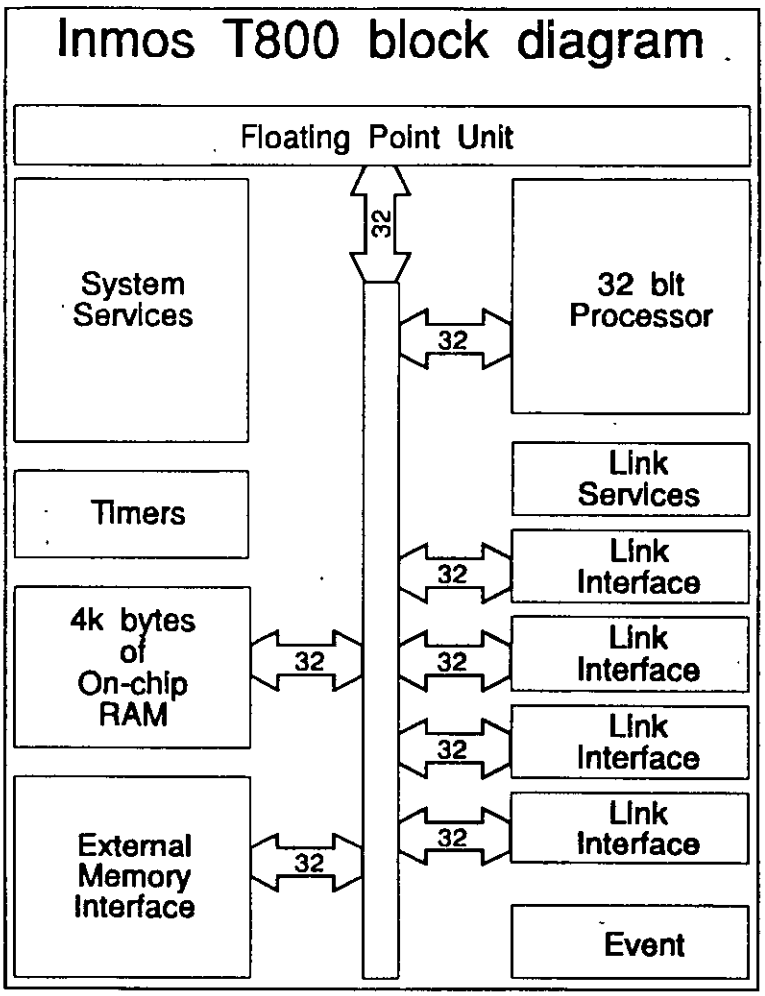


figure 1 : Layout T800

The system services take care of functions as reset, booting from ROM and hardware interfacing with a host computer (although the actual communication is done through the links). It also helps in analyzing run-time errors. The transputer has four communication links, this makes it possible for the transputer to communicate with other transputers and computers. Inmos supplies several interface chips and add-on boards for other protocols, such as RS-232, SCSI and Micro-computer busses.

The On-chip RAM, which is also connected directly to the bus, provides 2 to 5 times higher access speeds, than the External Memory Interface. For the 30 MHz version of the T800, the speed of the

External Memory Interface is 40 Mb per second, while the internal bus uses speeds up to 120 Mb per second. As all memory access is word-size only and the transputers which we are discussing are all 32 bits, the respective access times for external and internal memory are 100μs and 33 μs.

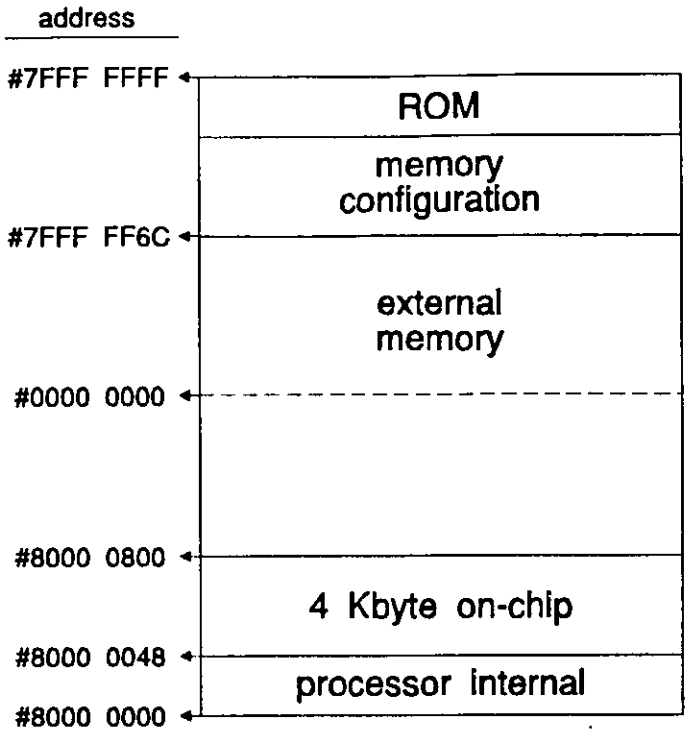


figure 2 : T800 memory map

[3]

3. Instruction Set

The transputer has 6 registers, each 32 bits wide. Three of these registers form a small but fast stack, registers A, B and C. The other three registers have explicit purposes. The O-register (Operand Register) contains the operand to the current instruction. The I-register (Instruction Register) contains a pointer to the next instruction. The W-register (Workspace Register) is a pointer to the program's workspace; all memory addresses are relative to the W-register.

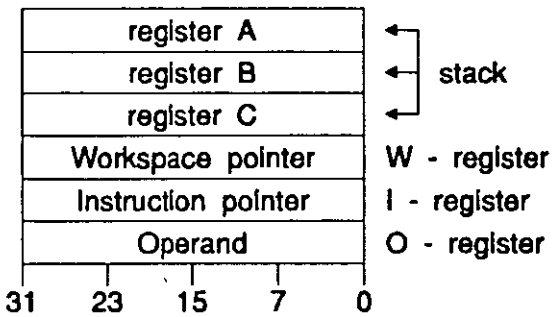


figure 3 : The 6 registers

The transputer uses a fairly conventional RISC-like instruction set, to which special instructions are added to facilitate features such as process scheduling and inter-process communication. This does not mean we can call the transputer a Reduced Instruction Set Computer (RISC). Using [16COII9x] we can compare the transputer with RISC design issues.

To start with, the transputer is microcoded. That means, every time the transputer executes an instruction, a small sequential program ("microcode") is interpreted which performs this instruction's function. RISC processors use hardwired control, which means that instructions are "executed" directly using hardware circuitry. This reduces the amount of needed processor cycles per instruction, as no microcoded sequential program is run. Hardwired control uses more circuitry on-chip in comparison with micro-programming. This could be a reason why no hardwired control is used in the transputer; the designers had to make the most out of the available chip space. Of course, the result of the microcoded scheme is that more time is spent on the decoding of the instructions.

An important advantage of a microcoded processor is that upward compatibility can be realized in the microprogram. This can be done "ad absurdum" such as in the Intel 80x86 processors, but microprograms take increasingly more chip-space and become much more complex as functions are added. The designers of the transputer used another scheme to accomplish upward compatibility, as we will see later on.

For each instruction in the transputer's instruction set 1 byte is reserved, of which the 4 most significant bits represent the function code and the 4 least significant bits represent a data value.

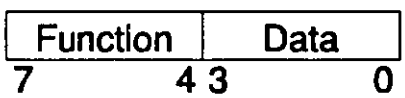


figure 4 : Instruction format

This fixed instruction format speeds up decoding. This is a typical RISC trait. The reader could argue that this results in only 16 possible instructions with each 16 possible values as an operand. Strictly speaking this is true, but of the 16 possible instructions, there are two which are used to extend the operand, PFIX and NFIX. This is where the operand register (O-register) comes in.

When a one-byte instruction is executed, its 4-bit operand is placed in the least significant 4 bits of this operand register. But before that, the existing contents in the operand registers are shifted 4 bits to the left, freeing a 4 bit space. Normally, instructions clear the operand register after they have executed. The mentioned PFIX and NFIX instructions do not clear this register. In fact, PFIX (PreFIX) does nothing but shifting its operand nibble into the operand register. Thus, by using a series of PFIX instructions, operands can range from 4 bits up to the length of the operand register (in steps of 4 bits). NFIX, (Negative PreFIX), works the same as PFIX but also complements the contents of the operand register after shifting it. This allows negative operands to be built up more quickly. Of course, just PFIX would be sufficient, but (small) negative operands tend to occur as frequently as (small) positive operands, so an extra instruction NFIX has been added to the instruction set.

For illustration purposes: the assembly language instruction `ldc #1234` (which loads the constant `0x1234` into the register stack) is split into the following machine language instructions.

`pfix`	`#1`	The previous instruction left the operand register blanked, so shift the register 4 bits left and #1 into the register. Don't clear the register.
`pfix`	`#2`	Shift the operand register four bits to the left and #2 into the register.
`pfix`	`#3`	ditto Now the value in the operand register has become 00000123.
`ldc`	`#4`	Shift the operand register four bits to the left and #4 into the register, then push the operand onto the A register.

This coded into memory as #21 #22 #23 #44, where instruction PFIX is 2 and LDC is 4.

This scheme of building up operands has several beneficial points. For one thing, PFIX and NFIX instructions are decoded and executed in the same way as all the other instructions. The decoding logic does not have to bother with the decoding of the different operand sizes. This simplifies and thus speeds up instruction decoding. Secondly, higher language compilers are simplified because every instruction can take operands of any size. Thirdly, operands are represented in a form independent of the processor word length. Transputers with a larger operand register will always be able to use executables of older transputers with a smaller operand register. The scheme also has some less favorable points. Larger operands are built 4 bits at a time. Every step 4 bits are "wasted" by the instruction field. A 32 bits address for instance, takes 8 bytes instead of the usual 4 bytes. The time delay caused by having to execute 8 instructions is somewhat lessened by the transputer's internal pipeline, as we will see later on.

A consequence of this scheme is that only 16 instructions have immediate operands, values directly following the instruction. All other instructions have implicit operands, which means that these instructions only work on registers. This makes the decoding of these instructions simpler as no other addressing modes have to be taken into account. It is up to the user (or the compiler) to LOAD or STORE values to or from the registers. This again is a RISC like phenomenon.

The transputer has only three data registers A, B and C, which form an evaluation stack. Three entries on a stack hardly seem adequate, but local memory can be used when the register stack would overflow. Surprisingly enough, the micro-code does not test for stack-overflow. So, when writing assembler, it is up to the programmer to make sure stack overflow does not occur (unless it is intended) and that registers are saved in memory when necessary. In higher level languages, such as OCCAM, the programmer does not need to worry about the stack, as this is done automatically. These instructions refer to the stack implicitly. For example, the add instruction pops the first two numbers off the stack and pushes their sum. Inmos states: "Statistics gathered from a large number of programs show that three registers provide an effective balance between code compactness and implementation complexity." [08INMO89]. As

[01FLYN66] shows, even only a few registers drastically reduces the processor-memory traffic, because temporary results can be held in the registers. According to Inmos [07HARP89], the number of such temporary results can be minimized by careful choice of the evaluation order. They even supply an algorithm to perform the optimization of stack usage. A stack is used because it removes the need for instructions to re-specify the location of their operands [07HARP89].

Until now we have only examined an instruction set of 16 one-byte instructions. This is a bit too modest. Luckily, one of these instructions, OPR n (Operate), provides a way to extend the number of instructions. OPR uses its operand as an opcode number, resulting in another 16 instructions which can be coded in one-byte. Of course, these instructions cannot have any operands for themselves, they have to use implicit operands (registers). Since the operand of the OPR instruction may itself be extended by PFIX and NFIX instructions, the instruction set of a transputer can be arbitrarily large. Depending on the size of the O-register, which will be N bits wide for the example, there are $32 + 2^N$ possible instructions. As it is unlikely that this number will ever be reached when using 32 bits registers, term 'arbitrarily large' is used by INMOS.

This does not mean that many instructions will be needed. The T414 for instance has around one hundred instructions, whereas the T800 has some additional instructions (due to its FPU and some graphics instructions). Using this scheme, new transputers can have extended instruction sets while retaining binary compatibility with older models. The disadvantages of a microprogrammed processor as mentioned in [16COII9x] do not apply fully to the transputer, when new instructions are added.

While micro-programs do become longer and more complex in new transputers, this is not because of the problems to maintain compatibility [16COII9x]. As we have seen, with the transputer's expandable instruction set, compatibility is in fact easily retained. The transputer's micro-program becomes longer only because of the added functionality.

The most used instructions of the transputer's CPU are assigned to the 32 available one-byte opcodes. Many of these instructions require one or two processor cycles. Again this is a typical RISC-trait. Inmos' measurements show that 70% of the instructions in programs are encoded using these one-byte instructions [07HARP89] which keeps the sizes of executables small. These one-byte instructions include simple ones such as "load constant", "add constant", "store", and "jump". Most of these instructions use just one processor cycle. Compact sequences of these instructions allow efficient access to data structures, and provide for simple implementations of the static links of displays used in the implementation of high level programming languages such as OCCAM, C, FORTRAN, Pascal or ADA.

Features which are common in RISC processors can be found in the transputer's CPU too. Functions for which CISC processors have special instructions must be built up from simpler instructions which are executed in less time due to less decoding overhead. In other words: the run-time complexity is moved to compile-time. As a result, the transputer does not have, for instance, an extensive range of comparison and conditional jump instructions. In fact, only EQC n (equals constant) and GT (greater than) are available. All other types of arithmetical comparison must be evaluated by using these two instructions together with the available logical and bitwise instructions.

Although elegant, this scheme for an open-ended instruction set has one small rigidity. The 32 one-byte opcodes are already occupied, which means that they are not available for new instructions in future transputers without loosing downward compatibility. New instructions will always use more than one processor cycle. Using one PFIX or NFIX instruction before an OPR instruction makes the instruction numbers -256 to 255 available, so one extra cycle is needed to make (512 - 16 =) 496 extra instructions available.

External memory takes longer time to access than the on-chip memory. To minimize the memory access time, the transputer has some special features. To start with, when data is to be written to external memory, this data is stored in the output buffers of the on-chip memory interface [17RTSI9x]. The CPU

does not have to wait until the write process has finished, it can proceed with the next instruction. Only when the CPU wants to access this memory location, it is forced to wait. Even then it is not idle because other processes can have their time-slices.

Another facility of the transputer is the built-in user transparent pipeline. As memory access is word-size only, several instructions can be fetched in one cycle. The transputer has a one-word prefetch buffer, which makes the instruction fetch pipeline two words long. As we have seen, due to efficient encoding, instructions are just one byte long, which means that in a 32-bit transputer (such as the T414 and T800) the pipeline can hold 8 instructions at a time. When the program to be executed is stored in internal memory, the transputer rarely has to wait for an instruction to be fetched. This speeds up execution.

The pipeline scheme used in the transputer is fairly simple compared to other processors. There are no techniques used to reduce the "branch penalty" (instructions that are unnecessarily fetched when the CPU jumps to another location). Since the prefetch buffer is only one word long, the "branch penalty" is quite small. Only when a branch is taken, one memory cycle is wasted (note: It costs two cycles to fill the pipeline with two words. The first word contains the branch instruction. So, that memory cycle is not useless, but the second memory cycle is, because the instructions in that word are not executed due to the branch).

Another reason why the transputer is microcoded is, it was a design choice to make the compilation of OCCAM program as easy as possible. In order to do this, functions were moved from the programming language to the processor. This is easier to do when a microcoded processor is used instead of a hardwired processor. A further benefit is that (OCCAM) programs can be compiled to a very compact form which gets the most out of the (limited) on-chip memory. Of course, when code has to be retrieved from memory outside the transputer, there is a time penalty. The high instruction code density then helps to reduce the amount of processor-memory traffic. (Note: the time penalty is the highest when code has to be retrieved from the transputer links, which are serial. The transputer can address 4 Gb DRAM directly using its memory interface, so that code does not always have to come over serial links.)

As an illustration of the fact that the instruction set has been strongly adjusted to OCCAM: There are special instructions in the instruction set to implement the OCCAM PAR and ALT constructors. 11 extra instructions were needed to implement ALT !

A side-effect of moving instructions from the programming language (OCCAM) to the processors micro-code, is that it becomes nearly impossible for the OCCAM compiler to optimize code.

The transputer is usually programmed in a high-level language, of which OCCAM gives the best performance. Programming in assembler is, according to INMOS [08INMO89] not necessary, but can be used for a more efficient memory management scheme or communication with peripherals, for manipulating internal scheduling and communications mechanisms or when recursion is needed.

4. Formal Correctness

As we mentioned earlier, OCCAM and the transputers instruction set have been designed to be used together. OCCAM is based on Hoares 'Communicating Sequential Processes', CSP for short. Due to mathematical nature of CSP, OCCAM has been proven correct (sound). It can be determined whether two programs are semantically equivalent, by using a set of transformations. When a FPU was added to the transputer (ANSI/IEEE Standard 754 for Floating Point Arithmetic, 1985), INMOS had to revert to the Z specification language and ML (a functional language) to prove correctness [03HOME87]. The proof was then extended to prove the micro-code of the transputer. This last step was done by computer.

5. Processes

Until now, we have only dealt with sequential instructions and everything seemed reasonably straight forward. But the transputer's instruction set also contains several instructions concerned with the execution of processes in parallel on the same transputer. In order to do this, the transputer will have to be able to switch from the state of one process to the state of another. The information representing this state, commonly known as the process description, will have to be saved. Preferably to (on-chip) RAM, because this switching has to be done quickly (especially in a real-time environment). So, the transputer supports fine grained parallelism, but in such a way that the programmer / compiler explicitly has to indicate parts which can be executed in parallel. The transputer cannot detect parallelism by itself.

The transputer's implementation of concurrency is closely based on the concurrency model of OCCAM. The transputer has special instructions (such as STARTP, STOPP, RUNP) to manipulate processes, and it has a special microcoded scheduler. The latter is in contrast with the conventional way of implementing a scheduler in a software, typically a special process in the operating system's kernel, which is triggered by a (clock-) interrupt or a trap (software interrupt). A scheduler in hardware has the usual firmware software pros and cons. With an on-chip scheduler, context switching can be very fast but on the other hand the scheduling is not very flexible. For some applications it could be preferable to use another scheduling algorithm; in a software scheduler this can be done quite easily.

A process on the transputer is described by several pieces of information, such as registers, workspace, program and priority. Such a process does not have to be a sequential process but can also consist of several subprocesses.

Processes on the transputer can be separated in two categories:

- active (executable) processes
- inactive (suspended) processes.

An active process is a process which is being executed or which is waiting for its next turn to be executed. An inactive process is a process which is waiting for interprocess communication or which is suspended until a certain specified time. For keeping track of these processes the transputer maintains in total 4 lists. The transputer has two priorities in which it can run - as we will discuss later on - and for both priorities it has two lists for the above mentioned categories of processes. Of course, the active process which is being executed at the moment does not have to be on the active process list because its process information is loaded in the registers of the transputer's CPU. The transputer contains two registers for each of the four lists.

Processes on the transputer can have two priorities: high or low. A high priority process which is executing on the transputer will run until it wants to stop. After that the next process on the high-priority executable process list is executed. When there are no more high priority processes, low-priority executable processes are allowed to run. Although there are only two levels of priority, this feature can be used for several purposes. For instance, a high-priority process can deal with interrupts (from the EventReq pin) so that when an interrupt occurs this event can be processed quickly. A user program does not have to run in high-priority as a whole. Only the part which handles interrupts needs to be run in high-priority mode.

The priorities of processes can be set by the assembly language programmer using a special instruction (RUNP; [11MITC90]). In OCCAM, priorities are statically determined at compile-time by the programmer using the PRI ALT and PRI PAR constructions.

A process (low or high priority) will be descheduled when one of the following conditions occur:

- The process executes an instruction in order to communicate with another process.
- The process executes the TIN (=Timer Input) instruction which causes it to wait until a specified time. In the case of interprocess communication the process will then be put on the list of inactive processes for that priority. Here the back-of-the-list pointer is used. One of the differences between low- and high-priority processes is that low-priority processes must share the CPU (pre-emptive multitasking). So, when the process is a low-priority process, there is another condition under which the process will be descheduled.
- The low-priority process has used up all its time-slice.

Low priority processes are subject to round-robin scheduling [19TANE87] with a time-slice period of about 1 ms in a T800. But there is a limitation: descheduling due to the expiration of a time-slice can *only* happen after the execution of certain instructions. These instructions are:

- an unconditional jump (J; jump)
- a special instruction which is very often used in loops (LEND; Loop End)
- several others

As a result, a particular low-priority process which cleverly avoids these instructions can dominate the other low-priority processes[1]. On the other hand, the scheduler does not consume any CPU time for processes which are descheduled, that is, for checking when it is another process' turn [07HARP89, page 18].

According to INMOS, in some instructions (J, LEND and others) there is almost no information in the registers so that very few registers have to be saved in memory. Process switching takes longer as there are more registers to be saved to memory.
When a high-priority process wants to run on the CPU (for instance, because of an interrupt) during the execution of a low-priority process, it is not desirable to wait for the low-priority process to execute a jump or a loop end. In this case, all the process information - such as the evaluation stack - is saved in on-chip memory.

A process running in high-priority can monopolize the CPU, as it cannot be descheduled. High priority processes are expected to execute for short time intervals but what if they do not? Maybe the use of, for instance, a monotonic rate algorithm [17RTSI9x] for scheduling high-priority process could be useful, but then only for cases where there are more or less predictable deadlines. The bottom line is that high-priority process must be very cautiously designed in order to insure that low priority processes do proceed as well.

The transputer has two timers, one that gives a tick every microsecond and one that gives a tick every 64 microseconds (for the 20 MHz T414). This can be considered another inconvenience because the two timers are associated with a level of priority. Low-priority processes cannot use the high-resolution timer. This means it can happen that processes run needlessly in high-priority, all because of the fact they have to use the high-resolution timer. A typical process switch takes about 1 microsecond (on a 20 MHz T414) [17RTSI9x, H13, page 13], with an upper limit of 4 microseconds. The exact duration of a process switch depends on the priorities of the processes. Instructions which take variable clock cycles (such as the MOVE (block copy) instruction of the T800 [11MITC90] are constructed so that they can be interrupted for a process switch.

The number of processes allowed is only bounded by the amount of free (on-chip) memory available for the processes and their workspaces [17RTSI9x, H13, page 13]. Each of the two lists with active processes has a register which points to the front of the list, and a register which points to the back of the

---

[1] Avoiding these instructions can be done quite easily. For instance, instead of the unconditional jump J, the programmer can use a conditional jump with the condition fixed on FALSE. That is: LDL #0, CJ #xxxx.

list. The latter makes it possible to add descheduled processes quickly at the back of a list, though it uses up extra on-chip space and time during a context switch.

6. Transputer Based Systems

We can identify three classes of hardware applications for transputers: in an accelerator board- in embedded systems- in (relatively) stand-alone MIMD machines.

Add-on boards with transputers

Especially the early transputers were used in accelerator board for conventional systems (e.g. IBM PC and DEC VAX). Add-on cards with one or little more transputers offered a good price/performance ratio for those who were willing to recode their applications into OCCAM. One example is INMOS' TDS, Transputer Development System.

Some years after the transputer appeared on the market several processors were introduced that clearly outperformed the transputer in raw computing power. [03HOME87] contains some results of floating-point benchmarks. The transputer comes out best of the then available processor/coprocessor combinations, according to this paper. But it must be stated that using the transputer as a floating point add-on processor is only efficient when the nature of the floating point problem is 'coarse grain'. With this we mean that the transputer has to do its work on its own, without too much communication with the host system. Only then the data traffic to and from the host processor will be acceptably high. Do not forget the transputer uses serial links (although high-speed).

Add-on boards are suitable for several applications. For instance, the real-time nature of the transputer makes it possible to use it in a simulation system for logical components, VLSI simulation [07HARP89, page 221, 04MUNT88, Welch page 145].
INMOS also supplies TRAMS for its transputers. TRAMS are modules containing a transputer and some other circuitry, such as extra RAM or a device controller. TRAMs are also equipped with an INMOS standard interface which makes it possible to mount it onto a variety of motherboards with specific host interface hardware. This way the same TRAMs can be used on different host systems.

Using this modular system has a great advantage: expandability. Users can expand their systems in time as their need grows. As TRAMs contain RAMs as well as transputers, expanding the system means increased processing power as well as more storage capacity so that these two factors remain in balance. Transputer software emulators are also available, which means that the use of transputers can be evaluated without purchasing extra hardware [04MUNT88, page 281]. These emulators can futhermore be used to separate the software development system from the production system.

Transputer based embedded systems

An important application for a device as the transputer is an embedded system. And that is not only from the manufacturer's point of view, so higher volumes of these devices are sold. They can also be applied in devices such as laser printers, hard and floppy disk controllers and are often used in scientific projects. In fact, INMOS even supplies special transputers as disk controllers. Again the real-time aspects of the transputer make this possible, in addition to the communication facilities. Other examples of the use in embedded systems are: graphical applications such as simulators and laser printer controllers (the latest transputers contain several instructions such as Draw2D and Clip2D for graphics applications); space-borne devices [07HARP89]; robot control and sensor data processing.
A common use of the transputer add-on boards mentioned above is as a development platform for embedded systems. Software for the embedded system can be edited, run and debugged from the transputer board. In the production phase of the embedded system, the ROMs or EPROMs from which the transputers will boot can be downloaded from a link.

MIMD transputer machines

When transputers are used as MIMD machines, a lot of the problems common to MIMD appear:

Topology

In a multi-processor system, processors need to communicate with each other. This is dependent on the topology: the way in which the processors are connected with each other. In fixed-communication systems a processor can only communicate directly with its neighbours while in general-communication systems each processor can communicate with any other processor. The advantage of a fixed-topology machine is simplicity. Processors are connected in a regular pattern. In some cases fixed-topology machines fit very well for a particular application, for instance a two-dimensional grid for image processing. General communication networks are easier to program for a wider range of applications, especially those who do have irregular or dynamic changing patterns of communication.

When choosing a topology several considerations can be made. These include:

- Diameter. This is the maximum number of "hops" a message has to travel from one processor to another. If the diameter is small, communication is likely to be quicker.
- Extendibility, so that a network of processors can be built of (preferably) any size.
- Redundant paths, so that in case of failures messages can be routed via a detour. Redundancy will also distribute communication over several paths which prevents "hot spots" (bottlenecks).

Of course, topology does not determine overall performance. Much depends on the actual problem, whether communication can be dealt with locally (that is on-chip or with neighbouring processors at most).

Some of the most used topologies for transputer systems are:

- Crossbar networks, in which every processor is connected to any other processor. INMOS supplies a circuit-switching chip, the C104 so that crossbar networks for transputers can be easily implemented. The Meiko has this kind of switch.
- Mesh networks, such as grids and toruses. These networks often match specific (two-dimensional space related) applications very well.
- Trees.
- N-hypercubes and other networks whose diameters size logarithmically with the number of processors. Each processor is connected in a n-cube manner with n other processors.

The transputer contains four links. This imposes restrictions on the topology of transputer networks. For instance, only hypercubes of degree four (16 processors in total) can be constructed. Performance depends on the interaction between the topology and the parallel programming paradigm used. Several are presented in transputer related literature:

- Algorithmic parallelism. The application is split in functional units. For simple cases, they can form a pipeline. Stages can perform in parallel, while communication is buffered so that the processor does not have to wait for I/O. The slowest stage limits the maximum performance.
- Geometric parallelism. Many (physical) applications have an underlying regular pattern with limited spatially interactions. A processor can be assigned to each spatial area, while communication is often to local neighbours only. This paradigm requires only a fraction of the total data on each processor though often the whole program must be resident as well.
- Processor farms. Sometimes applications can be divided in small, similar pieces. These jobs can be done without knowledge of previous calculations. A 'server' distributes ('farms out') the pieces over several 'slaves' when they finish their previous jobs. Each slave receives a workpacket which can contain a large set of data. Extra storage facilities may be required though less communication between processors is involved during calculating.

- Hybrid forms.

Mapping

Processes will have to be distributed over the whole system. Especially in large systems this can be hard for the programmer to do manually. Care has to be taken of load-balancing as well. A system is said to be loadbalanced when none of its processors force others to stay in an idle time for long. It would be nice to have a configuration tool which decides where processes and communication channels will have to be placed. Unfortunately, this problem was proven to be NP hard [04MUNT88, page 187] so that an efficient solution for all cases is not eminent. Several considerations will have to be made such as choosing between static mapping, dynamic mapping or new propositions such as post-game analysis. Tools based on heuristic algorithms have also been devised [04MUNT88, page 188].

Routing

When a transputer wants to communicate with another processor which is not its direct neighbour, it is clear that some kind of routing scheme is needed. Routing can be done statically or dynamically. Several algorithms have been devised in order to explore an in advance unknown network of processors [04MUNT88, page 188, worm program] [07HARP89, page 181]. Such an algorithm must be:

- complete (so all messages arrive)
- deadlock free
- optimal (packets take the shortest route)
- scalable (so that it can be used for networks of any size)

Message passing is typically implemented by some sort of "post-office"-like mechanism by which a message is forwarded to its destination. Other algorithms are based on multiplexing channels, i.e. implementing virtual channels. Virtual communication paths require duplicate queues and a complex protocol. Points of attention in packet-switching networks are deadlocks. Deadlocks occur when a routing algorithm is not properly designed and packets can no longer advance due to full buffers. [18GUNT81] gives an introduction in the prevention of deadlocks. When arbitrary communication between processors is needed for an application, the user has to implement his own message passing algorithm. This functionality is in fact often needed; to quote [12BOAR90, page 9]: "We suspect that store-and-forward functionality is so universally useful that it should be provided as an atomic transputer capability".

7. Operating systems

The need for an Operating System

Does a transputer based system need an operating system? Of course, this depends on what the transputer is intended to be used for. One of the most important reasons to use an operating system is to "virtualize" the system's resources, such as I/O, memory, CPU, so that different programs can share these resources. In a multi-user environment the operating system also has to provide some means of protection between users. Virtualization also improves the portability of software. Another advantage of virtualization is that, with luck, application programmers do not have to bother with low-level hardware specific matters; the operating system supplies the programmer with an easier to use development system. On a multi-tasking system, the operating system normally has to keep track of all the processes which are running, suspended, waiting, etc. But an operating system is not always needed. An important application field for transputers are embedded systems. In embedded systems, the given situation (hardware, software) is not very likely to change. Programs interact directly with the given hardware; system software to keep up with these changes is not needed. In addition, when there is no extra layer of system software, there is no associated overhead in memory space and execution speed as well. So, in the case of embedded systems, an operating system is just convenient for the programmer.

In the Transputer Development System (TDS) a lot of operating system functionality is handled by the host system. For instance, when a transputer wants to perform some I/O it has to communicate with a transputer connected to the host system. In such a case, a fully functioning transputer operating system is not used, but each processor must at least contain some kind of communication kernel in order to pass messages around to the host's operating system.

In many cases, a full-blown operating system is not needed but some of its functionality is. For some applications, such as the Fast Fourier Transform (FFT), the use of Unix for instance, results in a lot of unnecessary administrative overhead. It is imaginable that in a multi-transputer system, some transputers run the full operating system because they provide the interface with the user(s), while other transputers only run a real-time kernel for time-critical applications and the rest of the processors run a kernel of a small subset of Unix-like calls for doing a FFT.

An Operating System on a MIMD System

In other cases, a native transputer operating system comes in handy. But there are problems. Most existing operating systems are essentially meant for uni-processor systems. In such an operating system all decisions are taken in a centralized way, by one processor. It is not likely that such a scheme is very efficient and fault-tolerant when it is directly adapted to a multiprocessor system. Without rewriting major parts of such a operating system they are very difficult to port to a multi-processor environment. Even when an operating system is able to manage multi-tasking, this does not guarantee its portability to a MIMD system. These problems are caused by a number of reasons. First of all, the topology of a MIMD system does not necessarily have be known to the operating system in advance. Ideally, the operating system has to take care of that, so that the user/programmer is not bothered by such a task. Secondly, it is nice to have the operating system handle topology changes, such as when extra processors are added or when malfunctioning processors are shut down. When the owner of a system buys new processors, old software should be able to run as normal, with a scaled up performance. Also, all different kinds of hardware have to be supported by the operating system. For instance, earlier in this document we have seen a range of different transputers available. The transputer's design covers up a lot of differences between the several types of transputers, but the operating system still needs some information about the particular nodes in a system in order to distribute tasks, etc.

The operating system should be able to take care of all above mentioned points. Otherwise, chances are that very few applications are going to be available for such a system. What good is "shrink-wrapped" software which does only run on one specific system with 42 T414's in a 2D grid network?

An Operating System on a Transputer System

There are special considerations a transputer operating system should make because of the very nature of the transputer. The transputer contains an instruction set which is unusual in the way that it supports different functions more commonly handled by the operating system. The notion of "processes" is directly available at machine language level. This means that processes can be dealt with in a very efficient way. On the other hand it is arguable whether this kind of light-weight processes correspond the process model in a modern operating system. But there is no reason why a transputer operating system should not exploit the availability of these processes, albeit perhaps with some extensions. A real operating system type process will probably need a wider range of priorities, as opposed to the native "high/low" transputer processes.

Also, the transputer's built-in communication mechanism is an extra which the operating system has to take into account. The OS can use it for its own purpose (internal queues, for instance) but also it can increase the reliability of the system by prohibiting the direct use of the I/O mechanism by application programs.

While the above mentioned features can be quite useful, the transputer lacks some serious features commonly found in processors which are considered essential for a modern operating system. For starters, the transputer does not have a memory management unit (MMU) [19TANE87]. A MMU is convenient for a number of reasons. It aides in the implementation of virtual memory. Pages of RAM, connected to inactive processes, can be swapped to disk. In the transputer, without its MMU, when a program needs a large address space, it will have to use real memory. Such a transputer will have to be supplied with a large quantity of DRAM via its external memory interface.

A very important application of a MMU is the possibility of a private memory address space for each process. A "flat" address space is convenient for the programmer. Furthermore, a MMU provides protection between processes so that a user's process cannot read or write in the address space associated with another process. This is particularly dangerous when the address space of a system process is concerned. A MMU can be extended in order to make the sharing of code areas possible, so that the same program can run with different data using the same code, saving precious (on-chip) memory. In theory, the tasks of a MMU can be emulated by software but this will result in a severe performance loss.

So, why is there no MMU? It seems that the lack of a MMU on the transputer makes it unsuitable for (high) security systems. There is a technical reason: instructions are not restartable, in case of an occurring page-fault in the MMU. But according to [07HARP89, page 88] there is also a design decision behind it. The transputer is intended to be used with other transputers, each with its own memory space. The original transputer design envisaged each process running on its own processor. Then, protection between processes is the physical separation between processors. Communication is done by strict, controllable means: links. An advantage of such a scheme is that it is more reliable in terms of protection between processes. Badly written or malicious processes cannot touch the address space of other processes, using errors in the operating system or else. On the other hand, one process per processor is highly expensive and not realistic.

The transputer also lacks a privileged execution state. Such a state is mostly used to distinguish between system processes and user processes. A process in privileged execution state is entitled to access strategic system resources. Without it, virtualization of these system resources (and thus protection) is

more difficult. The priority scheme in the transputer is insufficient. Although it is true that some machine language instructions can only be executed in high-priority, a user process can easily obtain high-priority.

Several Operating Systems in more detail

All the above show that it is difficult to port an existing operating system to a MIMD environment, let alone a transputer system. In the remaining part of this chapter we will discuss some (commercially) available operating systems for the transputer, in which we will address the problems associated with it in some more detail.

Express

[07HARP89, page 86]
Express is based on some of the early development work in parallel computing which took place at the California Institute of Technology. It consists of a message-passing kernel and a set of library calls which provide the interface in order to create a parallel programming environment. I/O can be done to the console through a network of processors without knowing the topology in advance.

Unix

[07HARP89, page 88,89]
The lack of a memory management unit has seriously hampered the development of a Unix version for the transputer. There is in fact one port, Idris, which is a derivative of the older Unix V7 which does not explicitly require a MMU. The system provides a POSIX compatible set of calls. The Idris implementation for the transputer runs in one processor - which makes it quite vulnerable in our opinion - while other processors are equipped with a communication kernel through which it is possible to start up other processes.

Trollius

[07HARP89, page 90]
Trollius is a message passing operating system for parallel architecture computers, where the computer can be a transputer-based node or a conventional Unix-machine. Several transputer based machines are supported, such as Meiko and Parsytec. Trollius consists of a kernel for the transputer based machines and provides a set of library calls for it.

Meiko

Meiko supplies a transputer based system with the name "Computing Surface". A Computing Surface can be shared by different users at the same time through "Meiko Multiple Virtual Computing Surfaces" (M²VCS). It provides a way of dividing a single Computing Surface into several virtual Computing Surfaces. Each of these virtual machines is made up of a "domain" of processors. A domain consists of

building blocks known as "elements", see figure 5.

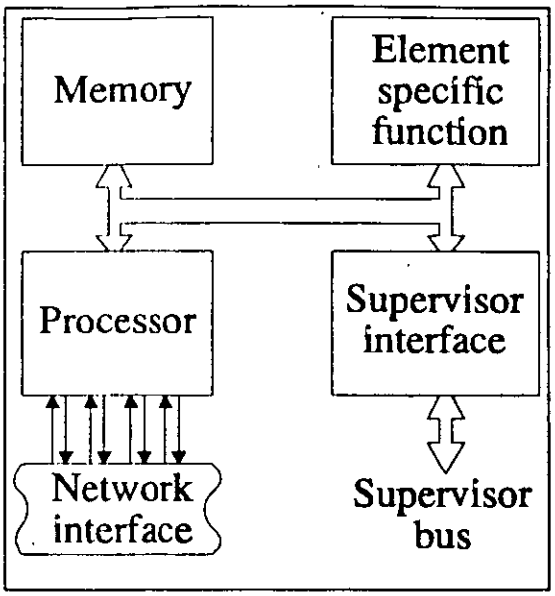


figure 5 : General diagram of an element

Examples of such elements are: compute elements, periphery elements (interfaces to other systems), mass store elements and display elements. The transputer links in a Computing Surface can be configured dynamically. This is done by a cross-bar switch, so that different topologies can be tried out by the user for his application. A cross-bar switch can be compared with a electronic telephone exchange station. The domains are connected by a central "spine" of dedicated transputers and system software called the Computing Surface Network (CSN). The spine can be any non-cyclical shape, such as a tree or double spine.

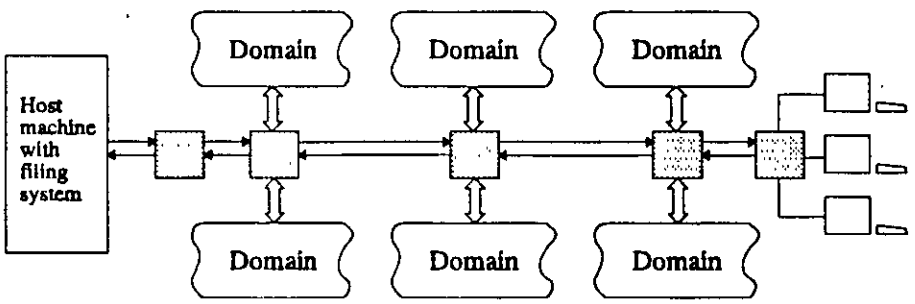


figure 6 : An example M²VCS configuration.

M²VCS provides protection between domains so that user processes in one domain cannot corrupt the rest of the Computing Surface. This is relatively easy to do because domains are already physically separated. This is exactly following the design which is the cause of not having a MMU on the transputer.

The CSN works on a client/server basis. That means a system resource is connected to the network by a server process associated with it, and users wishing to access this resource can do so by sending messages through a client process which communicates with the server. If many clients want to access a single server, this may cause some throughput problems as the CSN itself is a shared resource. This can be compared with clients calling the same telephone number. If the network becomes saturated (that is, messages take a very long time to arrive at the server) it may be needed to redesign the spine which forms the communication network of M²VCS. For example, when a file server is used by a lot of clients, it may be advisable to build a tree network with the server at its root. In that case the network bandwidth is more balanced than with a pipeline topology.

Every terminal connected to the system is associated with a login shell through the CSN. The user has to supply a password to the shell which is validated by a password server. After a successful login the user can interact with the shell by typing commands. At that moment the user is connected to a particular processor within a domain. This processor is called the "user seat". Onto this user seat the user can download an operating system and start running programs. Several operating systems are allowed for a user seat: Meikos (Meiko's own Unix-lookalike single user operating system), OPS (Occam Programming System) or Helios. These separate operating systems may share system resources. For instance, the files supplied by a file server are accessible by all operating systems while the file server handles the file protection. The CSN handles the translation of the local system's calls into the remote resource's calls transparently.

Once a user has obtained a user seat he can start developing or running programs on the other processors within the domain, without the risk of corrupting the work of any other user working in parallel with him. This is more or less something like a network of conventional computers using a conventional operating system, while it is also possible to use parallel programs directly.

Helios

[04MUNT88, page 41] [20SCHA90] [07HARP89, page 91]

Helios is a distributed operating system for transputer systems. It is influenced by the Cambridge Distributed System and Amoeba and it is similar to Unix at user level. In a distributed operating system there are no vulnerable central services upon which the whole system relies and which can become a bottleneck. In Helios this distributed nature is transparent to the user as well as to programs running within it. They do not need to be aware of the exact location of services. Helios' main goal is to make the different network topologies and number of processors as transparent as possible so that software (even binary object code) can be used in different environments.

Communication between processes is done by message passing. This is achieved through message ports, logical point-to-point connections. The physical links of the transputer can not be used directly by the user because of the danger of corrupting the system, but are managed by Helios. The semantics of message passing on the transputer are the same regardless whether the destination process is in the same processor or another. The routing mechanism for the message passing is built when Helios first attempts to locate a service. This is achieved through a "distributed search" algorithm which locates the processor on which the server resides. Instead of processes - which are handled efficiently by the microcode - Helios manipulates *tasks*. A task is a program in the state of execution, consisting of one or more processes. Data such as open files and memory are associated with a task rather than with processes.

Helios uses the client-server model as a concept (compare with Meiko). All servers are state-less which means that in case of a failure of the server no information about its state is lost. And in case of a network failure the message can be repeated without harm. This also means that messages from the client will, for instance in the case of a file server, have to contain file identification data and the position in the file. Multiple servers can coexist. When a message is sent, it is duplicated in the network and the quickest server responds.

All system servers are accessible using the General Server Protocol (GSP). Network resources such as processors, files, devices are all considered as objects. They are identified by a unique name and are ordered in the same way as a hierarchical file system (such as Unix). There is no central name server as this would surely become a bottleneck. Instead, each processor contains its own name table and when an object is needed that is not present on the processor itself, a global search will be done. All objects can be accessed in the same orthogonal manner. That means, for instance, not only files can be listed and removed using the commands "ls" and "rm" respectively, but also tasks. Objects in Helios - which after all is a multi-user system - are protected by capabilities [21TANE86]. A capability can be compared

with an entry ticket you have to show every time to the porter. Capabilities hold access rights to an object, and they are encoded by cryptographic means. When an object is created, a capability is created as well. Every time a client wishes to access this object it has to show its capability to the server. The server then can decode the capability and verify the client's access rights to the object. Capabilities for file systems are long lived and can be stored in files, directories or user programs. Once a user has identified himself to the system by the usual way (logging in), he obtains a capability for his home directory which contains capabilities for all the other objects he has access to. There is no super-user (hoi-hoi!), all privileges will have to be obtained by having capabilities with more access rights than others.

The Nucleus is the minimum system that must be present on every processor running Helios. Its purpose is to control the resources of a single processor and to integrate it in the whole Helios system. It is written in assembler in order to make it as small and efficient as possible. The Nucleus consists of four parts: the Kernel, the System Library, the Loader and the Processor Manager. The Kernel implements the message passing mechanism as already described and also manages the hardware resources of the processor. For instance, it controls the allocation of on-chip RAM and external RAM. It does not do process management as this is done on a lower level, as already stated. Also it does not provide protection between address spaces due to the lack of a MMU on the transputer. The System Library provides the equivalent of system calls of a conventional operating system. Its main task is to implement the "General Server Protocol" through which all Helios servers can be accessed. In addition, there is another set of calls which implement a Unix-like duplicate. These calls follow the POSIX standard to a large extend, but are in general not as efficient as the "native" set of calls. The loader has the task to load program code and data (such as bitmaps and fonts) to memory. It also unloads them as they are no longer needed. The Processor Manager creates Tasks, manages them while they are running and removes them when they terminate. This is done through the means of an I/O controller, a special process associated to each Task which supplies the interface between the Task and the rest of the system. Run-time failures and exceptions are also handled by the Process Manager [20SCHA90].

8. A large case study,
IBM's Victor V256 and some of its applications
[13SHEA91]

In this section we discuss the IBM Victor V256 partitionable multiprocessor. It is a message passing system with 256 Inmos T800 transputers. All the memory is distributed; this was considered essential for significant speedups. IBM chose to use transputers because of its specialized on-chip interprocessor-communication support.

Experience
A lot of work is done in the field of parallel processing. In designing the V256 experience was gained by studying the following project:

- **Cosmic Cube**, a 64 node machine based on the Intel 8086, made by the California Institute of Technology, a pioneer in this field.
- **Armstrong**, a 100 node machine based on the Motorola 68010, made by the Laboratory for Engineering and Man/Machine Systems at Brown University. Here reconfiguring the network has to be done manually using "patch-cords", this, however, can be done while the system is running. This also increases the fault-tolerance. The network layout is completely masked by the operating system. Also 32081 coprocessors have to be used for floating-point operations.
- **Hathi-2**, a 100 node machine based on the Inmos transputer, made by the Department of Computer Science at Abo Akademi in Finland. Each of the nodes has 256Kb of memory and the network configuration is set with manual switches on the backplane. Yet reconfiguring the network does require a reboot. The granularity for this system is 4 transputers, whereas the V256 transputers can be partitioned per processor.
- **Esprit P1085**, a 17 node machine based on the Inmos transputer. 16 of the nodes have 256Kb memory and a 17th transputer has 16Mb memory, this is used for storing and distributing data. The reconfiguration hardware allows any network setup that can be created with nodes having four links. The system provides more reconfiguration capability in hardware than Victor but lacks the hardware monitoring facilities.
- **V32 and V64**, IBM built smaller models of this machine, this gave them insight into working with transputer in this type of system. Many of the examples that will be mentioned were also tested on these machines.

The Victor system
The Victor system has four types of nodes (*processor*, *disk*, *host* and *graphics*), all using the T800 transputer. The processor nodes of the V256 form a 16x16 mesh. The host nodes are connected to the corners of the mesh, thus allowing four users on the multi-user system. A fifth host node is connected for the super-user.
The processor nodes are built up of one Inmos T800 running at 20MHz, 4Mb RAM and some partitioning and monitoring hardware.

The I/O speed of the system is not ideal. According to Langdon [22LANG82] the I/O capability of this system should be 100 MB/s, the V256 only reaches a discouraging 5 MB/s. This has nothing to do with transfer speed of the transputers, the bottleneck is the transfer rate of the 16 harddisks. The harddisks can be accessed in parallel and have a transfer rate of 310 KB/s. Each of the harddisks is connected via a SCSI bus to separate T800 which in turn connects to the 256 transputer mesh.

The partitioning of the mesh into four segments is controlled by the host-nodes. These nodes reserve "as many as needed" transputers for the client. A segment is usually rectangular. That is not essential but there must be a path from every node in the partition to the host. A partition size can range from 1 to 256 transputers.

The logic for the V256 is packaged on circuit cards. The cards are slightly modified off-the-shelf and contain 4 nodes. They all use the same oscillator, which is phase shifted before it drives each node. This phase shift is very interesting: Each card has 4 transputers and 16 megabytes of DRAM. The phase shift causes the hardware to refresh the DRAM at different points in the cycle, thus reducing peak current demand. An error correction code is added when each 32 bit word is stored into memory; it is a 32/39 ECC, which also provides double-bit detection of memory errors. These last two features and many more increase the system reliability. There have been few memory failures and only one link failure. The system dissipates 20 kW and is air cooled.

System software

Three message-passing environments have been tested on the V256. IBM does not mention which of these is preferred.

- **Communicating sequential processes.** This uses functions native to the transputer. These are based on the CSP paradigm introduced by Hoare. As many of the required features of the message passing environment were already implemented in hardware, IBM tried to use Occam as a base for all communications.
- **E-kernel.** The "embedding kernel" was intended to relieve the user of the concern of optimizing program communication for the system network topology. E-kernel is used both to optimize the performance of an application running on a Victor partition and to experiment with program performance for systems with different network topologies. The embedding is done in two phases. The first phase is program mapping, it maps the applications task graph onto the system network topology. The number of nodes in the task graph is assumed to be equal to number of processor nodes needed. The second phase is network reconfiguration. The E-kernel embeds the chosen network topology onto the 2D mesh of a partition on Victor. IBM does not mention how this is done in detail, but we assume there is some routing hardware on each node, as the T800 has no on-chip support for routing. The E-kernel was written in Occam.
- **Express.** This system is based on work done for the hypercube. Fox et al [23FOX88] have developed an environment called CrOS III (hypercube crystalline operating systems). In this system communication is grouped into "conceptual units". This has two reasons; the first is that it is easier for a programmer to think about communication and concurrency in terms of these units, second is that these units can be implemented very efficiently on the machine.

Applications

The following applications for the Victor system were chosen because they were real-world problems, not for ease of implementation on parallel system. Final results outperformed state-of-the-art mainframes. We will mention several of them because they give a good impression of what kind of problems can be handled by large parallel systems.

- Fractals and raytracing. These problems are typical problems for parallel systems. An near-lineair speedup is acquired when more processors are used. The balancing of the workload can be done dynamically and the only communication overhead is the transmission of parcels.
- Monte Carlo nuclear physics. Here the run-time was dominated by communication free computation. This also resulted in near-lineair speedups when more processors were used. The workload balancing formed more of a problem, as did the concurrent generation of pseudo-random numbers.
- Neural network simulation. Here communication took up a significant part of the runtime. The simulation consisted on two phases; in the first phase the entire neural network was reconstructed

on every node and parameters were calculated for the second phase. In the second phase the neural network layers were represented as rows in the processor mesh. Because a central control was needed for this problem, communications with the central node formed a bottleneck. Speedup 16 on a 32-processor system.

- Computational fluid dynamics. This involves solving of three-dimensional Euler and Navier-Stokes equations for compressible flow of gas over a solid body. Different partitioning schemes were tested and IBM concludes: the problem "scales well" in terms of execution time and memory requirements.
- Solution of linear systems using conjugate gradient method. Without mentioning any details, the efficiency was 0.976. As no time could be calculated for the evaluation of the problem on one node, a smaller problem was used.
- Logic fault simulation. Here a deliberately defective VLSI circuit is split into small clusters of gates, which are assigned to nodes. Independent clusters can be evaluated simultaneously. Then the results have to be shared; this makes this application very communication-intensive. The overhead of the store-and-forward message-passing plays up. Speedups were between 35 and 40 on a 256 node system. An interesting note is that the speedup showed no sign of leveling off at 256 nodes.
- Multirobot simulation. The problem was reduced to two robot arms with one prismatic and five revolting joints each. Each joint ran on one path-planning node and one graphics node, all data was then collected in another graphics node. The system was then able to produce 2.8 animated steps per second. Speedups were not expected due to the course grain implementation; it was adequate to run this on 16 nodes.

IBM summary states that their highest performance was 224 MFLOPS (32-bit arithmetic), this was reached with the fractal program. The applications parallelized well, even though the V256 is 'first-generation'. In some cases the performance on the V256 was superior to that obtained on a mainframe or a vector supercomputer.

Four significant reasons why speedups are not linear in the number of processors are distinguished:

- insufficient problem parallelism
- communication bottlenecks
- synchronization overhead
- workload imbalance

The researchers on this project reasoned that, for most of the applications, recoding the programs to run in parallel - at least on the V256 - was worth the time, in view of potential performance gains. From their conclusion we conclude that this was absolutely not the last that we have heard from IBM in this field.

9. Conclusion

One of the most striking features of the transputer is the integration on-chip of several functional units commonly found in external devices. For instance, the transputer is equipped with four hardware links. This can be considered as a built-in autonomous communication processor, as opposed to designs such as [14MOOI]. Chip designers who need good communications facilities, for instance for real-time applications, will at least consider integrating links on-chip. One of them is Texas Instruments which already ships the TMS320, a 50 MFLOP digital signal processing chip with 6 autonomous 20 Mb/s links with multi-channel DMA engines. The other functional units are the static memory and (in the case of the T800) the floating point unit. As a compromise, the VLSI implementations of these units are not always the most optimal, in order to save precious chip space. It is to be expected that other chip designers may follow this way as well. Another exciting feature is the extendable instruction set. This implies an upper bound decode time and a fixed size microcode decoding program.

Other manufactures have never implemented the decoding of instructions as expandable as INMOS has done with the transputer, this feature will have to prove itself in the future (perhaps with the introduction of the T9000).

OCCAM is not generally considered a pleasant language for writing applications, this increases the need for compilers for other languages. Writing compilers for the transputer is not an easy task, as we have seen. OCCAM creates the most efficient executables; this discourages writing in other languages.

The transputer is well suited for implementation in embedded and real-time systems. One of the features that makes this possible is the hardware support for manipulating processes. Although this scheme is simple, it is not flexible: if another type of priority scheduling is required, a solution will have to be programmed by the user, such as was encountered in [12BOAR90, page 40].

The transputer offers no kind of memory or process protection within the processor itself. This is caused by the lack of a memory management unit and a privileged execution state. Between processors this protection is very strong, as all input from the link can be checked. Operating Systems, when implemented for multiple users, can not assign different users to the same transputer without making sacrifices to speed. If needed, extra protection will have to be enforced in software. There is only a global mechanism for trapping errors: stopping the entire processor when an error is detected.

As shown, a general communications facility is required for many applications. Solutions will have to be implemented in software, for instance multiple virtual channels. Hardware support will alleviate this burden for users.

The announced top model transputer, the T9000, is said to have overcome some of the above mentioned limitations of current models. We will have to wait and see...

It is clear that traditional Von Neumann machines have (almost) reached their physical limits. Parallel processing has found its place in computer science research. One of the candidates as building blocks for parallel machines is the transputer. The switch to using transputers has not been an easy one and will not be so in the future but programming a Cray II was not easy either...

References

[01FLYN66] M.J. Flynn, Very high-speed computing systems, IEEE proceedings, December 1966

[02HILL85] W.D. Hillis, The Connection Machine, MIT Press, 1985

[03HOME87] M. Homewood et al., The IMS T800 Transputer, IEEE Micro, October 1987

[04MUNT88] T. Muntean, Parallel Programming of Transputer Based Machines, IOS, 1988

[05INMO88] INMOS, Transputer Instruction Set, Prentice Hall, 1988

[06DIES88] R.J. van Diessen, Transputer als bouwsteen voor parallelle architecturen, afstudeer scriptie UvA, 1988

[07HARP89] G. Harp, Transputer Applications, Pitman Publishing, 1989

[08INMO89] INMOS, The Transputer Databook, Redwood Burn, 1989

[09WAAR89] H.W. de Waard, Transputers, A.W.B. Uitgevers, 1989

[10WHIT90] C. Whitby-Strevens, Transputers, IEEE Micro, December 1990

[11MITC90] D.A.P. Mitchell et al., Inside The Transputer, Blackwell Scientific Publications, 1990

[12BOAR90] J.A. Board, Jr. ed., Transputer Research And Applications, IOS, 1990

[13SHEA91] D.G. Shea et al., The IBM Victor V256 partitionable multiprocessor, IBM Journal Of Research and Development, September/November 1991

[14MOOI] W.G.P. Mooij and A. Ligtenberg, Architecture of a Communication Network Processor

[15NETN92] Internet news comp.sys.transputer, several articles, March through September, 1992

[16COII9x] L.O. Hertzberger et al., Syllabus Computer Organization 2

[17RTSI9x] J. Vermeulen, Syllabus Real Time Systemen en Interfacing

[18GUNT81] K. Gunther, Preventions of Deadlocks in Packet-switched Data Transport Systems IEEE transactions on communications, April 1981

[19TANE87] A.S. Tanenbaum, Operating Systems, Prentice Hall, 1987

[20SCHA90] J. Schabernack und Alois Schutte, Helios, ein verteiltes Betriebsystem fur Transputer-Rechner, R. Oldenburg Verlag, 1990

[21TANE86] A.S. Tanenbaum et al., Using sparse capabilities in a distributed Operating System, IEEE proc. conference Distributed Computer Systems, 1986

[22LANG82] G. Langdon, Jr., Computer Design, IEEE Computer Press, 1982

[23FOX88] G. Fox et al., Solving Problems on Concurrent Computers, Prentice Hall, 1988

[24INMO89] INMOS, The Transputers Applications Notebook - Systems and Performance, 1989

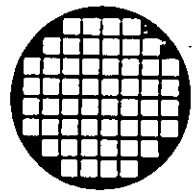
[25INMO89] INMOS, The Transputers Applications Notebook - Architecture and Software, 1989

5

Lenguaje Occam

A Tutorial Introduction to occam Programming

Dick Pountain
and David May



BSP PROFESSIONAL BOOKS
OXFORD LONDON EDINBURGH
BOSTON PALO ALTO MELBOURNE

First published 1987 by
BSP Professional Books
8 John Street
London WC1N 2ES
A division of Blackwell Scientific Publications Ltd
Reprinted with amendments 1988

Printed and bound in Great Britain by Hollen St. Press Slough

British Library
Cataloguing in Publication Data

Pountain, Dick
A tutorial introduction to occam programming.
1. Occam (Computer program language)
I. Title II. May, David
005.13'3 QA76.73.03

ISBN 0-632-01847-X

1 Introduction

The aim of this tutorial is to introduce the reader to concurrent programming using the occam language. It will provide examples of occam programs, and discuss the novel concepts which occam employs. It is not however the definitive guide to the syntax of occam, that you will find in the Formal Definition by David May which forms the second half of this book.

occam is rapidly being recognised as a solution to the problem of programming concurrent systems of all kinds, and as a powerful and expressive calculus for describing concurrent algorithms

occam bears a special relationship with the INMOS Transputer, a high performance single chip computer whose architecture facilitates the construction of parallel processing systems. The Transputer executes occam programs more or less directly (i.e. occam is the "assembly language of the Transputer").

Parallel computer systems can be designed in occam, and then implemented using Transputers as "hardware occam processes". This intimate relation between the software and hardware will be novel to most system designers, who are perhaps used to a more rigid division of labour.

The approach taken in this manual is therefore governed by the realisation that some of its potential readers will not be professional programmers, but rather professional engineers and system designers who wish to use occam to design hardware systems.

For this reason we do not assume extensive knowledge of any other high-level computer language, nor of machine level programming, on the part of the reader. We do however assume a familiarity with the general concepts of computing and computer programming, it is not a manual for the novice to computing.

The tutorial is concerned purely with the language occam and will only briefly address the issues of installing occam programs onto Transputer systems. It is intended as a general introduction to the language, equally suitable for those readers who intend to use occam on conventional computers

We shall not insist that any particular computer compiler combination (or indeed any hardware at all) be available to the reader: hardware dependent aspects of occam are concentrated into a single chapter at the end of the course

For the same reasons there will be no instruction in the detailed workings of particular occam compilers. Error reporting will not be covered except in a general way. Details of this kind are to be found in the manual which accompanies an occam compiler

Acknowledgements Many thanks to the INMOS staff who took time out from writing the compiler and other things to check this text and to improve the examples. In particular my thanks to David May, Ron Laborde and Steven Ericsson-Zenith. Steven has kept this book up to date during the course of the language development which took place during 1986 and 1987

2 Signposts

As an aid to the reader, the author has placed a variety of signposts throughout the text, to signal points of special interest. The meaning of these signposts is as follows.

Take care. Sections so marked are those which explain concepts which are especially likely to trip up novice occam programmers. This may be because

1 This concept is intrinsically difficult.

2 occam handles this area in a different way from traditional languages with which the reader may be familiar

3 This is a limitation or restriction in current implementations of occam.

These sections will repay frequent re-reading, especially if you have written an occam program that doesn't work!

Hint: These are tricks and devices which proved useful to the author while learning occam.

Key Idea: A concept which is fundamental to the understanding of occam. Make sure you thoroughly grasp it.

Aside A brief digression from the main thread of the tutorial into broader computing matters. Experienced programmers might wish to skip them.

Technical Note: A brief explanation of some implementation issue. If you don't understand it, don't let it hold you up but skip it and return later

3 The concepts

Concurrency

Since John Von Neumann discovered the principles over 40 years ago, all digital computers have been designed in a fundamentally similar way

A processor, which can perform a set of basic numeric manipulations, is connected to a memory system which can store numbers. Some of these numbers are the data which the computer is required to process. The other numbers are instructions to the processor and tell it which of its basic manipulations to perform

The instructions are passed to the processor one after the other, and executed. Execution of a computer program is sequential, consisting of a series of primitive actions following one another in time.

Everyday examples of similar activities in the real world could be reading a book (one word at a time), or "executing" a knitting pattern by following the instructions in sequence.

Computers are mainly employed to model the real world. Even the simple act of adding 2 and 2 is a model of the real world, except when it is performed by or for a mathematician who is interested in the pure properties of numbers. Far more frequently 2+2 is a model for the act of adding two pounds, or dollars, or apples, or airplanes, to an existing stock of two.

Certainly the major applications of computers, such as accounting, banking, weather forecasting, process control and even word processing, are explicitly modelling objects, events and activities in the real world.

The world which we inhabit is inherently concurrent. At the scale of human affairs, indeed at any scale between the cosmological and the quantum mechanical, the world behaves as if it were organised into three spatial dimensions and one time dimension

Events happen in both time and space It is possible for two events to occur in the same place one after the other in time (i e 'sequentially') and equally possible for events to occur in different places at the same time (i e concurrently or in parallel)

Concurrency is so much a feature of the universe that we are not normally concerned with it at all. The fact that, for instance, the population of this planet all live different lives in different places at the present time is so obvious that one feels slightly embarrassed in stating it.

However it is worthwhile to reflect on the contrast between the concurrent nature of the world and the sequential nature of the digital computer Since the main purpose of the computer is to model the world, there would seem to be a serious mismatch

In order to model the world with a computer programmers of conventional computers have to find ways to mimic concurrent events using a sequence of instructions This is not a problem in an application like accounting, where it is perfectly reasonable to regard goods despatched, materials and moneys flowing in and out as happening sequentially in time

It is more of a problem when you wish to control a petro-chemicals plant by computer Every process in every part of the plant must be monitored and controlled at the same time, all the time. It is not acceptable for a crisis in one reaction vessel to be overlooked because the computer happened to be looking at a different reactor at the time.

Concurrent programming

The earliest digital computers were programmed using the basic numeric instructions understood by the processor. Such programming is so tedious and error prone that computer scientists soon began to design "high-level" languages, starting with Fortran and leading to the current proliferation which includes Basic, Pascal, Modula 2, C, Ada, Forth, Lisp, Prolog and hundreds of others.

These languages allow programmers to express the logic of a program in notations which use readable

English (or French etc...) words, albeit with a tightly constrained and reduced syntax. A program called a compiler then translates these notations into the basic numeric instructions which the computer understands.

For the majority of languages, the product of the compiler is again a sequence of instructions, to be executed one at a time by the processor just as if they had been produced by hand. In other words these languages faithfully reflect the nature of the underlying sequential Von Neumann computer in a form more palatable to human programmers.

To adequately model the concurrency of the real world, it would be preferable to have many processors all working at the same time on the same program. There are also huge potential performance benefits to be derived from such parallel processing. For regardless of how far electronic engineers can push the speed of an individual processor, ten of them running concurrently will still execute ten times as many instructions in a second.

Conventional programming languages are not well equipped to construct programs for such multiple processors as their very design assumes the sequential execution of instructions.

Some languages have been modified to allow concurrent programs to be written, but the burden of ensuring that concurrent parts of the program are synchronised (i.e. that they cooperate rather than fight) is placed on the programmer. This leads to such programming being perceived as very much more difficult than ordinary sequential programming.

occam is the first language to be based upon the concept of parallel, in addition to sequential, execution, and to provide automatic communication and synchronisation between concurrent processes.

Synchronisation

It's possible to write concurrent programs in conventional programming languages, and to run them on conventional computers. In essence what happens is that the programmer writes a number of programs and the computer pretends to run them all at the same time by running a piece of each one in turn, swapping at very short intervals, until they are all done.

However this kind of programming is more difficult than straightforward "do this, then do this, then do this" sequential programming. Crudely put, this is because a sequential program has only one beginning and one end, but a concurrent program may have many beginnings and many ends.

A sequential program starts, runs and then finishes. It's either running or it's not. Often we are not even concerned about exactly when it finishes (though we usually want it to be as quick as possible on a given computer).

The well worn metaphor of a knitting pattern can be instructive here. A knitting pattern consists of a list of instructions on how to manipulate wool and needles which if followed faithfully lead to the production of, say, a sweater.

Some instructions will have to be repeated many times, and the pattern will use an appropriate notation which tells the knitter to do this without having to write out every single step, just like the repetition structures of computer languages.

For a single knitter who isn't in a hurry, the sweater will take as long as it takes to knit; the sweater is finished when they've performed every instruction in the pattern.
In occam this could be represented by, say,

```
SEQ
  ...knit body
  ...knit sleeve
  ...knit sleeve
  ...knit neck
```

where the SEQ means "do all these in sequence".

Aside The convention of using of three dots ... will be used throughout the rest of this tutorial to describe parts of a program, in ordinary English, whose internal details are not relevant to the example, as with `...knit body`. They should be distinguished from pure comments, introduced by two dashes, as in `--This is a Comment`. Such comments are purely for explanation purposes and do not form part of the program.

But what about a small firm of knitters, who split up the sweaters into components (bodies, necks, sleeves) and share out the jobs? They have orders to fulfill and so time now matters.

The most efficient way to proceed is for everyone to knit their individual bits concurrently. Unfortunately the finishers who put sweaters together can't make a sweater until they have a neck, two sleeves and a body....

From a picture of unconcerned rural bliss by the fireside we switch to one of irascible finishers screaming "hurry up with that sleeve". The point being that finishing the pattern for a sleeve is no longer sufficient indication that the sweater is finished.

The time of finishing now matters very much and, more importantly, finishing one job may depend on the finishing of other jobs outside the individual knitter's control. Unless all the knitters' activities can be suitably synchronised the result is very inefficient production, with everyone waiting on the slowest knitter.

Computers magnify this problem enormously. They are not as intelligent nor as patient as even the most bad tempered of knitters. If several cooperating programs don't finish their parts of the job at the right times, the result is usually that the program won't run at all, rather than it merely running inefficiently.

Computers are infinitely patient in another sense, for a program is perfectly prepared to wait forever for something which will never arrive because the synchronisation is wrong (a situation known to concurrent programmers as *deadlock*).

This being so, concurrent programs can be difficult to write. Achieving the necessary synchronisation between parts has up until now been largely the responsibility of the programmer who has to write in an elaborate system of signals by which each part can tell the others whether or not it is ready.

Each part of the program must continually look at these signals to see whether or not it can carry on. The program code required to achieve this is often considerable, and writing it consumes a lot of time which the programmer could have spent writing those parts of the program which actually do the job (knitting sweaters so to speak).

Given a concurrent program of any complexity it becomes difficult for the programmer to even understand how the parts should relate at all.

occam simplifies the writing of concurrent programs by taking most of the burden of synchronisation away from the programmer. For instance, our concurrent knitters could be described by

```
SEQ
  PAR
    ...knit body
    ...knit left sleeve
    ...knit right sleeve
    ...knit neck
  ...sew sweater
```

This expresses the fact that the parts are knitted in parallel (**PAR**) but that sewing follows sequentially when all the parts are finished.

Communication between the different parts of a program is built into the language itself, and it is synchronised communication - that is, a message will only be sent when both the sender and the receiver are ready.

If one party becomes ready before the other it will automatically wait for the other without any explicit command from the programmer. The only responsibility left with the programmer is that of avoiding deadlock by ensuring that the second party becomes ready sometime (that someone actually is knitting "that sleeve!").

We could add such communications to our knitting description like this:

```
PAR
  SEQ                          -- body
    ...knit body               -- knitter
    ...output body
  SEQ
    ...knit right sleeve       -- sleeve
    ...output right sleeve     -- knitter
    ...knit left sleeve
    ...output left sleeve
  SEQ                          -- neck
    ...knit neck               -- knitter
    ...output neck
  SEQ                          -- finisher
    PAR
      ...input body
      SEQ
        ...input right sleeve
        ...input left sleeve
      ...input neck
    ...sew sweater
```

This is a description of the making of one sweater by four knitters all working at the same time, and all the synchronisation required is implied in its structure. It works by combining simple processes ("knit body") into larger processes (each kind of knitter) which can themselves be combined into a still larger process (make a sweater).

Processes and channels

In occam programming we refer to the parts of a program as processes

Key Idea A process starts, performs a number of actions and then finishes

This definition fits an ordinary sequential program but in occam more than one process may be executing at the same time, and processes can send messages to one another

In conventional programming languages such as BASIC much of the activity of a program consists of changing the values such as numbers or strings of characters stored in variables Take for example this rather unexciting BASIC program

```
10 LET A = 2
20 LET B = A
40 PRINT B
50 END
```

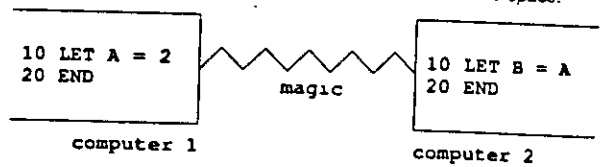
The result of running this program is that the value in both variables **A** and **B** becomes 2, and line 40 causes this value to be printed out on a VDU screen

There is communication of a limited sort going on in this program. The **PRINT** command provides one-way communication between the program and an external device, the VDU screen.

There is also a sense in which the value 2 has been communicated from **A** to **B**, though we wouldn't normally dignify this act with the name "communication" because there is only one BASIC program running and it's being executed one line after another. Instead we tend to regard the value 2 as being stored in both **A** and **B**

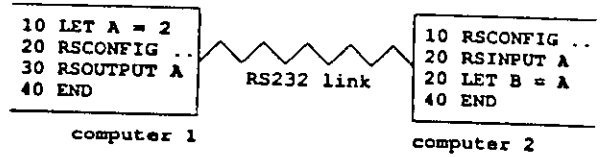
Now imagine that we could have two such programs running at the same time on different computers and

that in some as yet unspecified way they can communicate across space:



The desired result is that the value of **A** somehow crosses the gap between the computers and sets **B** to 2.

It is of course possible to achieve this end with a BASIC program, one could for instance connect the computers together by means of serial communication ports But the BASIC programs would both need to have extra lines added containing special input and output instructions to send or receive data from the serial port, and to match the physical attributes of the ports (e g. bits/second and word length), something like:



occam permits this sort of communication as a normal feature of programming, and doesn't require special instructions which have to be different for each kind of communications device.

More importantly, occam doesn't mind whether the two programs which so communicate are running on different computers, or are just two processes running concurrently on the same computer.

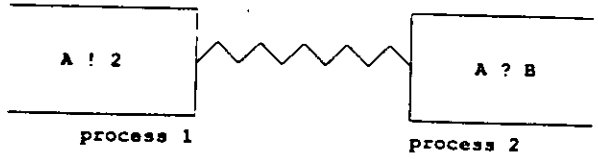
As well as variables for storing values occam uses channels for communicating values Channels look in many ways like variables, except that rather than assigning a value to them for storage (e.g **LET A = 2** in BASIC) we output to them or input from them

The value output by one process is input by another process the channel behaving like a pipe joining the two processes A single channel can only join two processes, it's like a person-to person call rather than a conference Channels are one-way only, so two would be needed for a two way communication

Key Idea A channel is a one-way point-to point link from one process to one other process

A transfer over a channel is actually an act of copying if the value is output from a variable then that variable retains its value and a copy of it is sent over the channel

occam uses the symbol ! to mean output and ? to mean input so we could express the above examples by:



where **A** is a channel and **B** is a variable. This reads as "output 2 to **A**" and "input from **A** to **B**".

Since processes 1 and 2 are independent, they might well be executed at different times. The act of transferring a value from one end of the channel to the other can only happen when both processes are ready.

In other words, if the output in process 1 is executed before the input in process 2 executes, process 1 will automatically wait for process 2 before sending a value Vice versa, if the input in process 2 were executed

before process 1 had output, process 2 would wait for a value to appear. There is no way for a value to be output into "thin air" and lost.

With our hypothetical BASIC programs above there is no such assurance. What would happen should the programs be "out of step" depends on the detailed workings of the particular link we used.

It might well be that if program 1 reached `RSOUTPUT A` before program 2 reached `RSINPUT A`, the value of `A` would be sent and lost. Equally if program 2 arrived first it might stop the program and report an error such as "bad connection".

The two novel features which distinguish channels from variables are:

1) A channel can pass values either between two processes running on the same computer, or between two processes running on different computers. In the first case the channel would in fact be just a location in memory, rather like a variable. In the second case the channel could represent a real hardware link, such as a Transputer link or other serial communication line. Both cases are represented identically in an occam program.

Key Idea. An occam channel describes communication in the abstract, and does not depend upon its physical implementation. You can thus write and test a program using channels without having to worry about exactly where the different processes will be executed. The program can be developed on a single processor workstation; when it's finished and proved you may decide to distribute various processes in the program onto different computers, and do so by making a few simple declarations at the beginning of the program.

2) Channels are patient and polite. If an input process finds that no value is ready it will wait until one is supplied, without any explicit instruction from the programmer. Equally an output will not send until the receiver is ready. This introduces the time factor into programming, but in a way which lifts much of the responsibility for "timekeeping" off the programmers shoulders.

The description of our knitters could now be written using channels to transport the parts:

```
PAR
  SEQ                          -- body knitter
    ...knit body
    bodychan ! body
  SEQ                          -- sleeve knitter
    ...knit right sleeve
    sleevechan ! right.sleeve
    ...knit left sleeve
    sleevechan ! left.sleeve
  SEQ                          -- neck knitter
    ...knit neck
    neckchan ! neck
  SEQ                          -- finisher
    PAR
      bodychan ? body
      SEQ
        sleevechan ? right.sleeve
        sleevechan ? left.sleeve
      neckchan ? neck
    ...sew sweater
```

Three different channels are needed because each may only join two processes, for example bodychan joins the body knitter to the finisher. As we shall see in the next chapter, when occam is used as a computer language, rather than for an informal description as here, channels and variables must be declared before they are used.

Communication over self-synchronising channels is a novel and powerful part of occam, and it can render the writing of concurrent programs a far less formidable task than it is with conventional languages. In the next chapter we shall start in earnest to construct occam programs from the simple processes just outlined.

4 Fundamentals of occam

Primitive processes

All occam programs are built from combinations of three kinds of primitive process. We have seen all three kinds already; they are assignment, input and output.

Assignment process

An assignment process changes the value of a variable, just as it would in most conventional languages. The symbol for assignment in occam is `:=` So the assignment process:

```
fred := 2
```

makes the value in variable `fred` two. The value assigned to a variable could be an expression such as:

```
fred := 2 + 5
```

and this expression could contain other variables:

```
fred := 5 - jim
```

Take care. Be sure not to mix up `=` and `:=` In occam `=` means a test for equality, not an assignment.

Multiple assignment, assignment to more than one variable at the same time, is also possible in occam:

```
fred, john := 2, 3
```

This multiple assignment process makes the value in the variable `fred` two and the value in the variable `john` three. This is really useful for swapping the value of variables:

```
fred, john := john, fred
```

It is important to note however that the rules of occam do not allow a variable to appear more than once on the left side of a multiple assignment. So:

```
fred, fred  = 2, 3   -- ILLEGAL! same variable twice on the left
```

Aside. This is not a particularly useful thing to do anyway but it is important to realise when using variables in the subscript of an array, as we shall see later.

Input process

An input process inputs a value from a channel into a variable. The symbol for input in occam is `?`. The input process:

```
chan3 ? fred
```

takes a value from a channel called `chan3` and puts it into variable `fred`.

Input processes can only input values to variables. It is quite meaningless to input to a constant or to an expression.

An input process cannot proceed until a corresponding output process on the same channel is ready.

Hint: As an aid to memory think of the question mark as meaning "Where's my value ?"

Output process

An output process outputs a value to a channel. The symbol for output in occam is !. The output process:

```
chan3 ! 2
```

outputs the value 2 to a channel called chan3.

The value output to a channel can be anything that you could assign to a variable, so it may be a variable or an expression, and the expression may contain variables.

An output process cannot proceed until a corresponding input process on the same channel is ready.

Hint: As an aid to memory, think of the exclamation mark as meaning "Here's your value !".

Communication

Communication over a channel can only occur when both input and output processes are ready. If during the execution of a program, an input process is reached before its corresponding output process is reached, the input will wait until the output becomes ready. Should the output be reached first, it will wait for its input.

A value communicated over a channel is copied to the input variable and the value of the output variable remains unchanged.

Key Idea: Communication is synchronised.

These then are the building blocks from which occam programs are made. Each such primitive process must occupy a separate line in an occam program, and is the simplest action that occam can perform, an "atom" of occam programming.

Key Idea: occam programs are built by combining primitive processes.

SKIP and STOP

occam has two special processes called SKIP and STOP.

Key Idea: The process SKIP starts, does nothing and then finishes.

SKIP may be thought of as representing a process which does nothing. It might be used in a partly completed program in place of a process which will be written later, but which for the moment can be allowed to do nothing.

For example a process which is to drive an electric motor could be replaced by SKIP when testing the program without a motor. There are also occasions when you want nothing to happen, but the syntax of occam requires a process to be present.

Key Idea: The process STOP starts but never proceeds and never finishes.

STOP may be thought of as representing a process which doesn't work, or is "broken". It might be used, like SKIP, to stand in for a process which has yet to be written.

For example a process to handle errors could be replaced by STOP in the early stages of testing a program.

The effect of a "broken" process tends to spread, because any process which communicates with a broken process will itself never finish, and hence it becomes broken too.

Termination and stopping

So far we have loosely used the term "finish" when referring to processes. Concurrent programming in occam requires us to be rather more precise than this.

A process which completes all its actions is said to terminate. Normally a process starts, proceeds and then terminates.

A process which cannot proceed is said to be stopped which is not at all the same thing. A stopped program never terminates. A process might be stopped by waiting for an event which will never happen, due to a programming error, in which case it is said to be deadlocked.

Correct termination of concurrent programs is not a trivial matter, since they may have many parallel processes which communicate with one another. This topic is of sufficient importance to merit a chapter to itself (see Chapter 9).

Constructions

Several primitive processes can be combined into a larger process by specifying that they should be performed one after the other, or all at the same time. This larger process is called a construction and it begins with an occam keyword which states how the component processes are to be combined.

SEQ construction

The simplest construction to understand is the SEQ (pronounce it "seek"), short for sequence, which merely says "do the following processes one after another". Here is an example:

```
SEQ
  chan3 ? fred
  jim := fred + 1
  chan4 ! jim
```

This says "do in sequence: input from chan3 to fred, assign fred + 1 to jim and output jim to chan4". In sequence means to be more precise that the next process does not start until the previous one has terminated. A SEQ process therefore works just like a program in any conventional programming language: it finishes when its last component process finishes.

Notice the way that the processes which make up this SEQ process are indented by two characters from the word SEQ, so that they line up under the Q. This is not merely to make the program look prettier, but is the way that occam knows which processes are part of the SEQ.

Whenever a construction is built, we indicate the extent of the new process by indenting all its component processes by two characters. Other languages use special characters like { . . . } or begin . . . end for this purpose, but occam uses indentation alone.

Key Idea. A SEQ construction terminates when its last process terminates.

Take care. SEQ is compulsory in occam whenever two or more processes are to run in sequence. In conventional programming languages, sequence is taken for granted and merely writing one statement after another guarantees they will execute in sequence. Because occam offers other modes of execution apart from the sequential, sequence must be explicitly requested.

PAR construction

The PAR construction, short for parallel, says "do the following processes all at the same time", i.e. in parallel. All the component processes of a PAR start to execute simultaneously. For example:

```
PAR
  SEQ
    chan3 ? fred
    fred := fred + 1
  SEQ
    chan4 ? jim
    jim := jim + 1
```

says "at the same time, input from chan3 to fred and then add one to the result, whilst receiving input from chan4 to jim and then adding one to the result".

Notice again the indentation. The first two character indent tells occam that the PAR process consists of two SEQ processes. The second level of indentation shows that each SEQ is composed of two primitive processes.

Notice also that the processes which are to run in parallel are still written in sequence just as in any ordinary program. This is purely a matter of writing convenience. The designers of occam could have chosen to make us write parallel processes side by side, which would give a stronger impression of what is going on:

```
PAR
  SEQ                          SEQ
    chan3 ? fred                 chan4 ? jim
    fred := fred + 1             jim := jim + 1
```

As you will quickly see though, this would become hopelessly clumsy once you had more than two or three parallel processes in a PAR: it would exceed the width of standard VDU screens and printer paper, as well as involving the typist in tedious tabulation.

The important thing to keep in mind is that in a PAR the written order of the component processes is irrelevant as they are all performed at the same time. PAR is not quite so easy to understand as SEQ because the idea of things happening simultaneously in computer programs is new to many programmers.

For instance we can now no longer know for sure which of the two parallel processes in the above example will finish first - it depends upon which input becomes ready first, which in turn depends upon a couple of output processes elsewhere in the program.

The beauty of occam is that this doesn't matter because the PAR construction itself has a single well defined beginning and a single well defined end. We know that the two SEQ processes will start at the same time, run when their inputs become ready and then terminate.

All the component processes in a PAR start at the same time, and the PAR itself terminates when all its component processes have terminated and that is all we need to know.

Key Idea This is the central principle of occam programming, compound processes built up from simpler processes behave just like simple processes i.e. they start, perform actions and then terminate. They can in turn become the components of a still more complex process.

There is a lot more to be said about PAR, especially in relation to communication over channels. Moreover there are several more constructions in occam, which build processes that repeat or make conditional choices.

But before going on to such matters, something needs to be clarified. Up till now we have been using channels and variables like chan3 and fred as if they, so to speak, grew on trees. This is most definitely not the case, in occam both channels and variables need to be specified before they can be used. It makes sense to discuss specifications and types before we go any further, so that the examples we study can be valid occam programs.

Types, specifications and scope

occam, like Pascal and many other languages, but unlike BASIC, requires that every object that is used by a program should have a type which tells occam what sort of object it is dealing with. Furthermore the type of an object must be specified before it can be used in a process.

We have been using named channels (chan3 and chan4) and variables (fred and jim) without any specification so far, a situation which will now be rectified.

Names

First let's deal with names themselves. In occam the names of objects can be as long as you like, and they must start with a letter of the alphabet. The rest of the name, if there is one, can be made up of letters, digits and the dot character. Upper and lower case are distinguished by occam, so that fred and Fred are different names. These are all valid names:

```
x Y  fred  chan3  Chan3  new.fred  old.fred
```

occam keywords such as SEQ, PAR and CHAN are always in upper case and they are reserved. In other words they cannot be used as names that you create.

These are not valid names:

```
3chan     -- doesn't start with a letter
old-fred  -- contains illegal character '-'
fred$     -- contains illegal character '$'
old fred  -- contains a space
CHAN      -- reserved word CHAN
```

Data types

Variables may take on one of several data types, i.e. kinds of value. The following are the types which are always provided by occam:

```
INT     -- an integer or whole number.
BYTE    -- an integer between 0 and 255;
        -- very often used to
        -- represent characters
BOOL    -- one of the logical truth
        -- values TRUE or FALSE.
```

We could specify the variables in the above examples as:

```
INT fred, jim :
```

which means that they can be used to represent positive or negative whole numbers. Several variables may be specified at once, as above, by listing them separated by commas.

Technical Note occam actually provides more data types than those outlined above. Catering for non-integral numbers by supplying various Real Number types. These types also provide fixed length number representation. INT16, INT32, INT64, REAL32, REAL64 are numeric types represented using 16, 32 or 64 bits respectively. The details of these types can be studied in the Formal Definition at the rear of this book. For the purposes of this tutorial we will work only with INT, BYTE and BOOL and will make no assumptions about the physical size of an INT.

Channel type and protocol

Channels are all of the type **CHAN OF** *protocol* It is necessary to specify the data type and structure of the values that they are to carry. This is called the channel protocol. For the present we shall be content to regard channels as able to carry single values of a single data type, rather like variables.

A channel which carries single integer values would be specified by:

```
CHAN OF INT chan3 :
```

where the **INT** specifies the type of values which may pass along the channel **chan3**. The type of **chan3** is **CHAN OF INT**. In general the protocol of a channel is specified by **CHAN OF** *protocol*.

Timer type

The type **TIMER** allows the creation of timers which can be used as clocks by processes. Timers will be discussed further in Chapter 7.

Characters and strings

occam does not have any type **CHAR** or **STRING** to represent alphabetic characters or words Instead characters are represented as numbers of type **BYTE** and strings as arrays of numbers of type **BYTE**. We shall return to this subject in a later chapter.

Boolean type

Boolean values or truth values are produced as the result of tests performed by comparison operators occam provides the following tests:

```
=     -- equal to
<>    -- not equal to
>     -- greater than
<     -- less than
>=    -- greater than or equal to
<=    -- less than or equal to
```

These tests may only be applied to two values of the same type and they always yield a value of type BOOL For example the test 2 <> 3 yields the value **TRUE** since 2 does not equal 3

The truth values **TRUE** and **FALSE** are occam constants which can be used in any situation where a test could be used, you may like to think of them as tests whose outcome is decided in advance

Constants

A name can be given to a constant value by specifying it with:

VAL *type name* **IS** *value*:

So we could write:

```
VAL INT year IS 365:
VAL INT leap.year IS 366:
```


The type can be omitted as occam can deduce it from the value

```
VAL year IS 365:
```

Possible ambiguities over **BYTE** and **INT** are resolved by explicitly specifying the type of the value, which we'll see later on.

Notice the colon, which is used to end all the different kinds of specification This colon joins a specification to the process which follows it.

Scope

In occam, variables, channels and other named objects are local to the process which immediately follows their specification. What this means is that the object to which the name refers effectively does not exist inside any other process. For instance in this example:

```
PAR
  INT fred :
  SEQ
    chan3 ? fred
    ...more processes
  INT jim :
  SEQ
    chan4 ? fred
    ...more processes
```

an error will be reported, because **fred** exists only inside the first SEQ and **jim** exists only inside the second SEQ The second **fred** will therefore look to occam like an unspecified variable

The colon which ends a specification in effect joins the specification to the process which follows it and to reinforce the connection specifications are indented to the same level as the process This following process is the scope throughout which the specification holds

The same name may be used for different objects with different scopes. For instance, we could use **fred** for both variables in the above example

```
PAR
  INT fred :
  SEQ
    chan3 ? fred
    ...more processes
  INT fred :
  SEQ
    chan4 ? fred
    ...more processes
```

the two **fred**s are now different variables, each local to its own SEQ process, and altering the value of **fred** in the first process has no effect on the second.

If inside the scope of a variable (or other named object), another variable is specified with the same name, then within its own scope this namesake replaces the original. The original object is masked by the newcomer.

For example:

```
INT fred :
SEQ
  chan3 ? fred
  INT fred :
  SEQ
    chan4 ? fred
    ...more processes
  ...more processes
```

In this case, the input from **chan4** goes into the second **fred**, and the first **fred** is effectively invisible throughout the second, nested SEQ. Let's now fix up the **PAR** example we saw in an earlier section with some correct declarations:

```
CHAN OF INT chan3, chan4:
PAR
  INT fred:
  SEQ
    chan3 ? fred
    fred := fred + 1
  INT jim:
  SEQ
    chan4 ? jim
    jim := jim + 1
```

Now the channels **chan3** and **chan4** are known throughout the **PAR** process; we could legally refer to either of them in either of the SEQs. On the other hand **fred** and **jim** are known only within their respective SEQs.

Take care: Specifying a variable in occam does not initialise its value to zero. The value of a variable is undefined garbage until it has been assigned to or has input a value. The value of a variable only has meaning during the execution of the process for which it is declared. Since the variable doesn't exist outside this process, it makes no sense to ask what is its value outside the process. But more importantly, it makes no sense either to ask what is its value once the process has terminated. The next time that process is executed, the variable starts out as undefined garbage again. You cannot and must not assume that it keeps the value which it had at the end of the previous execution. For example:

```
WHILE x >= 0  --ILLEGAL! x not declared here
  INT x :
  SEQ
    input ? x
    output ! x

INT x :
WHILE x >= 0  -- unwise: x is garbage here
  SEQ
    input ? x
    output ! x

INT x :
SEQ
  x := 0
  WHILE x >= 0     -- correct
    SEQ
      input ? x
      output ! x
```

(**WHILE** is one way that occam uses to repeatedly execute a process; we'll see it in more detail soon).

Hint: If you need a variable to keep its value from one execution of a process to another, declare it in an outer scope that is, before a process which contains the process which is being repeatedly executed.

Communicating processes

Communication between parallel processes is the essence of occam programming.

At its simplest it requires two processes executing in parallel and a channel joining them:

```
INT x :
CHAN OF INT comm :
PAR
  comm ! 2
  comm ? x
```

This trivial program merely outputs the value 2 from one process and inputs it into the variable **x** in the second. Its overall effect is exactly as if we had a single process which assigned 2 to **x**.

Shared variables : a warning

Communication between the component processes of a **PAR** must only be done using channels. occam doesn't allow us to pass values between parallel processes by using a shared variable.

In fact if a component of a **PAR** contains an assignment or input to a variable, then the variable must not be used at all in any other component:

```
INT x, y :
PAR
  SEQ
    x := 2
    ... more processes
  SEQ
    y := x        -- ILLEGAL!
    ... more processes
```

Keeping variables local to component processes and using channels to communicate values is the right way to do it.

This may seem like a severe restriction to programmers who have experience with conventional languages. It will certainly be the biggest source of errors when first programming in occam.

Like all prohibitions it will be more easily borne if the reason for it is understood. The reason is both simple and necessary.

Parallel processes run at the same time, and in general they run asynchronously, i.e. at their own pace, only coming into synchronisation with each other briefly when forced to by communication over a channel.

If occam allowed one parallel process to read from a variable which has its value altered in another parallel process, what value will be read? It depends upon whether or not the other process has altered it yet, and this can't be known since the processes are asynchronous. And what if the altering process chooses to alter the variable's value at the precise moment that the second process is reading it? What would the value be then?

Such a scheme is obviously unworkable, hence the prohibition. But couldn't we organise it so that a variable warns the other process that it has had its value changed? We could indeed, the resulting object already exists in occam and is called a channel! Q.E.D.

Key Idea: In occam variables are used for storing values, while channels are used for communicating values.

Let's now return to the main track with a more complicated example of a **PAR** which performs some arithmetic on a value before passing it on.

```
CHAN OF INT comm:
PAR
  INT x:
  SEQ
    input ? x
    comm ! 2 * x
  INT y:
  SEQ
    comm ? y
    output ! y + 1
```

Here we have two channels called **input** and **output** which lead to other processes or perhaps to the outside world. We assume that they have been declared elsewhere in a larger program. This piece of program uses two processes working in parallel one of which multiplies an input value by two, the other adds one to the result and sends it on its way to the output. The times-two process and the add-one process communicate on channel **comm**.

Aside: In case it worries you, this is not a particularly useful thing to do; it is purely for illustration. It would be much simpler to do times-two and add-one in a single SEQ process, or indeed in a single expression. But later on when we have more of occam at our disposal, we shall see how this sort of thing can be very useful indeed. At this early stage, all examples of communicating **PAR**s will tend unfortunately to appear trivial.

It's been said several times already that an occam channel is a one-way link between a pair of processes, but it is useful to now examine exactly what this implies. In a communicating PAR construct it means that:

1) Only two component processes of the **PAR** may use any particular channel, one as the sender and the other as receiver.

```
CHAN OF INT comm:
PAR
  SEQ
    comm ! 2
  INT y:
  SEQ
    comm ? y
  INT z:        -- ILLEGAL! two processes
  SEQ           -- inputting from same channel
    comm ? z
```

2) The sender process must only contain outputs to the channel and the receiver must only contain inputs from the channel

```
CHAN OF INT comm:
PAR
  SEQ
    comm ! 2
  INT y:
  SEQ
    comm ? y
    comm ! y+1
    -- ILLEGAL! input and output
    -- from the same channel in
    -- the same process
```


For two-way communication between two processes we would need two channels:

```
CHAN OF INT comm1, comm2:
PAR
  INT x:
  SEQ
    comm1 ! 2
    comm2 ? x
  INT y:
  SEQ
    comm1 ? y
    comm2 ! 3
```

The effect is that each process sends a value to the other: **x** ends up with the value 3 and **y** with the value 2. The order of the inputs and outputs in each SEQ matters very much here and it's important to understand why

If we were to write:

```
CHAN OF INT comm1, comm2:
PAR
  INT x:
  SEQ
    comm2 ? x
    comm1 ! 2
  INT y:
  SEQ
    comm1 ? y
    comm2 ! 3
```

then the program would never terminate: we have the dreaded deadlock

Why deadlock? Because both SEQs wait patiently for an input to become ready. But since each is waiting for the other to output neither can proceed to make the necessary output! It's rather like those comical scenes when two people passing in a narrow doorway repeatedly step to the same side to make way, so repeatedly blocking each other. Swapping the input and output in either process resolves the deadlock.

Take care Sequence your programs to ensure that two parallel processes are never each waiting for a sequentially later output from the other. This is the only circumstance in which occam requires you to worry about such matters, but watch out for it. Like certain stalemates in the game of chess, it may be disguised in complex processes

Repetitive processes

All programming languages provide some means of looping, i.e. performing an action repeatedly. In general it's convenient to distinguish two kinds of repetition: repeat for a specified number of times, or repeat while a given condition holds. occam has both types of repetition. The first, or counted loop we'll see later on. The second conditional loop is performed by a construction called **WHILE**, which includes a test such as **x < 0** or **fred = 100**. The resulting process is executed while the test result is true, or looked at another way, until it becomes false

For example:

```
INT x :
SEQ
  x := 0
  WHILE x >= 0
    SEQ
      input ? x
      output ! x
```

will continue to read values from channel `input` and send them to `output` so long as the value is not less than zero. Every time the inner `SEQ` process terminates, the `WHILE` process will be performed again and the test repeated. This continues so long as the test result is `TRUE` i.e. so long as `x` is greater than or equal to zero. When a negative value is received the `WHILE` process terminates.

Aside The net effect of this process is to buffer (i.e. store) a single value on its way from input to output. occam programs are often designed by making the major processes communicate on a channel, then inserting simple processes like this into the channel to buffer, filter, or transform the transmitted values, almost as if they were electrical components rather than programs.

The logical values `TRUE` and `FALSE` can be used as constants in an occam program, anywhere that a test could be used. So:

```
WHILE TRUE
  INT x :
  SEQ
    input ? x
    output ! x
```

will continue to read values for ever (or until you pull the plug!), whereas:

```
WHILE FALSE
  INT x :
  SEQ
    input ? x
    output ! x
```

is a pointless sort of process which terminates immediately and will read no values at all.

Conditional processes

In addition to repetition, all programming languages need to provide a way for programs to choose to do different things according to a condition i.e. the results of a test. In occam one form of conditional choice is provided by the construction called `IF`.

`IF` can take any number of processes, each of which has a test placed before it, and make them into a single process. Only one of the component processes will actually be executed, and that will be the first one (in the order in which they are written) whose test is true

```
IF
  x = 1
    chan1 ! y
  x = 2
    chan2 ! y
```

In this fragment of program (we assume `x`, `y`, `chan1` and `chan2` are declared elsewhere), the value of `y` will either be output on `chan1` or `chan2` depending upon whether the value of `x` is 1 or 2.

The tests `x = 1` and `x = 2` are boolean expressions which are used to choose which component of the `IF` is to be executed. The component parts of the `IF`, each composed of a boolean expression and a process, are called *choices*.

What if the value of `x` were 3? Then the `IF` process would cause the program to stop just as if `STOP` had been executed. The program can only proceed if one of the choices is executed.

(An `IF` with no choices in it just acts like a `STOP`. A `PAR` or `SEQ` with no component processes on the other hand acts like `SKIP` i.e. the program continues as if it were not there at all).

In many cases it will not be acceptable to have the program stop if `x` is not either 1 or 2. In that case we must add another choice which will be executed no matter what the value of `x`. This is accomplished by using `TRUE`:

```
IF
  x = 1
    chan1 ! y
  x = 2
    chan2 ! y
  TRUE
    chan3 ! y
```

Now `y` will be output on `chan3` if `x` has any other value but 1 or 2, because the test on the last choice is always true so it will always be executed by default if no previous choice has been executed.

If we wanted nothing to happen at all when `x` was not 1 or 2 we could say:

```
IF
  x = 1
    chan1 ! y
  x = 2
    chan2 ! y
  TRUE
    SKIP
```

This provides one example of the utility of `SKIP`; occam requires some sort of process after the guard and won't allow just blank space.

A better way of writing the above is to explicitly state each case as in the following example:

```
IF
  x = 1
    chan1 ! y
  x = 2
    chan2 ! y
  (x <> 1) AND (x <> 2)
    SKIP
```

Aside This is a much better way to write conditional processes as it is totally unambiguous. For convenience and simplicity in this tutorial we will, in places, continue to use `TRUE` as a condition. In real programs you should avoid doing so.

To make more complex choices, `IF`s can be nested by using an `IF` process in a choice of another `IF` construct

```
IF
  x = 1
    chan1 ! y
  x = 2
    IF
      y = 1
        chan2 ! y
      TRUE
        chan3 ! y
  TRUE
    SKIP
```

In this process, `chan3` is used for the output if `x` is 2 and `y` has any other value than 1.

Selection processes

Like many other programming languages occam provides a further means of making a choice depending upon the value of a variable. Such a construction is called a *selection* in occam and provides an efficient means of selecting one of a number of options in a CASE.

CASE can take any number of processes, each of which has a list of one or more expressions placed before it, and combines them into a single process. Only one of the component processes will actually be executed, and that will be the first one (again, in the order in which they are written) with an expression which has the same value as the selecting variable:

```
CASE x
  1
    chan1 ! y
  2
    chan2 ! y
```

In this fragment of program (which is similar to the first example used to describe IF), the value of y will either be output on chan1 or chan2 depending upon whether the value of x is 1 or 2. Typically, the constants used in a CASE will be named, and also there can be more than one case expression:

```
CASE x
  i, j
    chan1 ! y
  k
    chan2 ! y
```

In this program fragment each constant expression has been given a name: i, j and k. The value of y will be output on chan1 if x has the same value as i or j, and will be output on chan2 if x has the same value as k.

What if the value of x was none of these values? Then the process would cause the program to stop just as if STOP had been executed. The program can only proceed if one of the options is executed, just as an IF may only proceed if a choice is executed.

Once again (as we saw with IF), in many cases it will not be acceptable to have the program stop if x does not have the same value as one of the case expressions. We must then add a further option which will execute no matter what the value of x. This is accomplished by using ELSE.

```
CASE x
  i, j
    chan1 ! y
  k
    chan2 ! y
  ELSE
    chan3 ! y
```

Now y will be output on chan3 if x has any value other than i, j or k.

The component parts of the CASE, each composed of an expression and a process, are called *options*.

Alternative processes

In occam, choice has an extra dimension lacking in ordinary programming languages. We have just seen how to make choices according to the values of conditional expressions in a program using IF, and how to select an option according to the value of a variable. However we can also make choices according to the state of channels. This is made possible by the ALT construction, whose name is short for alternation.

Like IF, ALT joins together any number of components into a single construction, but the component parts of an ALT, called *alternatives* are rather more complicated than IF choices.

The simplest kind of ALT has as each alternative an input process followed by a process to be executed. The ALT watches all the input processes and executes the process associated with the first input to become ready. Thus ALT is basically a first-past-the-post race between a group of channels, with only the winner's process being executed:

```
CHAN OF INT chan1, chan2, chan3 :
INT x:
ALT
  chan1 ? x
    ...first process
  chan2 ? x
    ...second process
  chan3 ? x
    ...third process
```

If chan2 were the first to produce an input, then only the second process would be executed.

Here choice is being decided in the time dimension, the inputs causing the program to wait until one of them is ready.

An alternative may start with a test in addition to an input, just like the tests in an IF. If this is done, the associated process can only be chosen if its input is the first to be ready and the test is TRUE. occam makes this easy to remember by using the & sign, as in:

```
CHAN OF INT chan1, chan2, chan3 :
INT x:
ALT
  (y < 0) & chan1 ? x
    ...first process
  (y = 0) & chan2 ? x
    ...second process
  (y > 0) & chan3 ? x
    ...third process
```

If y is, say 3 and chan3 is the first to be ready then the third process will be executed. This form of alternative is most often used to impose limits on some process, by using a test such as (voltage < maximum).

As with IF, ALT behaves like STOP if there are no alternatives. Also like IF, an ALT can be nested as inside an outer ALT.

The ALT is an extremely powerful construction. It allows complex networks of channels to be merged and switched in a simple and elegant way.

Because of this power, and because it is unlike anything in conventional programming languages, ALT is far-and-away the most difficult of the occam constructions to explain and to understand. Fortunately we have now seen enough of occam to be able to work through some more serious examples, which should clarify its usage.

A simple controller program

Let's suppose that we are designing a program to control a portable music centre. Like so many modern appliances it has digital controls rather than rotating knobs.

To control the sound volume, there are two buttons, marked louder and softer. Pressing louder increases the volume one notch, and likewise pressing softer reduces it.

We have two occam channels, also called louder and softer which produce an input whenever a button is pressed, and a third channel called amplifier which transmits a value to the amplifier section where a control chip sets the volume to that value.

The processes which increase or decrease the volume value are easily written (we'll leave declarations until we have a complete program):

```
SEQ
  volume := volume + 1
  amplifier ! volume
```

and

```
SEQ
  volume := volume - 1
  amplifier ! volume
```

Now the program needs to decide which button was pressed most recently, and hence which action to take. Combining the two processes in an **ALT** will achieve this:

```
ALT
  louder ? any
    ...increase volume
  softer ? any
    ...decrease volume
```

The actual value sent by the button press is not important, so we'll declare a variable called **any** just to dispose of the input value. As things stand this process will only operate once, on the first button press, and then terminate. It needs to be continually repeated to scan the buttons.

The full program would look like this:

```
INT volume, any :
SEQ
  volume := 0
  amplifier ! volume
  WHILE TRUE
    ALT
      louder ? any
        SEQ
          volume := volume + 1
          amplifier ! volume
      softer ? any
        SEQ
          volume := volume - 1
          amplifier ! volume
```

Notice that **volume** is initially set to 0 so that the volume starts off low rather than at just any random value.

This program does the job, but using **WHILE TRUE** means that it can never end, when the music centre is switched off the program will just "die" wherever it happens to be at the time. Good programmers don't like that sort of messy ending so let's add another channel which reads the OFF button (call it **off**) and a

variable called **active**, which is **TRUE** so long as the music centre is switched on:

```
BOOL active:
INT volume, any :
SEQ
  active := TRUE
  volume := 0
  amplifier ! volume
  WHILE active
    ALT
      louder ? any
        SEQ
          volume := volume + 1
          amplifier ! volume
      softer ? any
        SEQ
          volume := volume - 1
          amplifier ! volume
      off ? any
        active := FALSE
```

This now terminates tidily when the OFF button is pressed.

Aside: For convenience and simplicity in this tutorial we will continue to use **WHILE TRUE** from time to time. In real programs you should avoid doing so, taking good care to ensure correct termination.

As a further refinement we can add tests to the volume increase and decrease processes to limit the values to the range which the control chip can accept (let's say the range is from 0 to 100 units).

We could merely add the tests **volume < 100** and **volume > 0**. As a matter of good programming style though it would be better to define the limits as named constants at the beginning of the program: if the values ever have to be changed (say a new control chip is introduced) then you will only have to change them in one place rather than searching the whole program to find out everywhere they have been used.

The final program looks like this:

```
VAL maximum IS 100 :
VAL minimum IS 0 :
BOOL active :
INT volume, any :
SEQ
  active := TRUE
  volume := minimum
  amplifier ! volume
  WHILE active
    ALT
      (volume < maximum) & louder ? any
        SEQ
          volume := volume + 1
          amplifier ! volume
      (volume > minimum) & softer ? any
        SEQ
          volume := volume - 1
          amplifier ! volume
      off ? any
        active := FALSE
```

The use of named constants can also make a program more readable than if it were strewn with unexplained numbers.

Two points arise from this program:

1 Notice that we have not declared the channels `louder`, `softer`, `off` and `amplifier`. That's because they connect to the hardware rather than to other occam processes, ultimately they represent physical bits of wire connected to a button. Later on we'll see how to connect channels to hardware in occam.

2 In a real control program there might be many other things for the program to do besides reading the volume buttons, for example auto-search for selected tunes on a tape. Processes to do these tasks could be combined with the above program using a `PAR` in place of the main `SEQ` so they all proceed at the same time.

Arithmetic in occam

So far we have not discussed what arithmetic operations are available in occam, though we have taken for granted that it has addition, subtraction and multiplication.

The basic arithmetic operations are these:

```
x  +  y  -- add y to x
x  -  y  -- subtract y from x
x  *  y  -- multiply x by y
x  /  y  -- quotient when x is divided by y
x REM y  -- remainder when x is divided by y
```

These operations can be performed on numbers of type `INT` or `REAL`. (see the Formal Definition for precise details of how remainders, overflow and other such matters are treated)

All operators have the same priority in occam so parentheses must be used in complex expressions to enclose component operations and allow them to be treated as single operands. This also establishes the order of evaluation. For example

```
(2+3)*(4+5)       -- answer 45
 2+(3*(4+5))      -- answer 29
(2+(3*4))+5       -- answer 19
 2+3*4+5          -- illegal
```

The tests are also operators in this sense and so parentheses will be needed to avoid incorrect interpretation. For example

```
fred = (2+jim)      -- legal expression
(fred+jane) > jim   -- legal expression
fred+jane > jim     -- illegal expression
fred+(jane > jim)   -- illegal expression
                    -- (mixed types)
```

For integers only there is a further set of modulo arithmetic operators. Modulo arithmetic, for those who have not encountered it before, deals with number systems where there is a limited range of numbers. Ordinarily we prefer to think of numbers as going on forever; so that there is always one bigger than any number you can think of.

As an example, (unsigned) arithmetic modulo 8 only allows the numbers 0 to 7 to be used - the result of adding 1 to 7 is zero again. So (5 + 7) modulo 8 is 4. An everyday example of modulo arithmetic is the arithmetic of clock times (modulo 12 or 24 with no zero); adding 3 hours to 11 o'clock gives 2 o'clock, not 14 o'clock.

Modulo arithmetic is important in computers because they always work with numbers that are limited by the size of memory used to store them. For example the largest signed number that can be represented by a 16 bit INT is 32,767.

occam has the operators `PLUS`, `MINUS` and `TIMES` for addition, subtraction and multiplication modulo 2^^(number of bits in an `INT`).

For boolean truth values, occam has the operators `AND`, `OR` and `NOT` which are defined by:

```
NOT FALSE   = TRUE        NOT TRUE    = FALSE
FALSE AND x = FALSE       TRUE AND x = x
FALSE OR  x = x           TRUE OR  x = TRUE
```

where `x` is any boolean value i.e. either `TRUE` or `FALSE`.

Type conversions

Sometimes it's convenient to convert one type to another in a program; it may for instance save having to declare several extra variables for a value that is only required once. Type conversion should be used sparingly as the whole point of types is to prevent values being used in inappropriate situations.

If `number` has been declared as `INT` and `digit` as `BYTE`, we could still add them together like this:

```
number := ( number * 10 ) + ( INT digit )
```

The reverse conversion, of `INT` to `BYTE` is only legal if the value is within the `BYTE` range of 0 to 255. For example, to output a number between 0 and 9 as a character we could write

```
output ! BYTE ( number + (INT '0') )
```

Values of type `BOOL` can be converted to type `INT` or `BYTE` and vice versa, using the following definitions:

```
INT TRUE or BYTE TRUE     is 1
INT FALSE or BYTE FALSE   is 0
BOOL 1                    is TRUE
BOOL 0                    is FALSE
```

so if the value of `active` is `FALSE`, `INT active` is 0

Bit operators

To allow low level operations on the individual bits in a value, occam provides bitwise operators ~ (bitwise not), /\ (bitwise and), \/ (bitwise or) and >< (bitwise exclusive-or) plus the left and right shift operators << and >>. These operators work only on integer values.

Aside. There isn't space here to explain the effect of these operators, which requires knowledge of binary arithmetic. If you are not already familiar with binary arithmetic, any good introductory computing book (e.g. A. Osborne's "An Introduction to Microcomputers - Vol 0") will explain it. But if you don't understand them already, you probably don't need them.

Numeric constants can be entered in hexadecimal notation by preceding them with the # sign:

```
#FE  ( equivalent to 254 decimal)
```

This covers occam arithmetic in more than enough depth to follow all the examples in this tutorial. Readers who are greatly concerned with numerical calculations should study the full details presented in the Formal Definition.

Abbreviations

The notation we saw earlier for naming constants (e g. `VAL maximum IS 1000`) is nothing but a particular form of a more powerful and general device called an abbreviation.

Abbreviations can be used to give a name to any expression in occam, providing a form of universal shorthand. For example:

```
VAL exp IS ((x + y) / (fred * 128)) :
```

defines `exp` to be an abbreviation for the value of the complex expression on the right. The fact that this specification ends with a colon tells us that abbreviations are local, like other occam objects, and their scope is the following process.

An expression abbreviation may, as above, contain variables on its right-hand side, but these variables must remain constant throughout the scope of the abbreviation, and the compiler will report an error if any of the variables are changed (by assignment or input). As a result, an expression abbreviation behaves like a constant throughout its scope.

The full form of an expression abbreviation includes a type before the name:

```
VAL INT exp IS ((x + y) / (fred * 128)) :
```

but this can always be omitted as occam can deduce the type from the types of the values on the right hand side. Programmers may nevertheless wish to sometimes include a type specifier as a reminder of the type of a complicated expression.

occam assumes that numbers below 256 are `INT`s unless told otherwise, which can be done.

```
VAL Esc IS 27 (BYTE) :
```

We shall see later on in Chapter 5 that abbreviations can also be used to name arrays and parts of arrays

Procedures

A procedure is a process with a name, and this name can be used to represent the procedure in other processes. To define a procedure, the keyword `PROC` and a name is followed by a process which is called the procedure body

The body of a procedure is executed whenever its name is found in a program, such an occurrence of the name is called an *instance* of the procedure

A procedure definition looks like this.

```
PROC delay ()                -- procedure heading
  VAL interval IS 1000 :     --
  INT n :                    --
  SEQ                        -- procedure body
    n := interval            --
    WHILE n > 0              --
      n := n - 1             --
:
INT y:                       -- main process
SEQ                          -- starts here
  input1 ? y
  delay ()                   -- instance
  output1 ! y
  delay ()                   -- instance
  input2 ? y
  delay ()                   -- instance
  output2 ! y
```

All this procedure does is to count downwards from 1000 to 0, so it could be used as a crude way of introducing a time delay into a program (the proper way to introduce a delay in occam, using timers, will be introduced later in Chapter 9) The empty parentheses after `delay ()` show that this procedure takes no parameters; we'll discuss parameters a little further on.

Note that like all specifications this is attached by a colon to the following process. This tells us that the procedure name obeys the same scope rules as variable and other names - the procedure is only known throughout the process which immediately follows, to which it is linked by the colon. occam puts the colon which ends a procedure definition on a new line, as above to mark clearly the end of the body The colon must appear directly below the "P" in `PROC`.

Execution of this program begins at the main process: it reads values from two input channels (assumed to be declared elsewhere), and is delayed for a while before sending them to the output channels Whenever an instance of delay is encountered, it is executed exactly as if the body of the procedure had been substituted for the name like this

```
INT y :
SEQ
  input1 ? y
  VAL interval IS 1000 :
  INT n :
  SEQ
    n := interval
    WHILE n > 0
      n := n - 1
  output1 ! y
  VAL interval IS 1000 :
  INT n :
  SEQ
    n := interval
    WHILE n > 0
      n := n - 1
  input2 ? y
  VAL interval IS 1000 :
  INT n:
  SEQ
    n := interval
    WHILE n > 0
      n := n - 1
  output2 ! y
```

PROC thus provides us with a shorthand; the name **delay** is not only much shorter than the code it replaces, but here it replaces this code three times over. So the main process becomes much more compact, and more readable too since the name **delay** gives us an idea of what it does.

Technical Note. A procedure can always be compiled either by substituting its body as above, or as a closed subroutine.

This is by no means the only benefit bestowed by procedures though. They provide a way of designing better structured programs. By breaking down a program design into the smallest parts which still make sense (called "factorising" the problem), and then writing these parts as procedures, the logic of the whole program is made clearer and easier to follow.

Often such procedures will be used more than once, hence reducing the size of the program, and some may be sufficiently general-purpose to be used again in other programs.

Sensibly factorised programs are easier to modify, debug and maintain because by modifying a single procedure declaration, the changes are automatically effected everywhere in the program that procedure is used.

Hint. There isn't space in this tutorial to cover the subject of structured programming and "top-down" design techniques in proper detail. Readers who are not familiar with these techniques are referred to the numerous books on the subject, of which a recommended example is "Structured Programming" by Dahl, Dijkstra and Hoare.

Parameters

Procedures can be made more useful still by introducing parameters, which allow different values to be passed to different instances of a procedure.

In the delay example above the length of delay is fixed as `1000` in the text of its definition (by the abbreviation `VAL interval IS 1000`) and it can only be changed by editing the program. A more flexible way would be to pass the delay length as a parameter:

```
PROC delay (VAL INT interval)
  INT n :
  SEQ
    n := interval
    WHILE n > 0
      n := n - 1
:
INT y :                          -- main process
SEQ                              -- starts here
  input1 ? y
  delay (1000)
  output1 ! y
  delay (2000)
  input2 ? y
  delay (500)
  output2 ! y
```

The name **interval** in the definition of **delay** is called a formal parameter. Formal parameters may be of any type, including **CHAN**, and a type specification is compulsory in the procedure heading. occam cannot read your mind, so it cannot work out what type a formal parameter is meant to be if you don't tell it. (In an abbreviation specification on the other hand, occam has an example value to look at, and so will always be able to deduce its type).

A formal parameter behaves like an abbreviation attached to the procedure body (in fact here it has replaced the abbreviation `VAL interval IS 1000`).

When the procedure body is substituted for an instance of the procedure name in a process, this formal parameter name becomes an abbreviation for a value called the actual parameter. In the above example, the actual parameters are `1000`, `2000` and `500`. In the first case **interval** becomes an abbreviation for `1000` throughout the procedure body so **delay** `(1000)` has exactly the same effect as our original non-parameterised procedure.

A procedure can have any number of formal parameters, which must be separated by commas in the definition heading. The actual parameters of an instance are similarly separated by commas. One actual parameter must be supplied for each formal parameter, and they correspond by position (the first actual parameter matches the first formal etc.):

```
PROC box.volume (VAL INT length, breadth, depth)
  ... body                       -- definition
:
box.volume (24, 16, 20)          -- instance
```

Procedures with parameters provide a still more powerful shorthand, for now we can use the same procedure in different places with different internal values. In the above program for instance, the second delay will last twice as long as the first and the third will last half as long.

We are not limited to calling **delay** with constants like `1000` as the actual parameter. Any variable of type **INT** could be used as an actual parameter e.g. **delay** `(x)`.

Occam parameter passing convention

In occam, when a variable is passed as an actual parameter to a procedure, it is as if the variable replaces the formal parameter throughout the procedure. Anything that is done to the formal parameter, is done to the variable, which may therefore have its value changed.

Take this example:

```
PROC decrement (INT number)
  number := number - 1
:
```

If we use **decrement** `(x)` as an instance of the procedure, the value of **x** will be reduced by one when **decrement** `(x)` terminates.

Take care. This behaviour differs from that found in certain other widely used languages. The commonly used call by-value convention (available in C and Pascal) has the effect of evaluating the variable (actual parameter) and using the result as the initial value of the formal parameter, which behaves as a local variable of the procedure body. Consequently, an assignment to the formal parameter has no effect on the actual parameter. The occam convention is more nearly equivalent to Pascal's call-by-reference (or VAR) parameters. This point is emphasised because it may trip up programmers who are experienced in these other languages.

Sometimes it is preferable that a procedure should not alter the value of a variable passed to it as a parameter. We have already seen how to achieve this in our delay example: decant the value of the parameter into a local variable (n in the example) and do any manipulations on this local value. Use this method if you need to translate Pascal procedures or similar with value parameters, into occam.

If only the original value of a variable is needed in a procedure, i.e. if the formal parameter is never altered by assignment or input, then a more efficient program may result if we explicitly say that only the value is to

be passed, using VAL:

```
PROC delay (VAL INT limit)
  INT n :
  SEQ
    n := 0
    WHILE n < limit
      n := n + 1
:
```

When VAL is used like this, the formal parameter can be thought of as representing a constant throughout the procedure body, and the compiler (after checking that it's true) may exploit the fact to produce more efficient code.

The volume controller program we developed in the last section could be rewritten using a procedure:

```
VAL step.up IS 1 :
VAL step.down IS -1 :
BOOL active :
INT volume, any :
PROC change.volume (VAL INT step)
  SEQ
    volume := volume + step
    amplifier ! volume
:
SEQ
  volume := 0
  amplifier ! volume
  active := TRUE
  WHILE active
    ALT
      louder ? any
        change.volume (step.up)
      softer ? any
        change.volume (step.down)
      off ? any
        active := FALSE
```

Notice that a single procedure now serves both to increase and decrease the volume

Another point to note is that this PROC body uses the variable volume even though that variable is not declared in the PROC, either as a local variable or a formal parameter. This is quite acceptable to occam. volume is called a *free* variable with respect to PROC change.volume and has the useful property of being able to retain its value from one call of the procedure to the next, which is precisely why it is used here Note that volume must be declared somewhere before the PROC definition, otherwise occam would reject it as an undeclared name.

A variable is free with respect to a procedure when the procedure is defined inside the scope of the variable

Functions

Most conventional languages support functions. Briefly, a function is a process which returns a value and thus may be used in expressions. Many functions are mathematical, and return such things as the sine and cosine of their arguement. There is a big difference between the type of functions mathematicans know and love, and functions as known by most programmers (which are nowhere near as trustworthy). occam provides the more trustworthy kind of function.

In occam a function gives a name to a special kind of process which returns a result called a *value process*. occam functions have the advantage of being side effect free. Practically, this means that occam is very strict about how you construct functions. The great advantage of this however, is that when you use a

function you can guarantee it will have no effect upon any other part of your program Many of the bugs which mysteriously appear in programs written in other languages are due to the fact that you cannot make the same guarantees.

Function definitions take the general form:

```
type FUNCTION name ( { , formal parameter } )
  specification :
  VALOF
    process
    RESULT expression
```

And, for example, a function which returns the value of the largest of two integers would look like this:

```
INT FUNCTION max (VAL INT a, b)
  INT answer:
  VALOF
    IF
      a > b
        answer := a
      b > a
        answer := b
      a = b
        answer := a  -- could in fact be either value
    RESULT answer
:
```

Notice that a type specifier precedes the keyword FUNCTION This is important as it specifies the type of the value returned by the function

The formal parameters of a function can only be value (VAL) parameters PARallel and ALTernation constructs cannot be used within the function Also input and output must not be used within the function You can only assign to variables declared within the scope of the function, "free variables" can be read but not assigned to Only procedures defined within the scope of the function and adhering to the above rules may be used

A function returns a value and is defined as an operand so functions can appear where-ever an expression would appear Using our function to return the maximum of two values in an assignment for example:

```
x := max (a, b)
```

Or in an expression to gain twice the maximum value

```
x := max (a, b) * 2
```

Functions share many of the advantages of procedures and extend the facility to factorise programs written in occam.

Hint There isn't space in this tutorial to go into functions in any great depth or to cover the many issues involved. A full description of functions in occam can be found in the occam *2 Reference manual* published by INMOS.

6

Programación de Procesos Paralelos

3 Getting started

This chapter contains a tutorial that shows you how to compile, link, and run a simple example program on a single processor.

A more complex programming example, illustrating separate compilation, can be found in chapter 4, together with a detailed description of program development for single transputers. While chapter 5 provides a description and examples of multitransputer programming.

The tutorial, given in this chapter, assumes that you have a boot from link board containing a IMS T400, T414 or T425 processor. If you have a board fitted with any other transputer you must compile and link the program for that transputer type, see section 3.3.6. The tutorial also assumes that certain environment variables have been set up. These are introduced in sections 2.10.3 and 2.10.4 and a description of how to set them up is given in the delivery manual supplied with this product.

If you do not have a transputer board use the T425 simulator tool `isim` to run the application program, see section 3.3.5.

3.1 Example command line

Where necessary, the example command lines are duplicated for different host versions of the toolset: the '-' switch character is used in command lines for UNIX based toolsets and the '/' character is used in commands for MS-DOS and VMS based toolsets. When reproducing the examples you should use the appropriate command line for your host system.

3.2 Interrupting programs

To interrupt an occam program while it is still running, press the host system BREAK key to interrupt the server. See the delivery manual, section 'Server Interrupts' for further details.

When the BREAK key is pressed the following prompt is displayed:

```
(x)exit, (s)hell, or (c)ontinue?
```

To abort the program type '**x**' or press RETURN. This terminates the host file server.

To suspend the program so that you can resume it later, type '**s**'.

To abort the interrupt and continue running the program, type 'c'.

3.3 Compiling and running a simple example program

The example program `simple.occ` reads a name from the keyboard and displays a greeting on the screen. The source of the program can be found in the toolset `examples` directory. The program uses the library `hostio.lib` and incorporates the include file `hostio.inc`.

The program is illustrated below.

```
#INCLUDE "hostio.inc" -- contains SP protocol

PROC simple (CHAN OF SP fs, ts, []INT memory)

  #USE "hostio.lib"  -- iserver libraries

  []BYTE buffer  RETYPES memory:

  BYTE result:
  INT length:
  SEQ
    so.write.string    (fs, ts,
                        "Please type your name :")
    so.read.echo.line  (fs, ts, length, buffer,
                        result)
    so.write.nl        (fs, ts)
    so.write.string    (fs, ts, "Hello ")
    so.write.string.nl (fs, ts,
                        [buffer FROM 0 FOR length])
    so.exit            (fs, ts, sps.success)
```

The first line in the program loads the file `hostio.inc`. This file contains the definition of protocol SP, used to communicate with the host file server, and a number of constants that are used in conjunction with the host i/o library.

The procedure `simple` is then declared. All the working code is contained within this procedure. Single processor programs must always use a similar parameter list.

The server library `hostio.lib` is referenced by the `#USE` directive. This library contains all the procedures used by the program. See part 2, section 1.4 for descriptions of the routines.

Before the body of the procedure a number of variables are declared. First, the `memory` array is retyped as a `BYTE` array. This enables the program to use the free memory on the board as a character buffer.

The variables `length` and `result` are then declared for use by the program. The variable `length` refers to the number of characters in the name read from the keyboard, and `result` is used by the library routine to indicate whether or not the read was successful. The result is ignored by this example for the sake of simplicity; it is assumed that screen writes and keyboard reads always succeed.

The working code is contained within a `SEQ`, indicating that the statements which follow are to be executed sequentially. All of the statements are calls to library routines in `hostio.lib`. The code prompts for a name, reads the name from the keyboard, and types a greeting on the screen.

The last statement calls a library procedure which terminates the server, returning control to the host operating system. Without this statement the program would finish and appear to hang, and the server would have to be terminated explicitly by interrupting the program.

3.3.1 Setting environment variables

Certain environment variables must be set up prior to using the toolset. These are introduced in sections 2.10.3 and 2.10.4 and a description of how to set them up is given in the delivery manual supplied with this product. For example, the compilation will fail with a message indicating that `hostio.inc` has not been found, should the environment variable `ISEARCH` not be set up correctly.

3.3.2 Compiling the example program

In order to compile the program in its simplest form (ie with all defaults enabled) the following command line should be used:

```
oc simple
```

Because the file has the default extension of `.occ` you can omit it when invoking the compiler.

The compiler will create a file called `simple.tco`, containing the code compiled for a T414 in HALT mode. The compiler will perform the necessary syntax, alias and usage checks and will insert code to perform run-time error checking. By default the compiler enables interactive debugging with `idebug`.

3.3.3 Linking the example program

To use the result of your compilation it must be linked with the libraries that it uses.

To link the program type:

```
ilink simple.tco hostio.lib -f occama.lnk          (UNIX)
ilink simple.tco hostio.lib /f occama.lnk          (MS-DOS/VMS)
```

The linked program will be written to the file `simple.lku`. As no output file is specified, the file is named after the input file and the default link extension `.lku` is added.

The library `hostio.lib` is the server library used by this program.

The '`f`' option specifies a linker indirect file containing commands and directives to `ilink`. Three indirect files are supplied to support different transputer types. They are `occam2.lnk`, `occama.lnk` and `occam8.lnk`; they are described in chapter 19. These files identify various libraries including compiler libraries which are required to be linked with the program. These files are provided as a short-hand method of specifying such libraries to the linker.

The file `occama.lnk` is the correct file to use for T4 series transputers.

Note: In more complex programs, libraries may be dependent on other files and libraries. To ensure all necessary libraries are linked into a program, the `imakef` tool may be used with a suitable MAKE program. (See below).

3.3.4 Creating a bootable file

Before the program can be run it must be made 'bootable'. This involves adding bootstrap information to make the program loadable and is achieved using the collector tool `icollect`. One of the following commands should be used depending on the type of host in use.

```
icollect simple.lku -t                             (UNIX)
icollect simple.lku /t                             (MS-DOS/VMS)
```

By default `icollect` expects the input file to have been produced by the configurer. Because the example program is going to run on a single processor there is no need to configure it. The '`t`' option instructs the collector to build a bootable file from a linked unit. The bootable program will be written to the file `simple.btl`.

`icollect` will also create a configuration binary file as a by-product of creating the bootstrap. Configuration binary files describe the network configuration, in this case a single transputer. This file will have the extension `.cfb` and is created by `icollect` for use by the debugger. For multitransputer programs the configurer is used to create configuration binary files.

Chapter 12 gives more information on the collector tool.

3.3.5 Running the example program

To run the program it must be loaded onto a transputer board using the host file server tool `iserver`. To load and run the program use one of the following commands:

```
iserver -se -sb simple.btl                         (UNIX)
iserver /se /sb simple.btl                         (MS-DOS/VMS)
```

The '`sb`' option specifies the file to be booted and loads the program onto the transputer board. It has the effect of resetting the board, opening communication with the host, and loading the program onto the network. The '`se`' option directs the server to terminate if the program sets the error flag. For more details about the server options see chapter 22.

Figure 3.1 shows an example of the screen display, obtained by running `simple.btl` on a UNIX based toolset, for a user called 'John'.

```
iserver -se -sb simple.btl

Please type your name :John
Hello John
```

Figure 3.1 Example output produced by running `simple.btl`.

If you are using the simulator to run the example program use one of the following commands:

```
isim -bq simple.btl                                UNIX
isim /bq simple.btl                                MS-DOS/VMS
```

The '`bq`' option specifies batch quiet mode which causes the simulator to run the program and then terminate. For more details about how to use the simulator see chapter 23.

3.3.6 Compiling and linking for other transputer types

If you are using a transputer other than a T400, T414 or T425 you must specify a target transputer type for the compilation and linkage function, since the default type T414 will be inappropriate. Chapters 25 and 19 describe the options available. The same option must be specified to both the compiler and the linker, otherwise the linker will report an error. In addition, you must change the linker indirect file as described in chapter 19.

For example to compile and link the program '`simple.occ`' so that it will run on a T800, T801 or T805 use the following command lines, as appropriate:

UNIX hosts:

```
oc simple -t800
ilink simple.tco hostio.lib -f occam8.lnk -t800
```

MS-DOS/VMS hosts:

```
oc simple /t800
ilink simple.tco hostio.lib /f occam8.lnk /t800
```

3.4 Using `imakef`

As an alternative method of program development the toolset Makefile generator `imakef` can be used. This tool can produce a Makefile for any type of file that can be built with the toolset tools. `imakef` serves two purposes:

- It enables the user to generate a target file automatically (e.g. a bootable file) without having to manually perform the intermediate stages of program development i.e. compiling, linking, configuring etc.
- For more complex programs, comprising several modules, it simplifies the incorporation of changes to the program by identifying dependencies and incorporating them into the Makefile.

In order for `imakef` to be able to identify file types, a different system of file extensions must be used to that used in the examples above. See chapter 21 for a description of `imakef` and the extensions used.

To create a Makefile for the example program, use the following command:

```
imakef simple.b4h
```


The `.b4h` extension informs `imakef` that we wish to build a bootable program for a T414 in the default HALT error mode. `imakef` will create a Makefile called `simple.mak` containing full instructions on how to build the program.

To build the program run the MAKE program on `simple.mak`. The entire program will be automatically compiled, linked and made bootable, ready for loading onto the transputer.

For example:

```
make -f simple.mak          UNIX
make /f simple.mak          MS-DOS/VMS
```

To run the program:

```
iserver -se -sb simple.b4h      (UNIX)
iserver /se /sb simple.b4h      (MS-DOS/VMS)
```

If you are using the simulator to run the example program use one of the following commands:

```
isim -bq simple.b4h         UNIX
isim /bq simple.b4h         MS-DOS/VMS
```

4 Programming single transputers

This chapter provides an introduction to occam programming using the toolset, using an example program for single processors. The chapter follows on from the information and example given in chapter 3 'Getting started'. For information on programming multitransputer networks see chapter 5.

Before reading this chapter the user should already be familiar with the concepts and syntax of the occam programming language. For detailed information about the language see the '*occam 2 Reference Manual*' and for an introduction to occam see '*A tutorial introduction to occam programming*'.

4.1 Program examples

A simple programming example, to get you started, is provided in section 3.3.

This chapter uses a more complex example, illustrating separate compilation; which can be found in section 4.12.

All the example programs are designed for boot from link boards. If you have a board that boots from ROM you should set it to boot from link or run the example programs using the T425 simulator tool `isim`.

4.2 occam programs

Within the toolset a single processor program is a single occam procedure with a fixed pattern of formal parameters, as illustrated below.

```
#INCLUDE "hostio.inc"
PROC occam.program (CHAN OF SP fs, ts,
                    []INT memory)
  ... body of program
:
```

The procedure and its parameters can have any legal occam names. You must always supply the procedure with the same type of formal parameters as shown above, to enable communication with the host.

All occam procedures are terminated by a colon (:), at the same indentation as the corresponding PROC keyword. Do not forget the colon at the end of a program.

Program input and output is supported by the host file server, which is resident on the host computer. Access to the host file server is via the i/o libraries, which are described in part 2, chapter 1. Whenever routines from these libraries are used the channels `fs` and `ts` must be passed to the routine so that it can communicate with the host file server.

Channel `fs` comes from the host file server and `ts` goes to the host file server. Both use protocol `SP`, which is defined in the include file `hostio.inc`. Figure 4.1 shows how these channels are connected.

The array `memory` contains the free memory remaining on the transputer evaluation board after the program code has been loaded and the workspace allocated. It is calculated by subtracting the area occupied by the program code and data from the value specified in the `IBOARDSIZE` host environment variable. The `memory` array is passed to the program as an array of type INT, where it can be used. By allowing programs to be run on boards with different memory sizes, this array aids program portability between different boards.

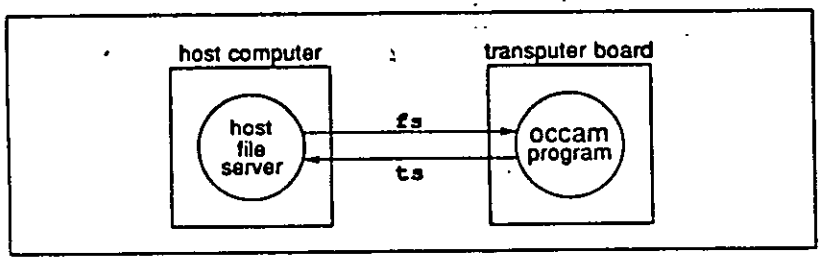


Figure 4.1 Program input/output

4.2.1 Compiling programs

The compiler produces object code in TCOFF format for input to the linker. The compiler is capable of compiling code for any one of a range of transputers (the IMS T212, M212, T222, T225, T400, T414, T425, T800, T801 or T805) in one of three error modes and with interactive debugging either enabled or disabled. The compiler enables interactive debugging by default unless the compiler '`Y`' option is used.

The standard error modes are HALT system and STOP process. A special mode, UNIVERSAL, enables code to be compiled so that it may be run in either HALT or STOP mode. The target processor and error mode must be specified for each compilation, using options on the command line. By default the compiler compiles for an IMS T414 in HALT mode, and when compiling for this transputer type and error mode you may omit the options. In all other cases the options must be supplied.

Other operating features of the compiler may be changed by o[illegible]. See section 25.2 for a full description of these options.

If the compiler detects any errors, the file name and line number of each error is displayed along with a message explaining the error.

If the compilation succeeds, the compiler creates a new code file in the current directory. The filename for the new file is derived from the name of the source file and the default file extension `.tco` is added. The filename can also be specified on the command line.

Compilation information

It is sometimes necessary to check how much workspace (data space) will be required to run the code. This information is stored in the code file produced by the compiler, linker and librarian. To display the information use the '`I`' command line option or use the binary lister tool `ilist`. For details of `ilist` see chapter 20.

4.2.2 Linking programs

When all the component parts of a program have been compiled they must be linked together to form a whole program. Component parts include the main program, any separately compiled units, and any libraries used by the program, including the compiler libraries.

If required, the compiler libraries are automatically loaded by the compiler unless specifically disabled with the compiler '`E`' option. If you are unsure whether your program uses the compiler libraries it is best to always link in the appropriate library. Only library modules actually used by the compiled code will be included in the linked code file. The correct library for your program depends on the transputer type of the compilation.

To assist the user, three linker indirect files are supplied listing the compiler libraries appropriate to different processor types. The relevant file should be included on the linker command line using the '`f`' option. `occam2.lnk` is provided for the T2 series, `occam8.lnk` for the T8 series and `occama.lnk` for other 32-bit transputers.

For further details of the compiler libraries see part 2, section 1.2.

By default, the order in which the code modules are specified on the command line determines their order within the linked unit; library modules being placed after the separately compiled modules. This default can be overruled by using the compiler directive `#PRAGMA LINKAGE` (see section 25.10.7) and the linkage

directive #SECTION (see section 19.3.1). These directives enable the user to prioritise the order in which modules are linked together and so influence the use of on-chip RAM. A map of the linked unit, showing the order of the modules, may be produced by specifying the linker command line option 'MO'.

4.2.3 Viewing code

Object code files produced by compiling or linking programs can be examined using the binary lister tool `ilist`. Information that can be displayed includes procedure definitions, exported names, external references within the code, and symbol data. For more details see chapter 20.

4.2.4 Making bootable programs

Code that has been linked to form a program cannot be loaded directly onto a transputer evaluation board, for two reasons. Firstly, object code produced by the linker and compiler tools contains extra information required by some tools. This information must be removed before the program can be loaded. Secondly, code to be run on a board which boots from link, such as the IMS B004, require the addition of bootstrap information to load the program and start it running.

Extraneous data is removed, and a boot-from-link bootstrap is added, by the collector tool `icollect`.

4.2.5 Loading and running programs

Bootable programs can be loaded onto the transputer evaluation board using the host file server `iserver` (see chapter 22).

The server must be given a number of parameters when it loads a program. All server options are two characters long, with 'S' as the first character. Server parameters are removed from the command line by the server, so you should avoid using the same options for your own program (it is best to avoid giving programs two letter options beginning with the letter 'S').

To load a program use the 'SB' option and specify the file to be loaded. This has the same effect as using options 'SR', 'SS', 'SI', and 'SC' together, that is, it resets the board, provides access to host facilities such as file access and terminal i/o, and loads the program. The 'SI' option directs the tool to display progress information as it loads the file. To terminate when the transputer error flag is set, thereby enabling the program to be debugged, add the server 'SE' option.

Programs can also be loaded onto transputer networks, without using code on the root transputer, by first using the `iskip` tool to set up a skip process and then loading the program using `iserver`. This can be useful when loading programs onto external networks via a transputer evaluation board. It is also useful for debugging programs that normally use the root transputer to run all or part of a program. The debugger always runs on the root transputer. Provided the network has at least one processor which is not used by the program, `iskip` may be used in conjunction with `iserver` to load the program over the root transputer. For details of skip loading see section 6.6.

4.3 Transputer types and classes

This section describes the meaning of transputer types and classes and how selection of the target processor affects the compilation and linking stages of program development. The section describes how to compile and link code targetted at a single processor type and then describes how to compile and link programs so that they can be executed on different processor types. The examples used in this section follow on from the example introduced in chapter 3.

4.3.1 Single transputer type

For those users who have a single transputer or indeed a network of transputers all of the same type, the compilation and linking stages of program development are very straighforward. Simply compile and link all your modules for the required processor.

The compiler and linker both support command line options to select the following processor types:

16-bit processors	T212, M212, T222, T225
32-bit processors	T400, T414, T425, T800, T801, T805

Example to compile and link for a T800:

```
oc simple -T800                                          (UNIX)
ilink simple.tco hostio.lib -T800 -f occam8.lnk

oc simple /T800                                      (MS-DOS/VMS)
ilink simple.tco hostio.lib /T800 /f occam8.lnk
```

The default target processor for both the compiler and linker is a T414, so if you are using this processor type the steps are even simpler:

Transputer class	Processors which class can be run on
T2	T212, M212, T222, T225
T3	T225
T4	T414, T400, T425
T5	T400, T425
T8	T800, T801, T805
T9	T801, T805
TA	T400, T414, T425, T800, T801, T805
TB	T400, T414, T425

Table 4.1 Transputer classes and target processor

```
oc simple                                                    (UNIX)
ilink simple.tco hostio.lib -f occama.lnk

oc simple                                                    (MS-DOS/VMS)
ilink simple.tco hostio.lib /f occama.lnk
```

4.3.2 Creating a program which can run on a range of transputers

The compiler and linker use the concept of transputer class to enable programs to be developed which may be run on different transputer types without the need to recompile.

A transputer class identifies an instruction set which is common to all the processors in that class. When a program is compiled and linked for a transputer class it may be run on any member of that class.

Note: Code created for a transputer class will often be less efficient than code created for a specific processor type. Therefore, creating code for a transputer class is discouraged in situations where program efficiency is a primary concern; it should only be performed where there is a genuine need to produce code which will run on a range of transputers or to reduce the size of a support library, where program efficiency is not a major concern.

Table 4.1 lists all the transputer classes which the compiler and linker support and indicates which processors the program can be run on.

In order to develop a program which will run on different processor types, perform the following steps:

xdoc1 /usr/nts/du705/examples

1 Identify the processors on which the program is to run.

2 Using table 4.1 select the class which may be run on all the target processors.

3 Compile and link all the program modules for this class.

For example to create a program which will run on both a T400 and a T425, compile and link for transputer class T5:

```
oc simple -T5                                                (UNIX)
ilink simple.tco hostio.lib -T5 -f occama.lnk

oc simple /T5                                                (MS-DOS/VMS)
ilink simple.tco hostio.lib /T5 /f occama.lnk
```

Alternatively to create a program which will run on a T400, T425 or a T800, compile and link for transputer class TA.

```
oc simple -TA                                                (UNIX)
ilink simple.tco hostio.lib -TA -f occama.lnk

oc simple /TA                                                (MS-DOS/VMS)
ilink simple.tco hostio.lib /TA /f occama.lnk
```

Programs compiled for the T212, M212 or T222 transputers, which make up class T2, can be run on a T225 (class T3) because a T225 has a similar but larger instruction set than class T2 transputers. Similarly code compiled for a T414 (class T4) may be run on a T400 or T425, which form class T5. The T400 and T425 have additional instructions to those of the T414. Likewise, code compiled for a T800 (class T8) may be run on a T801 or T805, which form class T9. Again the T801 and T805 have additional instructions to those of the T800.

4.3.3 Mixing code compiled for different targets

This section describes how object code compiled for one target processor or transputer class can call and be linked with code compiled for different transputer types or classes.

The ability to do this provides the user with greater flexibility in the use of program modules:

- An individual module can be compiled once e.g. for class T4, and then be called by separate programs to run on different processor types e.g. T414 and T425.

- When the user is preparing a library for use by programs intended to run on different processor types, a single copy of code compiled for a transputer class can be inserted instead of multiple copies for specific transputers.

When linking a collection of compiled units together into a single linked unit, the user must select a specific transputer type or transputer class on which the linked unit is to run. As before, this determines the set of transputer types on which the code will run. When linking for a particular type or class, the linker will accept compilation units compiled for a compatible class. Table 4.2 shows which transputer classes the linker will accept when linking for a particular class.

Link class	Transputer classes which may be linked
T2	T2
T3	T3, T2
T4	T4, TB, TA
T5	T5, T4, TB, TA
T8	T8
T9	T9, T8
TB	TB, TA
TA	TA

Table 4.2 Linking transputer classes

For example if the target processors are a T400 and a T425 the user may compile for classes T5 and TB and link the code for class T5.

Code for a different transputer class can be included in the final linked unit, as long as :

- it uses the instruction set, or a subset of the instruction set, of the link class.

- the calling conventions are the same, (see below).

The same rules must also be followed during the program design stage, when deciding which modules should call each other. Code for a different transputer class can be called provided that it uses the instruction set or a subset of the instruction set of the calling class. This is because the compiler needs to know which modules to select from libraries containing copies for different processor types.

Hence the headings in table 4.2 can be modified slightly to produce table 4.3 which identifies for each class the list of possible classes which it may call.

Calling class	Transputer classes which may be called
T2	T2
T3	T3, T2
T4	T4, TB, TA
T5	T5, T4, TB, TA
T8	T8
T9	T9, T8
TB	TB, TA
TA	TA

Table 4.3 Calling transputer classes

In addition, the order in which the program modules are compiled is affected, in that a module which is called must be compiled before the calling module is compiled. This is explained in section 4.9 and an example is given in section 4.12.

Classes T8 and T9 cannot call or be linked with class TA; this is a change from the iMS D705/D605/D505 versions of the toolset. The reason why these classes cannot be linked together is explained in section 4.3.4, which gives details of the differences between the instruction sets, as additional information.

A library can be made consisting of the same modules compiled for different transputer types or classes. The user then needs only to specify the library file to the linker, and the linker will choose a version of a required routine which is suitable for the system being linked.

The linker uses the rules given in table 4.2 to determine whether a compiled module, found in a library, is suitable for linking with the current system. So, for example, to create a library which may be linked with any transputer class or specific transputer type, all routines could be compiled for classes T2, TA and T8.

If there are a number of possible versions of a module in a library the best one (i.e. the most specific for the system being linked) is chosen.

4.3.4 Classes/instruction sets – additional information

The instruction sets of the transputer classes differ in the following ways:

- Classes T2 and T3 support 16-bit transputers whereas all the other transputer classes support 32-bit transputers.
- Class T3 is the same as class T2 except that T3 has some extra instructions to support CRC and bit operations and includes special debugging functions.
- Class T5 is the same as class T4 except that T5 has extra instructions to perform CRC, 2D block moves, bit operations and special debugging functions.
- Class T9 is the same as class T8 except T9 has additional debugging instructions.
- The T800, T801 and T805 processors use an on-chip floating point processor to perform REAL arithmetic. Thus a large number of floating point instructions are available for these transputers and for their associated classes T8 and T9. These instructions are listed in part 2, section B.6.
- For the T414, T400 and T425 processors i.e. transputer classes T4 and T5 the implementation of REAL anthmetic is in software. These transputers make use of a small number of floating point support instructions listed in part 2, section B.5.
- The instruction set of class TA only uses instructions which are common to the T400, T414, T425, T800, T801 and T805 transputers. Therefore it does not use the floating point instructions, the floating point support instructions or the extra instructions to perform CRC, 2D block moves or special debugging or bit operations.
- The instruction set of class TB only uses instructions which are common to the T400, T414 and T425 processors. Therefore it uses the floating point support instructions, but does not use the extra instructions to perform CRC, 2D block moves or special debugging or bit operations.

Note: code which includes CRC, 2D block moves and floating point operations implemented by ASM or GUY code cannot be compiled for classes TA or TB. The compiler will report an error if this is attempted.

When considering the similarities and differences in the instruction sets of different transputer classes it helps to divide them into the three separate structures as shown in figure 4.2.

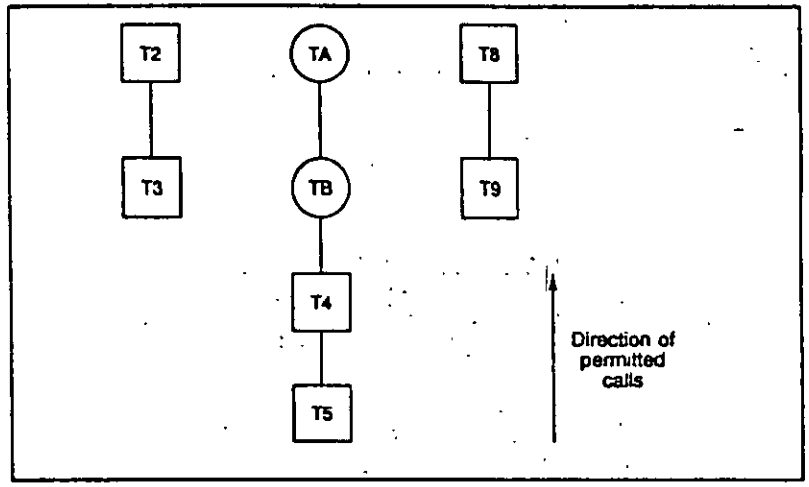


Figure 4.2 Structures for mixing transputer types and classes

By comparison with tables 4.2 and 4.3 it can be seen that a module may only call and be linked with modules compiled for a transputer class which belongs to the same structure.

Classes T2 and T3 which form the first structure are targetted at 16-bit transputers so it is obvious that they cannot be linked with the other classes which are all targetted at 32-bit transputers.

The reason why classes T8 and T9 cannot call or be linked with classes TA, TB, T5 or T4 is because floating point results from functions are returned in a floating point register for T8 and T9 code and in an integer register for all other 32-bit processors. Even if your code does not perform real anthmetic, linking code compiled for a T9 or T8 with code compiled for any of the other classes is not permitted.

To summarise, compiling code for the transputer classes TA and TB enables it to be run on a large number of transputer types, however, the code may not be as efficient as code compiled for one of the other transputer classes or for a specific transputer type. For example compiling code for class T5 enables the CRC and 2D block move instructions to be used, whereas these instructions are not available to code compiled for classes TA and TB.

4.4 Error modes

For systems that require maximum security and reliability, the error behaviour is of great concern. occam 2 specifies that run-time errors are to be handled in one of three ways, each suitable for different programs. The error mode to be used is supplied as a parameter to both the compiler and linker. The options are listed in table 4.4.

Option(s)	Description
H	HALT mode
S	STOP mode
X	UNIVERSAL mode

Table 4.4 Compiler and linker options for selecting error mode

The first mode, called HALT system mode, causes all run-time errors to bring the whole system to a halt promptly, ensuring that any errant part of the system is prevented from corrupting any other part of the system. This mode is extremely useful for program debugging and is suitable for any system where an error is to be handled externally. HALT system mode is the default for the compiler, and you should use this mode when you may want to use the debugger.

Note: on the IMS T414, T222 and M212, HALT mode does not work for processes running at high priority, as the **HaltOnError** flag is cleared when going to high priority.

The second mode, called STOP mode, allows more control and containment of errors than HALT mode. This maps all errant processes into the process STOP, again ensuring that no errant process corrupts any other part of the system. This has the effect of gradually propagating the STOP process throughout the system. This makes it possible for parts of the system to detect that another part has failed, for example, by the use of 'watchdog' timers. It allows multiply-redundant, or gracefully degrading systems, to be constructed.

The third mode, called UNIVERSAL mode, may behave as either HALT or STOP mode depending on the transputer's Halt-On-Error flag. For example if a library is compiled in UNIVERSAL mode, it may be linked in HALT mode with HALT mode modules and it will behave as if it had been compiled in HALT mode. Alternatively if it is linked in STOP mode with STOP mode modules it will behave as if it had been compiled in STOP mode.

If a program, targetted at a single processor, is compiled and linked in UNIVERSAL, the collector tool will treat the linked unit as though it had been linked in the default error mode which is HALT mode.

All separately compiled units for a single processor must be compiled and linked

with compatible error modes. Where a library is used the module of the appropriate error mode will be selected.

Code which is compiled in either HALT or STOP mode can call code compiled in UNIVERSAL mode, however code compiled in UNIVERSAL mode may only call code which has also been compiled in UNIVERSAL mode. Code which has been compiled in HALT mode may not call or be called by code compiled in STOP mode. The linker will report an error if user attempts to link HALT and STOP modules together.

4.4.1 Error detection

In some circumstances it may be desirable to omit the run time error checking in one part of a program, for example, in a time-critical section of code, while retaining error checks in other parts of a program, for debugging purposes.

The compiler provides three command line options to enable the user to control the degree of run time error detection; they are the 'K', 'U' and 'NA' options and they prevent the compiler from inserting code to explicitly perform run time checks.

These options should only be used on code which is known to be correct. The compiler does not insert a lot of error checking code so it should only be disabled as a last resort.

It is the user's responsibility to ensure that errors cannot occur. The ability to disable certain error checking code by using the 'K' and 'U' options should not be abused in an attempt to use illegal code, since there is no way of telling the compiler to ignore all errors.

The 'K' option disables the insertion of code to perform run time range checking. In this context range checking only includes checks on array subscripting and array lengths. **Note:** in any situation where the compiler can detect a range check error without specifically adding code, it *may* still do so. The type of situation where this is likely to happen is when an array subscript such as $[i + j]$ is used, and $i + j$ overflows.

The 'U' option disables the insertion of code whose only purpose is to detect some kind of error. This option is stronger than the 'K' option, and includes the 'K' option, so it is not necessary to use both options together. **(Note:** that the 'U' does not include the 'NA' option which is described below).

The 'U' option will disable the insertion of run-time checks to detect occurrences such as the following:

negative values in replicators
errors in type conversion values,
errors in the length of shift operations,
zero length moves,
array range errors,
errors in replicated constructs such as SEQ, PAR, IF and ALT.

Note: again in any situation where the compiler can detect an error without specifically inserting code, it *may* still do so. Thus arithmetic overflows, etc, can still cause an error. (To avoid overflow errors the operators PLUS, MINUS and TIMES can be used).

If the 'U' option is used in conjunction with HALT mode, it will prevent explicit checking for floating point errors in those cases where library calls are not used to perform floating point arithmetic (see below). In addition if the 'U' option is used with STOP or UNIVERSAL mode, it inhibits the ability of the system to gradually propagate a STOP process throughout the system. This means that the 'U' option, when used with any error mode produces identical code. The object file, however, is still marked as being compiled in a particular error mode.

Thus, faster code is produced by using the 'U' option with any error mode. Any libraries which are linked with the modules will maintain the error mode and level of error detection that they were compiled for. In practice, libraries compiled in HALT mode will be fastest, so for benchmarking, modules should be compiled in HALT mode and the 'U' option used.

If the user requires the equivalent of the UNIVERSAL error mode implemented by the IMS D705/D605/D505 versions of the toolset, then UNIVERSAL error mode should be used and the 'U' option specified. However, the compiler will not incorporate library entries compiled with the 'U' option.

The following points summarise the differences in the implementation of error detection between the current release and previous releases of the toolset i.e the IMS D705/D605/D505 toolsets.

Comparison of error modes with the IMS D705/D605/D505 toolsets

The detection of errors and the action that is taken when an error is detected are separated in the current toolset.

HALT and STOP mode behave the same as they did in the previous toolsets.

UNIVERSAL mode no longer turns error detection off, instead it produces code which may be linked in either HALT or STOP mode.

The degree of run-time checking may be reduced by using the 'K' and 'U' command line options.

To obtain the equivalent of the UNIVERSAL mode implemented by the IMS D705/D605/D505 toolsets, compile in UNIVERSAL mode and use the 'U' option. Note: this will not cause the compiler to incorporate libraries compiled with the 'U' option.

To obtain the equivalent of occam UNDEFINED mode (see the *'occam 2 Reference Manual'*), compile in any error mode and use the 'U' option.

The 'NA' option disables the insertion of code to check calls to ASSERT.

The occam 2 compiler recognises a procedure ASSERT with the following parameter:

```
PROC ASSERT (VAL BOOL test)
```

At compile time the compiler will check the value of test and if it is FALSE the compiler will give a compile time error; if it is TRUE, the compiler does nothing. If test cannot be checked at compile-time then the compiler will insert a run-time check to detect its status. The 'NA' option can be used to disable the insertion of this run-time check.

4.5 Interactive debugging

The compiler and linker tools support interactive debugging by default. When interactive debugging is enabled the compiler or linker will generate calls to library routines to perform channel input and output rather than using the transputer's instructions. This does cause a performance penalty to be incurred when interactive debugging is enabled. Disabling interactive debugging by using the command line option 'Y' results in faster code execution.

Interactive debugging must be enabled in order to use the interactive features of the debugger. However, the debugger does not have to be present in order to run the code.

Code which has interactive debugging disabled may call code which has interactive debugging enabled but not vice versa. If interactive debugging is disabled for any module in a program this will prevent the whole program from being debugged interactively.

4.6 Alias and usage checking

The compiler implements the alias and usage checking rules described in the *'occam 2 Reference Manual'*. Alias checking ensures that elements are not referred to by more than one name within a section of code. Usage checking ensures that channels are used correctly for unidirectional point-to-point communication, and that variables are not altered while being shared between parallel processes. For a further discussion of the rationale behind these rules, see sections 25.13 and 25.14. Information is also given in *The Transputer Applications Notebook – Architecture and Software, Chapter 6 – The development of occam 2*'.

Alias and usage checking during compilation may be disabled by means of the compiler options '**A**' and '**N**'. Using the '**N**' option it is possible to carry out alias checking without usage checking. However, it is not possible to perform usage checking without alias checking, as the usage checker relies on lack of aliasing in the program. If you switch off alias checking with option '**A**', usage checking is automatically disabled.

The '**K**' and '**U**' options will also disable the insertion of alias checks that would otherwise be performed at run-time. These options do not affect the insertion of alias checks at compile time nor the insertion of usage checks which are only performed at compile time.

Alias checking can impose some code penalties, for example, extra code is inserted if array accesses are made which cannot be checked until runtime. The '**WO**' command line option will produce a warning message every time one of these checks is generated. However, alias checking can also improve the quality of code produced, since the compiler can optimise the code if names in the program are known not to be aliased.

The compiler usage check detects illegal usage of variables and channels, for example, attempting to assign to the same variable in parallel. The compiler performs most of its checks correctly, but with certain limitations. Normally, if it is unable to implement a check exactly, it will perform a stricter check. For example, if an array element is assigned to, and its subscript cannot be evaluated at compile time, then the compiler assumes that all elements of the array are assigned to. If a correct program is rejected because the compiler is imposing too strict a rule, it is possible to switch off usage checking.

It should also be noted that usage checking can slow the compiler down. For example, programs which contain replicated constructs defined with constant values for the *base* and *count*, will be checked for each iteration of the routine. Replicated constructs which have variable *base* and *count* values are only checked once with a stricter check, because the compiler cannot evaluate, at this point, the actual limits of the replication.

4.7 Using separate vector space

The compiler normally produces code which uses separate vector space. Arrays which are declared within a compilation unit are allocated into a separate 'vector space' area of memory, rather than into workspace when they are either:

- greater than 8 bytes or
- greater than 1 word, where the elements are smaller than a word (e.g. `[5]BYTE`).

This decreases the amount of stack required, which has two benefits: firstly, the offsets of variables are smaller, access to them is faster; secondly, the total amount of stack used is smaller, allowing better use to be made of on-chip RAM.

The compiler option '**V**' disables the use of a separate vector space, in which case arrays are placed in the workspace.

When a program is loaded onto a transputer in a network, memory is allocated contiguously, as shown in figure 4.3.

This allows the workspace (and possibly some of the code) to be given priority use of the on-chip RAM. Generally, the best performance will be obtained with the separate vector space enabled.

The default allocation of an array can be overridden by an allocation immediately after the declaration of an array. This allocation has one of the forms:

```
PLACE name IN VECSPACE :
```

or

```
PLACE name IN WORKSPACE :
```

For example, in a program which is normally using the separate vector space, it may be advantageous to put an important buffer into workspace, so that it is more likely to be put into internal RAM. The program would be compiled with

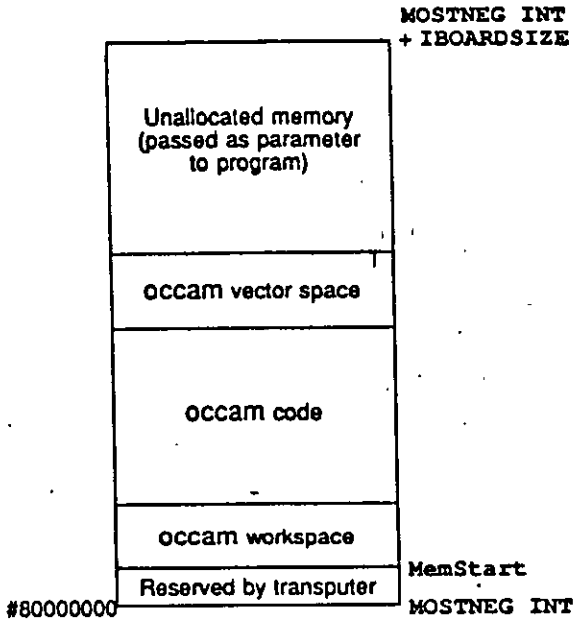


Figure 4.3 Memory allocation on a 32-bit transputer

separate vector space enabled, but would include something like:

```
[buff.size]BYTE crucial.buffer :
PLACE crucial.buffer IN WORKSPACE :
```

For a program where it is required to put all of the data apart from one large array into the workspace, the program would be compiled with separate vector space disabled, and the array allocated to vector space by a place statement such as `PLACE large.array IN VECSPACE`.

Within a program it is possible to mix code compiled with separate vector space on and code compiled with separate vector space off. The parts of the program which have been compiled with separate vector space enabled will be given use of the vector space.

Note that certain libraries such as `hostio.lib` use vector space. Therefore, it is likely that some use of vector space will be made, even if vector space is disabled for a program module.

4.8 Sharing source between files

The source of a program can be split over any number of files by using the `#INCLUDE` directive. This directive enables the user to specify a file which contains occam source. The contents of this file are included in the source at the same point and with the same indentation as the `#INCLUDE` directive. Include files may be nested to a maximum depth of twenty. By convention `.inc` file extension should be used for occam constant and protocol definitions. An example of using the `#INCLUDE` directive is given below:

```
#INCLUDE "infile.inc"  -- source in infile.inc
```

The name of the file to be included is placed in quotes. All of the line following the closing quote may be used as for comments. Directives must occupy a single line.

4.9 Separate compilation

Separate compilation reflects the hierarchical structure of occam, and the occam compiler compiles occam procedures and/or functions (`PROC`s and `FUNCTION`s). Any number of procedures and/or functions may be compiled at any time, provided the only external references they make are via their parameter lists.

A group of procedures and/or functions that are compiled together are known as a compilation unit. Each procedure and/or function in such a group may be called internally by other procedures declared later in that group, or externally by any occam in the scope of the directive which references that separate compilation unit. Constant declarations and protocols are also permitted inside a compilation unit, for the use of the procedures and functions within it. The scope of a separate compilation unit is the same as any normal occam procedure or function.

Separately compiled units are referenced from occam source as object code files, using the `#USE` directive. The object file may be a compiled (`.tco`) or library (`.lib`) file. If the file extension is omitted the compiler adds the extension of the current output file. This will be (`.tco`) unless an output file has been specified using the 'O' option.

An example of how to reference a separately compiled unit is shown below.

```
#USE "scunit.tco"  -- code in file scunit.tco
```

The filename must be enclosed in double quotes. All of the line following the closing quote can be used as comment. The directive must occupy a single line.

Separate compilation units may be nested to any depth and may contain #INCLUDE directives. They may also use libraries, as described in section 4.11.

A separate compilation unit must be compiled before the source which references it can be compiled.

4.9.1 Sharing protocols and constants

occam constants and protocols may be declared and used within a compilation unit according to the rules of the language. Where a constant and/or protocol is to be used across separate compilation boundaries, it should always be placed in a separate file. The file should be referenced in any compilation unit where it is needed, by using the #INCLUDE directive before any #USE directive, which introduces procedures using the protocol in their formal parameter lists. Protocols will also need to be referenced in any enclosing compilation unit (because the channels will either be declared there or passed through). For example, suppose we have a protocol P defined in a file myprot.inc. We might then use it as follows:

```
PROC main()
  #INCLUDE "myprot.inc"
  #USE "mysc.tco"

  CHAN OF P actual.channel :
  PAR
    do.it(actual.channel)
    ...
:
```

The separately compiled procedure do.it, in the file mysc.occ, would look like this:

```
#INCLUDE "myprot.inc" -- declares protocol P
PROC do.it (CHAN OF P in)

  SEQ
    ... body of procedure
:
```

Since the protocol name P occurs in the formal parameter list of the separately compiled procedure do.it, the compilation unit must include a #INCLUDE directive, preceding the declaration of do.it, to introduce the name P.

4.9.2 Compiling and linking large programs

Building a program which includes separate compilation units and library references is straightforward. Separate compilation units in the program can be compiled individually by applying the compiler to them. Nested compilation units must be compiled in a bottom-up order before the top level of the program is compiled; finally the whole program is linked together.

Separate compilation units must be compiled before the unit which references them can be compiled. This is because the object code contains all the information about a unit (names, formal parameters, workspace and code size, etc.) which is needed to arrange the static allocation of workspace and to check correctness across compilation boundaries. This information may be viewed using the ilist tool.

When a program is linked the code for all the separate compilation units in the program is copied into a single file. In addition, code for any libraries used is included in the file. Where libraries contain more than one module, only those modules containing routines actually required in a program are linked into the final code. This helps to minimise the size of the linked code.

The target processor or transputer class and error mode must be specified to the linker to enable it to select appropriate library modules. Only one processor type or class may be used for the linking process and this must be compatible with the transputer type or class used to compile the modules. The error mode used for the linking process must also be compatible with the error mode(s) used to compile the modules. Compatible use of the compiler and linker 'Y' option must also be adopted for the modules to be linked.

If there are a large number of input modules, they may be supplied to the linker, within an indirect file, as a list of filenames. Indirect files may also contain directives to the linker. Linker directives enable the user to customise the linkage operation and include facilities to modify the use of workspace, create forward references to symbols and to nest indirect files. Chapter 19 provides detailed information of how to run and use the linker.

4.10 Using imakef

When a change is made to part of a program it is necessary to recompile the program to create a new code file reflecting the change. The purpose of the separate compilation system is to split up a program so that only those parts of the program which have changed or which depend on the changed units, need to be recompiled, rather than needing to recompile the whole program. However, it would be tedious to have to remember which modules had been edited, which modules might be affected by calls and the order in which the modules were

compiled and linked. For this reason a Makefile generator **imakef** is supplied with the toolset and may be used to assist with building programs consisting of several modules. This tool, when applied to a program (or part of a program), compiles a list of dependencies of compilation units and uses this list to produce a Makefile. The Makefile can be used with a suitable MAKE program to recompile only the changed parts of a program. This ensures that compilation units will always be recompiled where a change has made this necessary.

To use the Makefile generator you must tell it the name of the file you wish to build. The tool can produce a Makefile for any type of file that can be built with the toolset tools. In order for **imakef** to be able to identify file types, a different system of file extensions must be used to that used in this chapter. The file name rules for **imakef** are described in chapter 21 together with details of how to use the tool.

4.11 Libraries

A library is a collection of compiled procedures and/or functions. Any number of separately compiled units may be made into a library by using the librarian. Separately compiled units and libraries can be added to existing libraries. Each compilation unit is treated as a separately loadable module within a library. When compiling or linking, only modules which are used by a program are loaded. The rules for selective loading are described in the following section.

Libranes are referenced from occam source by the **#USE** directive. For example:

```
#USE "hostio.lib"      -- host server library
```

The filename is enclosed in quotes. The rest of the line, following the closing quote, may be used for comments. Directives must occupy a single line.

Libraries should always use a **.lib** file extension, and this must always be supplied in a **#USE** directive.

4.11.1 Selective loading

Each module (separately compiled unit) in a library is selectively loadable by the linker; i.e. parts of a library not used or unusable by a program are ignored. The unit of selectivity is the library module; i.e. if one procedure or function of a library module is used then all the code for that module is loaded.

The compiler is selective when a library is referenced. Only modules of a library that are of the same, or compatible, transputer type or class, error mode and

method of channel input/output, are read (see sections 4.3, 4.4 and 4.5).

Selective loading is based on the following rules:

1 The transputer type or class of a library module must be the same as, or compatible with, the code which could use it.

2 The error mode of the library module must be the same as, or compatible with, the code which could use it.

3 The interactive debugging mode (i.e. whether interactive debugging is enabled or not) of the library must be the same, or compatible with, the code which could use it.

4 At least one routine (entry point) in a module is called by the code.

Rules 1 to 3 apply to the compiler. All the rules are used by the linker. The compiler only selects on transputer type, error mode and method of channel input/output. It is not until the linking stage that unused modules are rejected. For details on mixing processor classes and error modes see sections 4.3 and 4.4 respectively.

4.11.2 Building libraries

Libranes are built using the librarian tool **ilibr**. Libranes can be created from either separately compiled units (**.tco** or library files **.lib**) or from linked units (**.lku** files) but not a combination of both. The librarian takes any number of input files and combines them into a single library file. Each separately compiled unit forms a single module in the library.

When forming a library the librarian will warn if there are multiply defined routines (entry points). In other words, for each combination of transputer type, error mode and method of channel input/output there may only be one routine with a particular name. For further information on building libranes see chapter 18.

As an example consider building a library called **mylib.lib**. The source of this library is contained in a file called **mylib.occ** and has been written to be compilable for both 16 and 32 bit transputers. We want the library to be available for T212 and T800 processors in halt on error mode only. Having compiled the source for the two processors we will have two files, for example: **mylib.t2h** and **mylib.t8h**. To form a library from these compilation units use the following command line:

```
ilibr mylib.t2h mylib.t8h
```

When an output filename is not specified, as in this example, the librarian uses the first file in the list to make up the output file name and adds the extension `.lib`. In this case it will write the library to the file `mylib.lib`.

The librarian can also take an indirect file containing a list of the files to be built into the library. Such files should have the same name as the library, but with a `.lbb` file extension. So, still using the above example, if the names of the files to make up the library were put in a file called `mylib.lbb`, we could then build the library using one of the following commands:

```
ilibr -f mylib.lbb -o mylib.lib        (UNIX)
ilibr /f mylib.lbb /o mylib.lib        (MS-DOS/VMS)
```

Compiled modules can be added to an existing library file. However, if the librarian attempts to create an output file with the same name as an input library file, an error will be produced. This can be avoided by specifying a different output filename using the 'o' option. Alternatively if one on the compiled modules to be added to the library has a different name, this could be specified first on the command line. Once the new library file has been created it can be renamed if necessary. Adding modules to an existing library does not require programs which call it, to be recompiled, provided it is given its original name in its final form.

The Makefile generator `imakef` can be used to assist with the building of libraries. This is particularly useful where libraries are nested within other libraries or compilation units, because `imakef` can identify the dependencies of libraries on other modules or separately compiled units. For further information about the `imakef` tool see chapter 21.

For further details of how to use the librarian and how to optimise libraries see chapter 18.

4.12 Example program – the pipeline sorter

This section introduces an example which serves to show how a large program might be structured, in terms of separate compilation units, libraries, and a shared protocol.

4.12.1 Overview of the program

The program sorts a series of characters into the order of their ASCII code values.

Figure 4.4 shows the basic structure of this program. There are three processes:

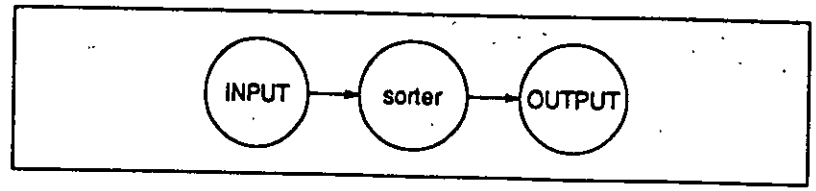


Figure 4.4 Basic structure of sorter program

the input process, the output process and the sorter process. We can decompose the sorter process by using a pipeline structure. This uses the algorithm described in *'A tutorial introduction to occam programming'*. If we design the pipeline carefully we can ensure that each element of the pipeline is identical to all the other elements. The pipeline is served by an input process, which reads characters from the keyboard, and an output process which writes the sorted characters to the screen. Figure 4.5 shows the structure of the program using a pipeline.

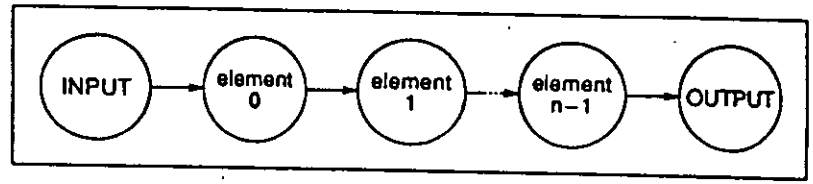


Figure 4.5 Pipeline of n elements

An obvious implementation would be to write an occam process for each process in figure 4.5, using a replicated process for the pipeline. Communication between the processes is via occam channels and to aid program correctness we should use an occam `PROTOCOL` for these channels. This protocol must be shared by all the processes. As the occam compiler compiles processes (`PROC`s) and as each of the processes is independent we can implement each one as a separately compiled unit. The processes share a common protocol and the best way to ensure consistency is to place the protocol in a separate file and use the `#INCLUDE` mechanism to access it. These processes can then be called in parallel by an enclosing program which can access the code of each process by the `#USE` mechanism.

There is a problem with this implementation because two processes require access to the host file server. The host file server is accessed via a pair of occam channels and occam does not allow the sharing of channels between processes. There are a number of ways around this problem. One solution is to use a multiplexor process for the server channels, as described in section 8.5. Another solution is to merge the two processes into a single process. This solution is used because the program accesses the server in a sequential manner (read a line then display sorted line, read a line etc.). Figure 4.6 gives the final

process diagram for the program.

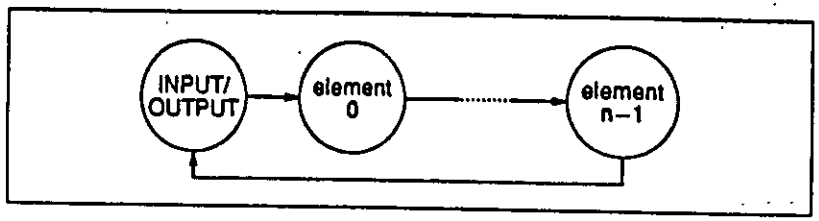


Figure 4.6 Program with combined input/output process

The implementation can be split into four files:

`element.occ`	the pipeline sorting element
`inout.occ`	the input/output process
`sorter.occ`	the enclosing program
`sorthdr.inc`	the common protocol definition

Figure 4.7 shows the way these files are connected together to form a program.

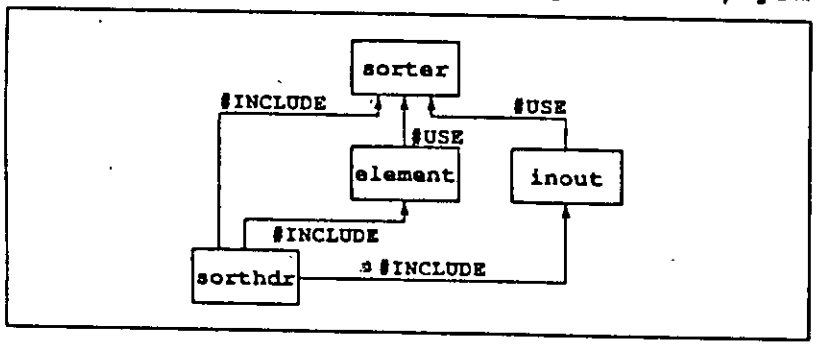


Figure 4.7 File structure of program

The source of the program is given below and is supplied in the 'examples' directory. You can either copy these files to a working directory or you can type in the source as given below. For details of the toolset directories see the Delivery Manual that accompanies the shipment.

Two other files are required to complete the program. These are the host file server library `hostio.lib` and the corresponding `.inc` file containing the host file server constants.

4.12.2 The protocol

Declarations of constants and channel protocols are contained in the include file `sorthdr.inc`, which is listed below.

```
PROTOCOL LETTERS
  CASE
    letter; BYTE
    end.of.letters
    terminate
:

VAL number.elements IS 100:
```

This declares a protocol called `LETTERS`, which permits three different types of message to be communicated:

`letter`	– followed by the character to be sorted.
`end.of.letters`	– marks the end of the sequence to be sorted.
`terminate`	– signals the end of the program.

The constant `number.elements` is also declared. This defines both the number of sorting elements in the pipeline and the maximum length of the sequence of characters that can be sorted.

4.12.3 The sorting element

The sorting element `element.occ` is listed below:

```
#INCLUDE "sorthdr.inc"

PROC sort.element (CHAN OF LETTERS input, output)

  BYTE highest:
  BOOL going:

  SEQ
    going := TRUE
    WHILE going
      input ? CASE
        terminate
          going := FALSE
```

```
        letter; highest
          BYTE next:
          BOOL inline:
          SEQ
            inline := TRUE
            WHILE inline
              input ? CASE
                letter; next
                  IF
                    next > highest
                      SEQ
                        output ! letter; highest
                        highest := next
                    TRUE
                      output ! letter; next
                end.of.letters
                  SEQ
                    inline := FALSE
                    output ! letter; highest
            output ! end.of.letters
    output ! terminate
:
```

This program consists of two loops, one nested inside the other. The outer loop accepts either a termination signal or a character sequence for sorting. If it receives a character it enters the inner loop. The inner loop reads characters until it receives an 'end of letters' signal, signifying the end of the string of characters to be sorted. The sort is performed by storing the highest (ASCII) value character it receives and passing any lesser (or equal) characters on to the next process. The 'end of letters' tag causes the stored value to be passed on and the inner loop terminates.

The maximum number of characters which can be sorted is determined by the number of sorter processes. One character is sorted per process.

4.12.4 The input/output process

This process consists of a loop which reads a line from the keyboard, then sends the line to the sorter and, in parallel, reads the sorted line back. It then displays the sorted line. If the line read from the keyboard is empty the loop is terminated. At the end of the process the host file server is terminated with the success constant `sps.success`, which is defined in the file `hostio.inc`.

If any I/o errors occur the program will stop, allowing it to be examined by the debugger.

The input/output process `inout.occ` is listed below.

```
#INCLUDE "sorthdr.inc"
#INCLUDE "hostio.inc"

PROC inout (CHAN OF SP fs, ts,
            CHAN OF LETTERS from.pipe, to.pipe)

  #USE "hostio.lib"

  [number.elements - 1]BYTE line, sorted.line:
  INT line.length, sorted.length:
  BYTE result:
  BOOL going:

  SEQ
    so.write.string.nl (fs, ts,
    "Enter lines of text to be sorted *
    *- empty line terminates")
    going := TRUE
    WHILE going
      SEQ
        so.read.echo.line(fs, ts, line.length,
                          line, result)
        IF
          result <> spr.ok
            STOP -- stop if an error occurs
          TRUE
            so.write.nl (fs, ts)
        PAR
          SEQ
            IF
              (line.length = 0)    -- no more input
                to.pipe ! terminate
              TRUE
                SEQ
                  SEQ i = 0 FOR line.length
                    to.pipe ! letter; line[i]
                  to.pipe ! end.of.letters
          BOOL end.of.line:
          SEQ
            end.of.line := FALSE
            sorted.length := 0
            WHILE NOT end.of.line
              from.pipe ? CASE
                terminate
```

```
                SEQ
                  end.of.line := TRUE
                  going := FALSE
              letter; sorted.line[sorted.length]
                sorted.length := sorted.length + 1
              end.of.letters
                SEQ
                  so.write.string.nl(fs, ts,
                    [sorted.line FROM 0
                     FOR sorted.length])
                  end.of.line := TRUE
  so.exit(fs, ts, sps.success) -- terminate server
:
```

4.12.5 The calling program

This process calls the input output process in parallel with the sorter elements, in a pipeline. The `memory` parameter must be declared, but the program does not use it.

The calling program `sorter.occ` is listed below.

```
#INCLUDE "hostio.inc"

PROC sorter (CHAN OF SP fs, ts, []INT memory)

  #USE "hostio.lib"  -- host i/o library
  #INCLUDE "sorthdr.inc"

  #USE "inout"       -- separately compiled units
  #USE "element"

  [number.elements + 1]CHAN OF LETTERS pipe:
  PAR    -- run pipe between i/o processes
    inout(fs, ts, pipe[number.elements], pipe[0])
    PAR i = 0 FOR number.elements
      sort.element(pipe[i], pipe[i + 1])
:
```

4.12.6 Building the program

To build the program, first compile each component of the program separately, link them together, and add bootstrap code to the main compilation unit.

The program's components must be compiled in a bottom up fashion, that is, `element.occ` and `inout.occ` first (in either sequence), followed by the main program `sorter.occ`.

First, compile the sorting element `element.occ` using the following command:

```
oc element
```

The file extension can be omitted on the command line because the source file has the conventional extension `.occ`.

The compiler produces a file called `element.tco`, compiled for a T414 in HALT mode.

Compile the input/output process using the following command:

```
oc inout
```

The compiler will produce a file called `inout.tco`, compiled for a T414 in HALT mode.

Then compile the main body using the command line:

```
oc sorter
```

The compiler will produce a file called `sorter.tco`, compiled for a T414 in HALT mode.

Having compiled all the components of the program you can now link them together to form a whole program. Any libraries used by the program must also be specified to the linker. The library `hostio.lib` is the server library used by this program. Remember the include file, `occama.lnk`, which identifies the other libraries, such as compiler libraries, required in the linking process. (See section 4.2.2). To link the files use one of the following commands:

```
ilink sorter.tco inout.tco element.tco hostio.lib -f occama.lnk
ilink sorter.tco inout.tco element.tco hostio.lib /f occama.lnk
```

When specifying options for any of the tools remember to use the correct prefix character for your version of the toolset ('-' for UNIX implementations, and '/' for the IBM PC and VAX/VMS implementations).

The linker will create the file `sorter.lku` linked for a T414 in HALT mode.

If a main entry point is not specified, the linker uses the first valid entry point that it encounters in the input. Therefore, in the above example, it is important

to list the file 'sorter.tco' first. A main entry point may be specified within an indirect file using the linker directive #mainentry or on the command line using the 'MI' option.

Before you can run the program you must add bootstrap code. To do this use the collector tool icollect, using one of the following command lines:

```
icollect sorter.lku -t          (UNIX)
icollect sorter.lku /t          (MS-DOS/VMS)
```

The 't' option informs the collector tool that the input file is a linked unit rather than the output of the configurer tool. (The configurer is used for multi-processor applications).

The collector tool will create the files sorter.btl and sorter.cfb. The .btl file contains the bootable program code. The .cfb file is a configuration binary file which is created by icollect as a by-product of creating the bootable file; it is redundant as far as this example is concerned.

To run the program on a transputer board use one of the following commands:

```
iserver -se -sb sorter.btl      (UNIX)
iserver /se /sb sorter.btl      (MS-DOS/VMS)
```

The 'sb' option specifies the file to be booted and loads the program onto the transputer board. It has the effect of resetting the board, opening communication with the host, and loading the program onto the network. The 'se' option directs the server to terminate if the program sets the error flag. For more details about the server options see chapter 22.

The program reads characters from the keyboard, sorts the line and redisplays it. The program will run until input is terminated by typing RETURN on an empty line.

Figure 4.8 shows an example of the screen display, obtained by running sorter.btl on a UNIX based toolset. The user inputs the string 'Sorter program ' and terminates the program by pressing RETURN.

```
iserver -se -sb sorter.btl

Enter lines of text to be sorted - empty line terminates

Sorter program
 Sorter program
```

Figure 4.8 Example output produced by running sorter.btl.

To run the program using the simulator use one of the following commands:

```
isim -bq sorter.btl             (UNIX)
isim /bq sorter.btl             (MS-DOS/VMS)
```

The 'bq' option specifies batch quiet mode which causes the simulator to run the program and then terminate. For more details about how to use the simulator see chapter 23.

4.12.7 Automated program building

The imakef tool can be used to automate the development process. From the above example it can be seen that there are many steps to go through when building a program of any size. Some of these steps must be performed in a specific order and if part of the program were changed then all affected parts must be recompiled and relinked etc.

MAKE is a common tool for building programs. It uses information about when files were last updated, and performs all the necessary operations to keep object and bootable files up to date with changes in any part of the source. Makefiles are the standard method of providing the MAKE program with the information it needs.

The occam toolset is designed in such a way that it is possible for a tool to construct Makefiles to build occam programs. The Makefile generator imakef produces Makefiles in a format acceptable to most MAKE programs.

imakef requires the user to adopt a particular convention of file extensions. The user then only has to specify the target file he requires i.e. a bootable file and imakef, using its knowledge of file names rules, creates a suitable Makefile. This file has full instructions on how to build the program.

By running the MAKE program for the file the entire program will be automatically compiled, linked and made bootable, ready for loading onto the transputer.

For more details about the imakef tool and an example of how to create a makefile for the pipeline sorter program used in this chapter, see chapter 21.

5 Configuring transputer networks

This chapter describes how to build programs that run on networks of transputers. It describes how to configure an occam program for a network of transputers using the occam configurer tool `occonf` and describes how to load the program onto a transputer network. These procedures are illustrated with an example program for four transputers.

The chapter introduces the configuration language, whose syntax is specified in part2, appendix E and the configurer tool `occonf`, described in chapter 26. This chapter also includes examples illustrating various aspects of configuration.

5.1 Introduction

In order to build programs for multitransputer networks a program is split into a number of self contained components, and each of these is implemented as an occam process. Each process may communicate with other processes resident on the same transputer or, via links, with processes on other transputers.

Programs consisting of occam processes can be run on single or multiple transputers, in any combination. Performance requirements can be met by adapting the application to run on differing numbers of transputers, and by using differing network topologies. The mapping of processes to processors on a transputer network is known as configuration.

Transputer programs can be configured to run on any physical network of transputers. They can be configured to be loaded from an external host down a transputer link, or to be loaded from ROM.

Configuration is achieved by including the program in a configuration description written in the occam configuration language. A configuration description is created by the user as a text file using the configuration language which is an extension of occam. The file is expected by `occonf` to have the file extension `.pgm`. A configuration description may be processed by the configurer tool to generate a configuration data file, which in turn may be processed by the collector tool `icollect` to generate a transputer loadable file.

Conventional file name extensions may be used for these various file types to facilitate the construction of Makefiles using the Makefile generator tool. Chapter 21 describes how to use the Makefile generator for program development and the extensions which should be used.

Within a configuration description the hardware network and the software description are kept separate. This enables the software description to be used for running the same parallel program on a variety of alternative hardware networks. Likewise a particular physical network may be described once for use in a variety of configurations describing different programs that may be run on the same network.

By using the facilities for calling other languages from occam, programs compiled from mixed language sources may also be configured using the occam configurer. (These facilities enable the foreign language code to be incorporated into the occam program as equivalent occam processes. An example of this is provided in the **examples** directory supplied with the toolset. A description of this method of mixed language programming is given in *ANSI C toolset user manual*). Similarly it is possible to configure occam modules (which are called by C programs) using the configurer provided with the ANSI C toolset. Details of how to do this are also given in the *ANSI C toolset user manual*.

5.2 Configuration model

The configuration model consists of the following parts:

- A hardware network description which declares a network as a connected graph of processors.
- A software description in the form of an occam process.
- A mapping between the processes and channels of the software and the nodes (processors) and arcs (transputer link connections) of the network. The mapping is achieved by declaring names and, in the scopes of these declarations, referring to the names in the structures of the configuration description. Normal occam scope rules apply.

The software description takes the form of an occam process with at least as many parallel sub-processes as there are hardware processors in the network. Within the description, each process which may be independently placed on a processor, is introduced by a PROCESSOR construct naming a processor. Processors so named may either be the hardware processors declared in the network description, or may be logical processors mapped onto the hardware processors in a separate mapping structure. In either case the processor name must have appeared in a NODE declaration in whose scope the software description is written.

The connections between processes in the software description are defined by occam channels. It is thus possible for the configurer tool to determine what code is to be loaded onto what processor, and to choose its own mapping of

channels onto physical connections between processors.

Some channels may be used to connect to hardware outside the network, such as the development host or other hardware connected by means of link adaptors. External objects of this kind are declared as EDGEs in the hardware description.

All processors which are connected together are connected via their links, represented in the language as attributes, of type EDGE of declared NODEs.

The connections to external edges, or those between processors may optionally be declared as ARCs, which associate a name with a particular connection. This enables explicit mappings of channels onto these arcs to be made.

5.2.1 Configuration language

A configuration description consists of a sequence of declarations and statements in an extension to occam and follows the usual occam scope rules. These declarations and statements are evaluated by the occam compiler, which is called during configuration by the configurer tool **occonf**. Appendix E (in part 2) defines the syntax of the occam configuration language and also gives details of how it differs from previous implementations of the toolset i.e. the IMS D705/D605/D505 products.

Configuration declarations introduce physical processors, arcs and edges of the network, network connections and processor attributes, logical processors to be mapped onto physical processors, the software description, and the mapping between logical and physical processors.

Arrays of NODEs, EDGEs, and ARCs may also be declared. A configuration description includes one NETWORK, one CONFIG and, optionally, one MAPPING. Each of the items appearing before CONFIG behaves as an occam specification, and ordinary VAL abbreviations may be included amongst these components to facilitate the description of scalable configurations. A NETWORK, CONFIG or MAPPING is optionally named by an identifier following its opening keyword.

Configuration declarations are usually followed by statements which perform various actions relating to the declaration. Actions are defined by SET, CONNECT and MAP statements. The DO construct enables these statements to be grouped or replicated. PROCESSOR statements introduce processes which may be mapped onto named processors.

The MAP statement may be replicated, via the DO construct, within a MAPPING declaration. SET and CONNECT statements may be used within a NETWORK declaration and may be combined in any order using the DO construct.

Declaration	Description
NODE	Introduces processors (*nodes* of a graph). These processors are considered to be *physical* if they are defined as part of the hardware description, or *logical* if they are defined as part of the software description and mapped to a physical processor as part of the mapping.
ARC	Introduces named connections (*arcs* of a graph) between processors (using the transputer links). These connections need not be declared as ARCs unless channels are required to be explicitly placed on particular links.
EDGE	Introduces external connections of the hardware description. External edges may be the host, or any peripheral connected via a link adaptor e.g. a joystick, disc drive.
NETWORK	Defines the connections and attribute settings of previously declared NODEs (physical processors).
MAPPING	Defines mappings between logical processors and physical processors.
CONFIG	Introduces the software description.

Table 5.1 Configuration description declarations

Statement	Description
SET	Defines values for NODE attributes.
CONNECT	Defines a connection between two EDGEs, either of two nodes or between a node and a declared external EDGE.
MAP	Defines the mapping of a logical processor onto a physical processor declared as a NODE.
PROCESSOR	Introduces a software process and associates it with a logical or physical processor.
DO	Groups one or more actions defined by SET, CONNECT or MAP statements.

Table 5.2 Configuration description statements

Code from other files may be referenced by means of the #USE directive, either at the top level, or within the CONFIG construct. #INCLUDE directives can be used to include other source files.

It is suggested that the distinct sections are kept in different files, accessed by #INCLUDE directives from a 'master' file.

5.2.2 Overall structure of a configuration description

A configuration description consists of two or three parts; a hardware network description, a software network description, and an optional mapping between the two.

The hardware description defines processor connections. It also defines attributes such as processor types and memory sizes. These processors are known as *physical* processors.

The software description is basically an occam parallel process, annotated with PROCESSOR statements to indicate which processes are to be compiled for which processors. These processes are allocated to *logical* processors.

The mapping section can be used to ease the task of changing a particular program to execute on a different hardware network. The mapping section enables this to be performed without modifying the software description in any way, by flexibly mapping the *logical* processors onto the *physical* processors. As an optimisation, for simple programs, or for programs which will never need to be re-mapped, the software description may reference the *physical* processors directly, avoiding the need to introduce *logical* processor names.

The following example illustrates the basic style of the language:

```
-- hardware description, omitting host connection
VAL K IS  1024 :        -- useful constants for memory
VAL M IS K * K :        -- sizes

NODE root.p, worker.p :     -- declare two processors
NETWORK  simple.network
  DO
    SET root.p (type, memsize := "T414", 1 * M)
    SET worker.p (type, memsize := "T800", 4 * M)
    CONNECT root.p[link][3] TO worker.p[link][0]
:
-- mapping
NODE root.l, worker.l :
MAPPING
  DO
    MAP root.l ONTO root.p
    MAP worker.l ONTO worker.p
:

-- software description
#INCLUDE "prots.inc" -- declare protocol
#USE "root.lku"      -- must be linked units
```

```
#USE "worker.lku"
CONFIG
  CHAN OF protocol root.to.worker, worker.to.root :
  PLACED PAR
    PROCESSOR root.l
      root.process(worker.to.root, root.to.worker)
    PROCESSOR worker.l
      worker.process(root.to.worker, worker.to.root)
:
```

Note that the configurer can, in this example, automatically place the channels onto the single connecting link, assuming that the two channels are used in different directions. The configurer can make this check by means of the normal occam usage checking rules.

This example is illustrated in figure 5.1.

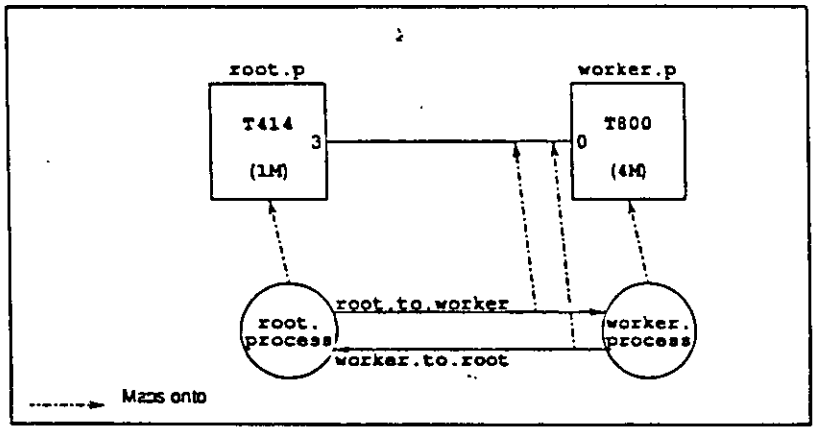


Figure 5.1 Mapping of software onto hardware

In a simple configuration such as this one where each physical processor is mapped onto a single logical processor, a shortened configuration description may be used which omits the mapping section altogether and uses the physical processor names directly in the software description.

To devise this shortened description remove the mapping section and delete the suffixes `.p` and `.1` from the NODE declarations, SET, CONNECT and PROCESSOR statements.

5.3 Hardware description

5.3.1 Declaring processors

Processors are declared to have NODE type, as if they were occam data items:

```
NODE worker :                        -- single processor
[No.of.workers]NODE pipeline : -- array of processors
```

5.3.2 NODE attributes

A NODE has a set of attributes, analogous to fields of a record. An attribute is referenced by subscripting the name of the node with the name of the attribute. The attributes are:

```
[]BYTE type :     -- String describing processor type,
                  -- see list below
[]EDGE link :     -- Link connections, number may
                  -- depend on type
INT memsize :     -- Memory size in BYTEs
BOOL root   :     -- Defines root processor if there is
                     no HOST connection
INT romsize :     -- Size of ROM attached to processor
order.code  :     -- Defines the priority of the program
                     code in memory
order.vs    :     -- Defines the priority of the
                     program's vectorspace in memory
```

The list of permissible attributes is in general dependent upon the NODE `type` field, and may be extended for other NODE types in the future.

The attribute names, which are predeclared by the configurer, do not follow the occam scope rules; they are only recognised in the correct context.

The use of `order.code` and `order.vs` is explained in section 5.5.3.

5.3.3 NETWORK description

The NETWORK keyword introduces a section which describes the connectivity, and attributes of previously declared NODEs. These should be declared outside of the NETWORK description, so that they are visible inside and below the NETWORK description.

To describe a single processor, the SET statement provides values for the pro-

cessor's tes in the style of a multiple assignment.

```
NETWORK single
  SET processor ( type, memsize := "T800", 1024*1024)
:
```

The `type` attribute must be set to a `BYTE` array (of any length) whose contents describe the processor type. Trailing spaces at the end of the processor's type are ignored.

Supported types are:

```
"T212"  "T222"  "T225"  "M212"
"T400"  "T414"  "T425"
"T800"  "T801"  "T805"
```

The `memsize` attribute must be set to the amount of usable memory attached to that processor, as a contiguous amount starting at the most negative address. It is specified in `BYTE`s.

Both the `type` and `memsize` attributes must be defined for all processors. No attribute may be defined more than once for each processor.

The above example could also be written as a sequence of `SET` statements in a `DO` construct:

```
NETWORK single
  DO
    SET processor ( type    := "T800")
    SET processor ( memsize := 1024*1024)
:
```

Since the `DO` construct does not imply any particular ordering, there is no constraint on the order in which attributes may be defined.

If a network is to be configured to be loaded from ROM, the attribute `root` must be set to `TRUE` for one processor only. By default this attribute is `FALSE` for all processors. The attribute `romsize` should be set to the number of bytes of ROM on the root processor. These attributes are ignored if the network is configured to be booted from link.

`IF`, `SKIP` and `STOP` may be used in `DO` constructs and are effectively executed at configuration time.

Processors must be connected together by means of `CONNECT` statements quot-

ing a pair of edges:

```
VAL K IS 1024:
NETWORK pair.from.ROM
  DO
    SET proc1 ( type, memsize := "T800", 2048 * K)
    SET proc1 ( root, romsize := TRUE, 256 * K)
    SET proc2 ( type, memsize := "T414", 1024 * K)
    CONNECT proc1[link][0] TO proc2[link][3]
:
```

The order of the two edges in a `CONNECT` statement is irrelevant.

Arrays of processors do not need to all have the same types or attributes. They can be set by using `DO` replicators within the `NETWORK` construct, and by using conditionals, as in this (rather contrived) example:

```
NETWORK pipe
  DO
    DO i = 0 FOR 100
      IF
        (i \ 4) = 0
          SET processor[i] (type, memsize := "T800",
                                4 * (1024 * 1024) )
        TRUE
          SET processor[i] (type, memsize := "T414",
                                2 * (1024 * 1024) )

    DO i = 0 FOR 99
      DO
        CONNECT processor[i][link][1] TO
                processor[i+1][link][0]
        IF
          (i \ 2) = 0
            CONNECT processor[i][link][2] TO
                    processor[i+2][link][3]
          TRUE
            SKIP
:
```

More complicated expressions may also be used, as long as they can be evaluated at configuration time:

```
VAL processors IS ["T414", "T414", "T414", "T800"] :
NETWORK fancy   -- every fourth processor is different!
  DO i = 0 FOR SIZE array
    SET array[i] ( type := processors[i \ 4] )
:
```

5.3.4 Declaring EDGEs

Declared EDGEs define the ends of external connections of a NETWORK. For instance, a connection to another machine whose internal structure is irrelevant. They are declared as though they were occam data types, and as usual we can declare arrays of them:

```
[10]EDGE diskdrive :
NETWORK disk.farm
  DO i = 0 FOR 10
    DO
      -- insert code to set attributes, then:
      CONNECT processor[i][link][0] TO diskdrive[i]
:

EDGE joystick :
NODE controller :
NETWORK n
  DO
    SET controller (type, memsize := "T212", 64 * 1024)
    CONNECT controller[link][2] TO joystick
:
```

5.3.5 Declaring ARCs

In some circumstances a programmer may require to name a connection between two processors. This isn't normally necessary, because the configurer can place channels between processors onto links automatically, but where a channel must be connected onto an external EDGE this is required. Also, if there are multiple links between two processors, and one link is set for some reason to go at a different data rate than another, the programmer might wish to have more control.

These named links are called ARCs, and are declared as though they were occam data types. They are associated with a link connection by adding a WITH clause to the end of a CONNECT statement.


```
EDGE joystick :
ARC  link.to.joystick :
NODE controller :
NETWORK n
  DO
    SET controller (type, memsize := "T212", 64 * 1024)
    CONNECT controller[link][2] TO joystick WITH
                                        link.to.joystick
:
```

5.3.6 Abbreviations

occam style abbreviations are permitted, to enable easier reference to elements of arrays, etc:

```
[10]NODE pipe :
NETWORK pipeline
  DO i = 0 FOR 10
    NODE this IS pipe[i] :
    SET  this (type, memsize := "T414", 1024*1024)
:
```

Since NODEs have an attribute `link`, whose type is `[]EDGE`, we can abbreviate one link of a processor as an EDGE:

```
[10]NODE pipe :
NETWORK pipeline
  DO
    DO i = 0 FOR 10
      SET pipe[i] (type, memsize := "T414", 1024*1024)
    DO i = 0 FOR 9
      EDGE this IS pipe[i  ][link][2] :
      EDGE that IS pipe[i+1][link][3] :
      CONNECT this TO that
:
```

Simple one-to-one mappings of logical to physical processors may also be expressed as abbreviations:

```
NODE root.1 IS root.p :
```

5.3.7 Host connection

There is a predefined EDGE named HOST, which indicates the connection to a host computer:

```
NODE single :
ARC  hostlink :
NETWORK B004
  DO
    SET single (type, memsize := "T800", 1000000)
    CONNECT single[link][0] TO HOST WITH hostlink
:
```

When configuring a program which is designed to be booted via a transputer link, one processor *must* be connected to the predefined EDGE HOST.

5.3.8 Examples of network descriptions

1) *Single processor configuration connected to host:*

```
NODE MyB004:
ARC hostlink:
NETWORK B004
  DO
    SET MyB004 (type, memsize := "T414", 2 * M)
    CONNECT MyB004[link][0] TO HOST WITH hostlink
:
```

This configuration is illustrated in figure 5.2.

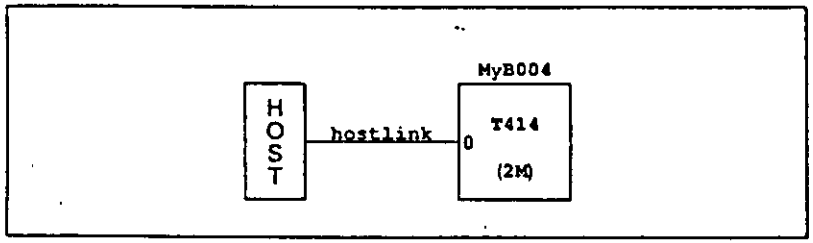


Figure 5.2 Example of host connection

2) *Simple pipe with one processor with different memory size:*

```
[p]NODE Pipe:
ARC hostLink:
NETWORK simple.pipe
  DO
    CONNECT HOST TO Pipe[0][link][0] WITH hostLink
    DO i = 0 FOR p-1
      CONNECT Pipe[i][link][2] TO Pipe[i+1][link][1]
    SET Pipe[0] (type, memory := "T800", 2*M)
    DO i = 1 FOR p
      SET Pipe[i] (type, memory := "T800", 1*M)
:
```

This network is illustrated in figure 5.3.

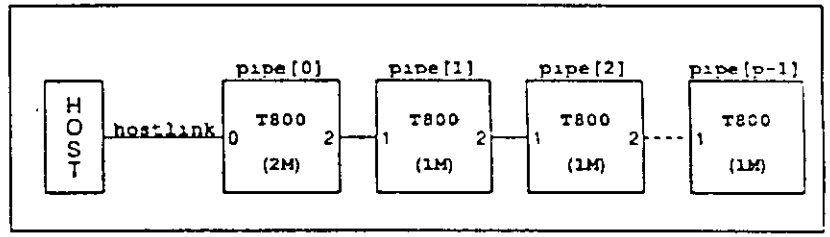


Figure 5.3 Simple pipeline with different processor memory sizes

3) *Square array with host interface processor:*

```
VAL Up IS 0:
VAL Left IS 1:
VAL Down IS 2:
VAL Right IS 3:
NODE HostSquare:
[p][p]NODE Square:
ARC hostlink:
NETWORK square
  DO
    SET HostSquare (type, memsize := "T414", 2*M)
    CONNECT HOST TO HostSquare[link][0] WITH hostlink
    CONNECT HostSquare[link][1] TO
            Square[p-1][p-1][link][Down]

    DO i = 0 for p
      DO j = 0 for p
        DO
          SET Square[i][j] (type, memsize := "T800", 1*M)
          IF
```

```
        (i = 0) AND (j = 0)
          CONNECT HostSquare  [link][Down] TO
                  Square[0][0][link][Up]
        i = 0
          CONNECT Square[p - 1][j - 1][link][Down] TO
                  Square[0    ][j    ][link][Up]
        TRUE
          CONNECT Square[i - 1][j][link][Down] TO
                  Square[i    ][j][link][Up]

DO i = 0 for p
  DO j = 0 for p
    IF
      j = (p-1)
        CONNECT Square[i][j][link][Right] TO
                Square[(i + 1)\p][0][link][Left]
      TRUE
        CONNECT Square[i][j][link][Right] TO
                Square[i][j + 1][link][Left]
```

5.4 Software description

The software description is an OCCAM process, PAR or PLACED PAR, with processes annotated by PROCESSOR statements. These identify which processes may be placed on particular processors. The keyword PLACED is retained for compatibility with earlier products; it is no longer required and has no effect.

The NODEs which are referenced by a PROCESSOR statement may be either *physical* processors if they are described as part of the hardware description, or *logical* processors if they are described as part of the software description. If the latter, they are mapped onto physical processors by means of a MAPPING section.

Physical processor names are allowed here to simplify small networks, or those which will not be re-mapped, so that the programmer does not need to invent two names for each processor.

The *logical* processor names must be introduced first by means of NODE declarations. These look identical to those used in the hardware description, but cannot have attribute settings. Since these must be visible to a following MAPPING section, they must be declared *outside* the CONFIG construct. Channels which are to be placed on ARCs by mapping statements must also be declared outside the CONFIG construct.

The process 'inside' the PROCESSOR statement may consist of OCCAM text.

However, it is recommended that the code should be restricted to simple procedure calls i.e. to separately compiled procedures, referenced as linked compilation units using the #USE directive. Code which generates library calls is not allowed.

A PROCESSOR statement associates the process instance (*process*) it labels with the logical or physical processor it names. The same name may be referenced in more than one PROCESSOR statement. The set of processes so named will run in parallel on that processor.

Note: when `imakef` is used to build the program, any linked units referenced by the software description must be given extensions of the type `cxx`. This is because `imakef` uses a different convention for file extensions to the normal TCOFF file extensions, see chapter 21.

5.4.1 Libraries of linked units

The facility to create libraries of linked units provides an easy method of targetting a process at different processor types within a software description.

For example, suppose a process is compiled and linked once for a T2 and once for a T8 and the linked units are given `imakef` file extensions in order to distinguish them. Referencing the two linked units directly within the software description by #USE directives, will cause one of them to hide the other from the configurer.

If, however, the linked units are used to create a library and this is referenced by a single #USE directive, the configurer will be able to extract the correct copy of the process for each PROCESSOR statement it finds.

Only libraries containing linked units may be referenced from within a software description.

5.4.2 Example

The following example of a software description, is for the pipeline sorter program introduced in chapter 4. The example is developed to show the complete configuration description for the program, in section 5.6. Figure 5.4 illustrates the mapping of the software processes onto a network of logical processors, which in this example is achieved without an actual mapping section. This method of mapping is explained in section 5.5.4.

```
#INCLUDE "hostio.inc"  -- declares SP
#INCLUDE "sorthdr.inc" -- declares LETTERS
```

```
#USE "inout.lku"          -- linked unit
#USE "element.lku"        -- linked unit
NODE inout.p :            -- logical processor
[string.length]NODE pipe.element.p : -- logical
                                     -- processors
CONFIG
  CHAN OF SP app.in:
  CHAN OF SP app.out:
  PLACE app.in, app.out ON hostlink:
  [string.length+1]CHAN OF LETTERS pipe:
  PAR
    PROCESSOR inout.p
      inout (app.in, app.out, pipe[string.length],
             pipe[0])
    PAR i = 0 FOR string.length
      PROCESSOR pipe.element.p[i]
        sort.element (pipe[i], pipe[i+1])
:
```

This example names a single processes `inout.p` and an array of processes `pipe.element.p`. The code may be mapped onto any hardware configuration onto which these logical processors may be mapped and which includes an ARC declaration for the host connection `hostlink`.

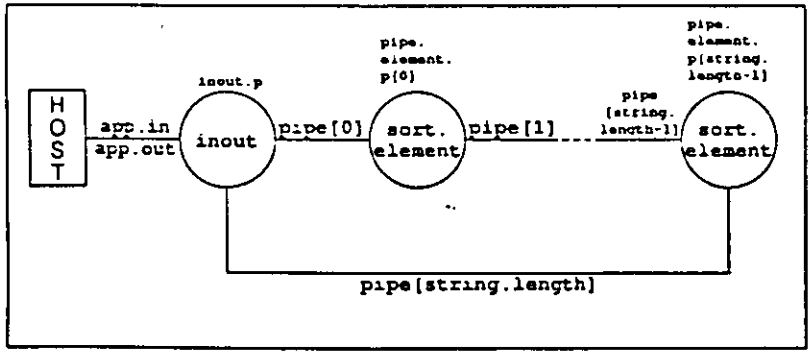


Figure 5.4 Pipeline sorter – mapping processes onto processors

5.5 Mapping descriptions

A `MAPPING` structure is used if the user has declared logical processors. The `MAPPING` maps logical processors used in the software description onto physical processors used in the hardware description. It is possible to map any number of logical processors onto any physical processor.

The priority at which a process runs may be determined as part of mapping, if that logical process does not explicitly include high priority code. This reflects the fact that changes in mapping may not affect the overall structure of the software, but can often change the decisions made about which processes should be prioritised.

`IF`, `SKIP` and `STOP` may be used in a mapping structure.

As would be expected from the occam scoping rules, logical processor names must be declared as `NODE`s in the software description, before the opening keyword `MAPPING` of the mapping description. Each name so declared must appear once and once only on the left hand side of a mapping item. Physical processors may appear on the right hand sides of multiple mapping items.

The mapping structure itself may appear either before or after the software description.

5.5.1 Mapping processes

Having declared *physical* processors, as part of the hardware description, and *logical* processors, as part of the software description, we can assign logical processors to physical processors using the `MAP` statement.

```
MAPPING map
  MAP logical.proc ONTO physical.proc
:
```

We can also supply a list of logical processors to all be mapped onto the same physical processor:

```
MAPPING map
  MAP router.proc, application.proc ONTO root.processor
:
```

This is exactly equivalent to:

```
MAPPING map
  DO
    MAP router.proc      ONTO root.processor
    MAP application.proc ONTO root.processor
:
```

And we can use `DO` replicators, and `IF` constructs, etc:

```
MAPPING map
```

```
  DO
    DO i = 0 FOR 10
      MAP router.proc[i] ONTO router.processor[i]
    DO i = 0 FOR 5
      MAP sieve.proc[i] ONTO sieve.processor
:
```

If we require that the process's priority be determined by the mapping, we can use the optional **PRI** clause. The argument to **PRI** can be either 0 to indicate *high* priority, or 1 to indicate *low* priority:

```
MAPPING map
  DO i = 0 FOR 10
    MAP logical.proc[i] ONTO physical.proc
                            PRI (INT (i = 0))
:
```

The configuration tool will reject the mapping at high priority of a process which itself includes a **PRI PAR**.

5.5.2 Mapping channels

Channels between processors need not be placed by the user. The configurer will determine that a connection exists, and will allocate all the channels to links if they are available. However, if a user wants to override the default allocation, channels may be mapped onto named ARCs. Also, channels connecting processors to external EDGEs must be mapped onto an ARC which connects to that EDGE.

Channels are mapped onto ARCs in exactly the same way as logical processors are mapped onto physical processors. Two channels may be mapped onto the same ARC, as long as they are used in different directions (the configurer will check this). Obviously the ARC must connect EDGEs of the processors onto which are mapped the processes which use the channel.

```
EDGE peripheral :
ARC  peripheral.arc :
NODE root.proc :
NETWORK n
  DO
    -- insert code to set attributes, then:
    CONNECT root.proc[link][0] TO peripheral WITH
                                  peripheral.arc
:
CHAN OF protocol to.periph, from.periph :
```



```
NODE process :
CONFIG
  PLACED PAR
    PROCESSOR process
      -- reads from channel from.periph, writes to
      -- channel to.periph
:

MAPPING
  DO
    MAP process ONTO root.proc
    MAP to.periph, from.periph ONTO peripheral.arc
:
```

5.5.3 Moving code and data areas

Two processor attributes may be used to provide greater control of the layout of code and data areas in memory. Note that changing the default ordering means that the INMOS debugger cannot be used with the program, and for this reason these attributes must be explicitly enabled on the command line by means of the 'RE' option.

Normally the configurer arranges for the program's workspace to be given the highest priority, and hence placed at the lowest address on chip. This means that the workspace can make best use of the transputer's on-chip RAM. Program code is treated with next priority, and vectorspace has the lowest priority.

These priorities can be overridden by setting two processor attributes 'order.code' and 'order.vs', which correspond to the program code, and to the program's vectorspace, respectively. These can be set to INT values, where lower integers indicate a higher priority. The workspace is given priority 0. Hence setting 'order.code' to -1 means that the code will be placed at a lower address than the workspace. If an attribute is not set, the priority is considered to have value 0. The relative ordering of sections whose priorities are equal is undefined.

Since these attributes are essentially properties of the user's program, not of the hardware description, the settings must be made as part of the MAPPING section. However, the processor which is referenced must be a physical processor.

Thus we may a mapping section like so:

```
MAPPING prioritise.code
  DO
    SET physical.processor (order.code := -1)
    MAP logical.processor ONTO physical.processor
:
```

If code re-ordering has not been explicitly enabled by the command line option 'RE', these attributes will be ignored.

5.5.4 Mapping without a MAPPING section

Without a mapping section a channel allocation may be used instead of a channel mapping.

Any channel in scope at the point where a process is labelled is available for explicit *placement* on an arc declared in the hardware network. This is done by adding the following *allocation* immediately after the declaration of the channel:

```
CHAN OF protocol to.periph, from.periph :
PLACE to.periph, from.periph ON peripheral.arc :
CONFIG
  PLACED PAR
    PROCESSOR root.proc
      -- as before
:
```

Allowing more than one channel to be placed in a single allocation or mapping statement allows the two channels on any one physical transputer link to be placed in a single line of code.

5.5.5 Mapping examples

1) *pipeline sorter on a single processor*

```
MAPPING
  DO
    MAP inout.p ONTO MyB004
    DO i = 0 FOR string.length
      MAP pipe.element.p[i] ONTO MyB004
:
```


2) *pipeline sorter on a ring of processors, one per process*

```
MAPPING
  DO
    MAP inout.p ONTO MyB004
    DO i = 0 FOR string.length
      MAP pipe.element.p[i] ONTO ring[i]
:
```

5.6 Example: A pipeline sorter on four transputers

This section describes how the pipeline sorter program, described in section 4.12, may be distributed over four T414 transputers. Each processor has many processes allocated to it.

An example of how to design and write a configuration description is given, followed by detailed instructions about how to compile, configure and run the program.

In the configuration description it is assumed that there is a transputer network of four T414 transputers connected as shown in figure 5.5. It does not matter if you don't have such a network – you should read through this example and then try modifying it for your network.

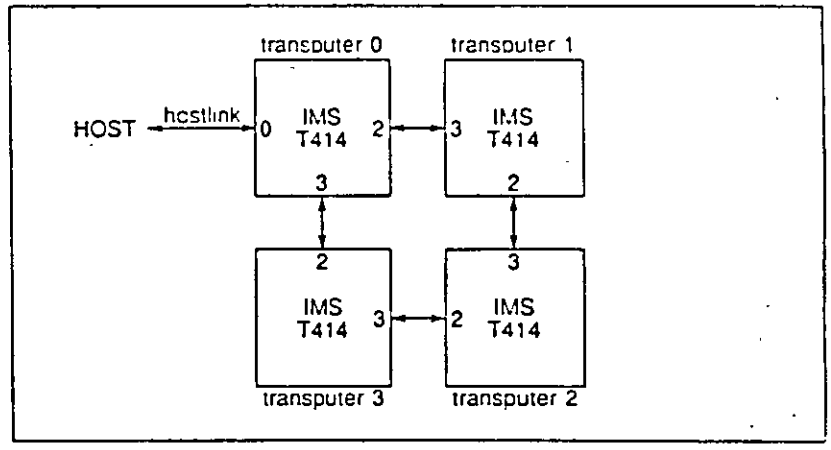


Figure 5.5 Network of four transputers

The occam source and configuration description developed in this example is supplied with the toolset in the "examples" directory, and you should copy these files to a working directory in order to build the program. Alternatively you can

type in the source of the program, as it is given below and in section 4.12.

The files are as follows:

`sorthdr.inc`	the common protocol definition.
`element.occ`	the sorting element.
`inout.occ`	the interface to the host file server.
`sortb3.pgm`	the configuration description for the network.

The contents of the files `sorthdr.inc`, `element.occ` and `inout.occ` are described in section 4.12. The contents of the other files used in the program are described below.

To complete the program the host file server library `hostio.lib`, the hostio include file `hostio.inc`, and the compiler library code will be used from the toolset library directory.

The following code is in the file `sortb3.pgm`, it describes the hardware network shown above and a mapping of processes onto this network which puts an equal number of processes on all processors after the first one, which also gets any remainder:

```
-- problem size
VAL string.length IS 80:

-- hardware description
VAL number.of.transputers IS 4:
VAL number.of.elements IS string.length:
VAL elements.per.transputer IS number.of.elements/
                               number.of.transputers:
VAL remaining.elements IS number.of.elements\
                          number.of.transputers:
VAL elements.on.root IS elements.per.transputer +
                        remaining.elements:

VAL K IS 1024:
[4]NODE B003.t:
ARC hostlink:
NETWORK
  DO
    CONNECT B003.t[0][link][0] TO HOST WITH hostlink
    DO i = 0 FOR 4
      DO
        SET B003.t[i] (type, memsize := "T414", 256*K)
        CONNECT B003.t[i][link][2] TO
```



```
                B003.t[(i+1)\4][link][3]
:
-- mapping
VAL HIGH IS 0:  -- priorities
VAL LOW IS 1:
NODE inout.p:
[number.of.elements]NODE pipe.element.p:
MAPPING
  DO
    MAP inout.p,
        pipe.element.p[elements.on.root-1] ONTO
        B003.t[0] PRI HIGH
    DO i = 0 FOR elements.on.root-1
      MAP pipe.element.p[i] ONTO B003.t[0] PRI LOW
    DO j = 0 FOR number.of.transputers - 1
      VAL first.element.here IS elements.on.root +
          (j*elements.per.transputer):
      VAL last.element.here IS first.element.here +
          (elements.per.transputer-1):
      DO
        MAP pipe.element.p[first.element.here],
            pipe.element.p[last.element.here] ONTO
            B003.t[j+1] PRI HIGH
        DO i = first.element.here + 1 FOR
                  elements.per.transputer - 2
          MAP pipe.element.p[i] ONTO
              B003.t[j+1] PRI LOW
:
#INCLUDE "hostio.inc"
#INCLUDE "sorthdr.inc"
#USE "inout.lku"
#USE "element.lku"
CONFIG
  CHAN OF SP app.in:
  CHAN OF SP app.out:
  PLACE app.in, app.out ON hostlink:
  [string.length+1]CHAN OF LETTERS pipe:
  PAR
    PROCESSOR inout.p
      inout (app.in, app.out, pipe[string.length],
             pipe[0])
    PAR i = 0 FOR string.length
      PROCESSOR pipe.element.p[i]
        sort.element (pipe[i], pipe[i+1])
:
```

In the mapping structure shown, the logical processors named in the software description are mapped onto the physical processors declared in the hardware description. Note: that on each processor, processes which communicate on external channels are mapped to be run at high priority. The allocation of processes to transputers is shown in figure 5.6.

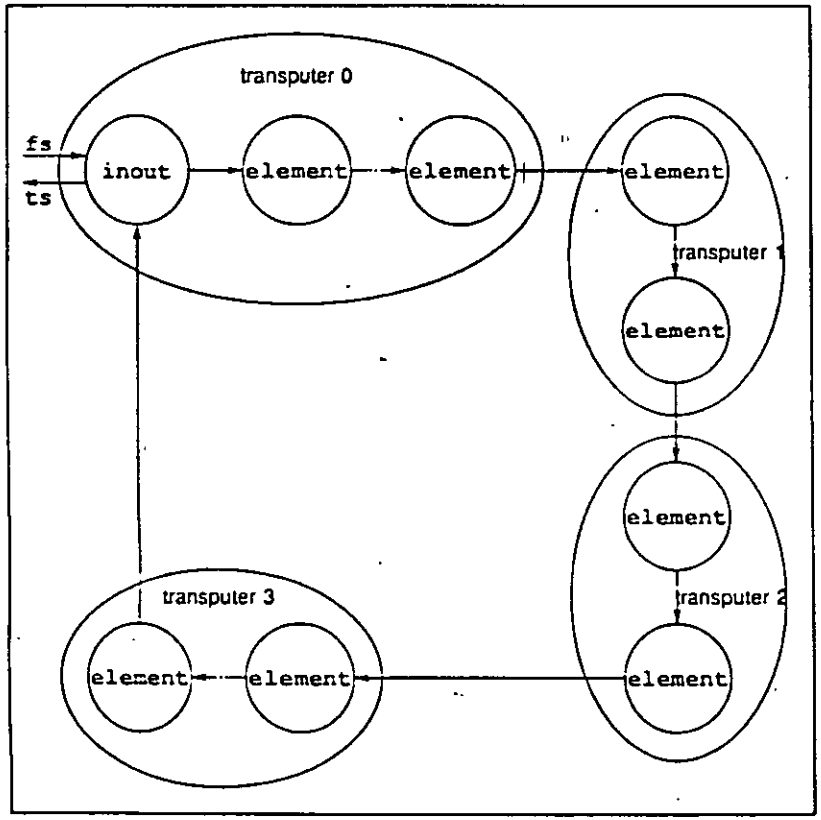


Figure 5.6 Pipeline sorter processes

5.6.1 Building the program

The components of the program must be compiled in a bottom up fashion. First compile the sorting element using the following command:

```
oc element
```


Because the file has a `.occ` file extension you can omit the extension from the filename. The command line options to specify the target processor and error mode may also be omitted because the defaults are required i.e. `T414` and HALT mode. The compiler will produce a file called `element.tco`.

Next compile the input/output process using the following command:

```
oc inout  (creates the file inout.tco)
```

Each of these files must now be linked. The files are linked in separate operations, together with any files they reference. Each linking operation creates a unit of code which may be loaded onto the transputer network, according to configuration defined in the configuration description.

To link `element.tco` use one of the following commands:

```
ilink element.tco -f occama.lnk          (UNIX)
ilink element.tco /f occama.lnk          (MS-DOS/VMS)
```

Both of these commands will create a file called `element.lku` The linker indirect file `occama.lnk` contains the necessary references to the compiler libraries. This file is supplied with the toolset.

To link `inout.tco` use one of the following commands:

```
ilink inout.tco hostio.lib -f occama.lnk     (UNIX)
ilink inout.tco hostio.lib /f occama.lnk     (MS-DOS/VMS)
```

Both of these commands will create a file called `inout.lku`.

Now configure the file `sortb3.pgm` which defines both the communication channels between the processes and how they should be loaded onto the network:

```
occonf sortb3.pgm
```

This command will create an output file called `sortb3.cfb`

To make the program runnable you must add bootstrap code. To do this use the collector tool `icollect`:

```
icollect sortb3.cfb
```

The collector will create the file `sortb3.btl`

5.6.2 Running the program

The program in the file `sortb3.btl` may be loaded and run using the skip loader from the host via the root transputer which is assumed to be connected by its link 2 to link 0 of the first transputer of the IMS B003 external network.

One of the following command sequences should be used:

UNIX based toolsets:

```
iskip 2 -e -r
iserver -se -ss -sc sortb3.btl
```

MS-DOS and VMS based toolsets:

```
iskip 2 /e /r
iserver /se /ss /sc sortb3.btl
```

To run the program on the transputer network which includes the root transputer, use one of the following commands:

```
iserver -se -sb sortb3.btl          (UNIX)
iserver /se /sb sortb3.btl          (MS-DOS/VMS)
```

The program will run until you type 'RETURN' on its own. The '`se`' option directs the server to terminate if the program sets the error flag.

5.6.3 Automated program building

As with the single processor version of this program it is possible to automate the building of this program with the Makefile generator tool and a suitable MAKE program. The version of the configuration program supplied in the file `sortb3c.pgm` is written using `imakef` file naming conventions. For example, the linked units are given file extensions of the form *cxx*.

To produce a Makefile for the entire program type:

```
imakef sortb3c.btl
```

The Makefile generator will produce a file called `sortb3c.mak` containing a MAKE description for the program. It will also produce linker indirect files for the two compiled units which comprise the program; these will refer to any necessary modules from the library.

To build the program run the MAKE program on the file `sortb3c.mak` and

all the necessary compiling, linking and configuration will be done automatically. For more information about MAKE programs see chapter 21.

5.7 Use of conditionals in a configuration

Conditional constructs (`IF`) are permitted inside `NETWORK`, `MAPPING` and `CONFIG` constructs. This makes it possible to create configuration descriptions which can be 'conditionally compiled' for different network structures.

For example, while developing a program, it may be useful to modify a program to bypass the root processor, so that an application may be placed directly onto an application processor. The following, rather trivial, example demonstrates this:

5.7.1 Example: Configuration using conditional `IF`

In this example, when a single processor is in use, the application communicates directly with the host, as shown in figure 5.7. When two processors are available, a buffer process is loaded onto the root processor. This process buffers the communication between the application and the host. See figure 5.8.

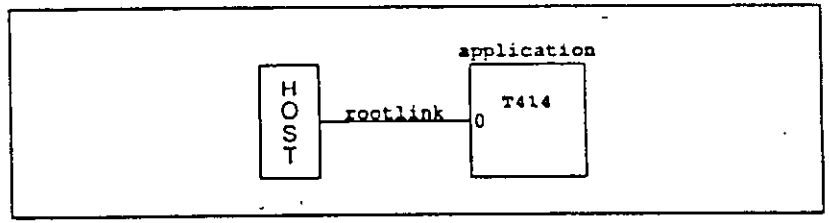


Figure 5.7 Direct host connection

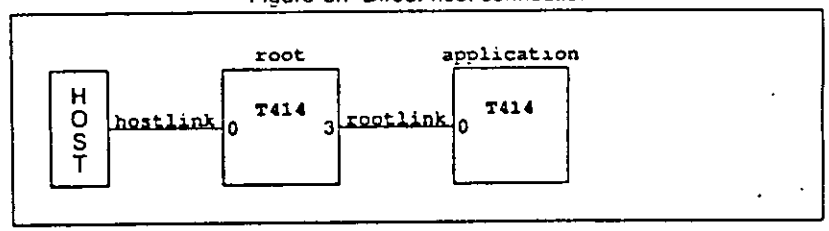


Figure 5.8 Communication via the root processor

The implementation is split into the following files:

`app.occ` – the application
`buff.occ` – the buffer process

myprog.pg he configuration description file

ie content of app.occ is as follows:

```
NCLUDE "hostio.inc"
SE "hostio.lib"

OC application.process(CHAN OF SP fs, ts)
SEQ
  so.write.string.nl(fs, ts, "Hello world")
  so.exit           (fs, ts, sps.success)
```

ie content of buff.occ is as follows:

```
NCLUDE "hostio.inc"
SE "hostio.lib"

OC buffer.process(CHAN OF SP fs, ts, from.app, to.app)
CHAN OF BOOL stopper :
-- This never terminates
so.buffer(fs, ts, from.app, to.app, stopper)
```

ie content of myprog.pgm is as follows:

```
L number.of.processors IS 1 :  -- 1 when running,
                               -- 2 for developing
DE root, application :
C hostlink, rootlink :
TWORK
DO
  IF
    number.of.processors = 2
      DO
        SET root (type, memsize := "T414", #100000)
        CONNECT root[link][0] TO HOST WITH hostlink
        CONNECT root[link][3] TO application[link][0]
                              WITH rootlink
    TRUE
      CONNECT application[link][0] TO HOST WITH rootlink
  SET application(type, memsize := "T414", #100000)

NCLUDE "hostio.inc"
SE "app.cah"
SE "buff.cah"
NFIG
CHAN OF SP fs, ts :
```



```
  PLACE fs, ts ON rootlink : -- Note that this is 'rootlink'
                             -- not 'hostlink'
  PAR
    IF
      number.of.processors = 2
        CHAN OF SP fs0, ts0 :
        PLACE fs0, ts0 ON hostlink :
        PROCESSOR root
          buffer.process(fs0, ts0, ts, fs)
      TRUE
        SKIP
    PROCESSOR application
      application.process(fs, ts) .
:
```

NODEs which are declared, but do not have any attributes set, are ignored when configuring a program.

5.8 Summary of configuration steps

To summarise, the steps involved in building a program that runs on a network of transputers are as follows:

1 Decide how your program will be distributed over the transputers in your network.

2 Write a configuration description for your program by:

(a) Describing your hardware network.

(b) Inserting PROCESSOR statements into your program and adding any necessary mapping description.

3 Compile all the separate compilation procedures that form the code for each transputer in a bottom up fashion.

4 Link each configuration procedure with its component parts into a file with the name used in #USE directives in the configuration source file.

5 Run the configurer on the configuration description file.

6 Collect the code using `icollect`.

7 Load the program into the network using the host file server.

Steps 3 to 6 can be automated by using `imakef` and a suitable MAKE program.

7

Aplicaciones

Transputer Applications
M. Jane et al., Eds.
IOS Press, 1992

REAL-TIME CONTROL APPLICATIONS OF TRANSPUTERS

George W. Irwin[1] and Peter J. Fleming[2]
[1]Electrical and Electronic Engineering, The Queen's University of Belfast, Belfast. U.K.
[2]Automatic Control and Systems Engineering, University of Sheffield, Sheffield. U.K

Abstract. The exploitation of the transputer and occam in real-time automatic control engineering is described. Following consideration of the special nature of this application domain, hardware and software features relevant to real-time implementation are discussed. Mapping strategies adopted by control engineers, are examined together with a detailed consideration of pertinent granularity issues. The paper concludes with a review of some successful applications chosen from robotics, aerospace, Kalman filtering and instrumentation.

1. Introduction

Control engineers use transputer-based systems to increase the execution speed of control software, to implement more sophisticated control laws and to exploit the potential of parallel systems to realise fault-tolerant implementations. They also are attracted to parallel programming languages and occam, in particular, as a means of directly expressing concurrency arising in real-time control. While transputer-based systems are also being used by control engineers for off-line applications, such as computer-aided control system design [1] and training neural networks [2], this paper deals only with on-line, and therefore real-time, applications.

Control techniques are applied right across the spectrum of engineering, including electrical, aeronautical, chemical, mechanical, environmental and medical. Consequently the variety and scale of examples of control systems are equally wide, e.g. fly-by-wire aircraft, power station boilers, robots, greenhouse environments, electricity generators, domestic heating systems, insulin pumps and video recorders.

Typically, a control system must function at a number of levels. Loop control generally involves the use of negative feedback, perhaps with dynamic compensation to produce a fast, accurate, well-damped response which is resistant to external disturbances and robust to changes in the system being controlled. Sequential control is concerned with producing the sequence of operations which a system should perform, such as the timed program on a domestic washing machine. Supervisory control ensures that the overall or

global objective of the control system is being achieved. The need for parallel processing has arisen in both loop control and supervisory control applications.

In real-time digital control, amongst other functions, the calculation of controller output must be performed within a loop sample interval of typically 5-20 ms, on the basis of periodically sampled measurements. Despite the increasing computing power of conventional sequential processors, this can be difficult to realise in a growing number of cases. A modern real-time control system might typically involve algorithms for control, simulation, optimisation, filtering and identification, in addition to simpler practical tasks such as event logging and data checking. Clearly, the more complex the algorithm, the more difficult the problem of performing the necessary calculations in real-time and it is arguable that the pull-through of advanced control theory into industrial applications is often hampered by the amount of computation required. In addition, different applications place varying demands on a real-time controller. For example, the sample intervals for the control of electric motors will be short because of the small time constants involved. Multivariable systems will add complexity to the control calculations since several control signals must be calculated simultaneously and, in aerospace and nuclear control applications, reliability is essential. Techniques such as expert systems and artificial intelligence are finding increasing application in control, as in "jacketing" software for adaptive controllers.

The transputer has been readily accepted by the control engineering community as the most suitable computing element available for embedded parallel processing systems. This paper describes how control engineers have exploited its special features. The special nature of the real-time control problem is outlined in the next Section. In Section 3, hardware and software features of the transputer and occam, relevant to real-time implementation are discussed. Section 4 describes hardware and software mapping strategies adopted by control engineers when using transputers and Section 5 looks in some detail at granularity issues which are highly pertinent to this application domain. The paper concludes with a review of some selected successful applications ranging over such areas as robotics, aerospace, Kalman filtering and instrumentation.

2. Parallel Processing and the Real-Time Control Problem

It was pointed out in the Introduction that implementation of parallel processing for real-time control involves mapping tasks onto a parallel system which must compute the control functions within a critically short time window. This is illustrated in Fig.1, where filtering and control calculations are computed first, secondary control functions processed next, and, if there is sufficient processing time available, it is mopped up by housekeeping tasks. The important issue to note here is the scale of operations concerned when compared with the more usual supercomputing applications of parallel processing. The modest level of task complexity and critically short time intervals demand special attention to implementation on MIMD machines.

Due to their flexibility and ability to operate on unstructured and unpredictable operations and data, MIMD machines, such as the transputer, can deal with a wider range of problems than other classes of parallel processing system. However, the designer must play a direct and fundamental rôle in successfully extracting the potential parallelism of the problem and in evaluating the trade-offs involved. Performance benefits strongly depend on the compute/communicate ratio [3]. This ratio expresses how much communication overhead is associated with each computation and clearly a high compute/communicate

ratio is desirable. The concept of task granularity can be also viewed in terms of compute time per task: when large, the task implementation is coarse-grain; when small, it is fine-grain. Although large grains may ignore potential parallelism, partitioning a problem into the finest possible granularity does not necessarily lead to the fastest solution, as maximum parallelism also incurs the maximum overhead, particularly due to increased communication requirements. Because of the special nature of their problem domain, control engineers are acutely aware of these widely appreciated tenets of parallel processing and strive to match the granularity of their application to the hardware granularity of the transputer.

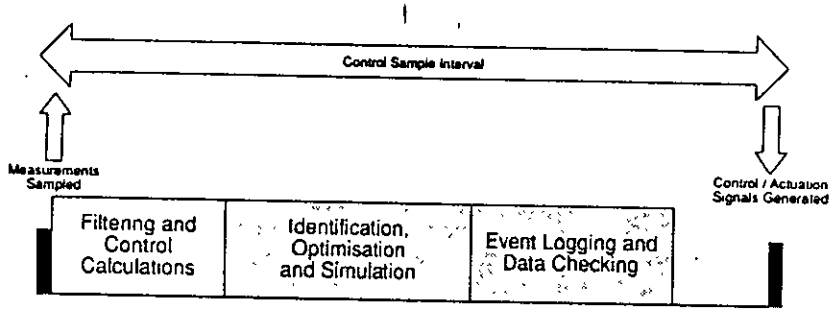


Fig.1 Real-time control computation window

3. Transputer Implementation of Real-Time Control

Control engineers view the transputer as a powerful processing element capable of easy implementation in an embedded parallel processing system. Its primary language, occam, is sufficiently high-level to permit clear expression of parallel software structures while retaining the efficiency of a low-level language. The facility whereby parallel software can be developed on a single processor and subsequently mapped onto a multiprocessor system is appealing for embedded system developers.

To a certain extent, occam on the transputer removes the need for a real-time operating system. It has a timer, manages I/O and communication between processes using the CSP model [4], handles interrupts on an event channel, provides constructs for handling concurrency and offers one high-priority and one low-priority operating mode.

Another important difference between the transputer and other microprocessors is its built-in scheduler which can schedule a process in one of its two priority modes. However, having only two priority levels can prove restrictive for real-time control application. Bakkers et al. [5] suggest that the sampling process of a control system may be divided into three categories:

Time-bounded processes, Time-limited processes, and Background and alarm processes.

Time-bounded processes include sampling and actuation operations and must be scheduled at specific instants. Time-limited processes are actions which should be scheduled to meet their deadline such as control signal calculations. Background and alarm processes involve

operations of varying levels of priority such as alarms, condition monitoring, optimisation and event logging.

A breakdown of these categories suggests a variety of levels of priority ranging from high priority, pre-emptive to a low level implemented by the transputer's low-priority, round-robin scheduler. These are not accommodated within the basic transputer-occam combination and a number of real-time kernels have been developed such as Trans-RTXc [6]. In an attempt to retain efficiency, Welch [7] has proposed two schemes, implemented in occam with an acceptably low level of overhead, to realise multiple, fixed priority or multiple, dynamic priority scheduling on transputers.

The interrupt handling facility available on the basic transputer-occam combination is too rudimentary for many applications and it is often necessary to extend this feature. The provision of four links on the transputer immediately releases the control engineer from the restrictions of bus-based systems but, in its turn, generates a demand for even more links per processor. (Recently, David May of Inmos has identified processor power, communications speed and connectivity as the three main concerns of parallel system developers. He suggests that processor power and communications speed are being effectively addressed by VLSI designers but that connectivity requires special attention).

Typical transputer networks are shown in Fig.2. Topologies used in control are, in general, small scale and, typically, conform to the 2-D array model. Pipelines are used

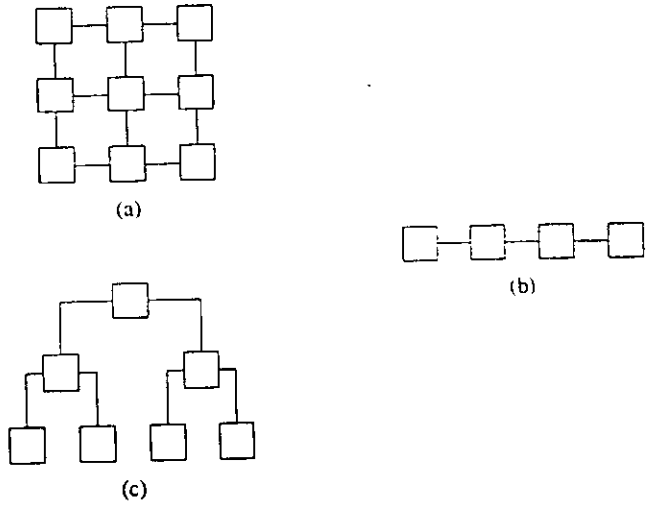


Fig.2 Transputer networks: (a) 2-D array; (b) pipeline; (c) tree.

occasionally but attention must be paid to the associated latency in processing time. The tree topology has received some attention in the context of processor farms (see Fig.3). Static task allocation is used more readily than dynamic task allocation because of its lower run-time overhead. However, dynamic task allocation has been used in conjunction with processor farms in an attempt to gain flexibility in processing variable task sizes and also to develop a generic system architecture for real-time control.

Processor farm architectures consist of a "farm" of processors receiving instructions from, and reporting back to, a controller. Each processor runs the same program (with data dependent branches) and has a complete, but different, set of data. There is limited communication between processors, although memory costs may be significant because large amounts of storage are required on each processor. Dynamic task allocation is required in this approach. It has been found that there is a significant performance penalty associated with scheduling software overheads and increased communication requirements which can prove unacceptable in some real-time applications.

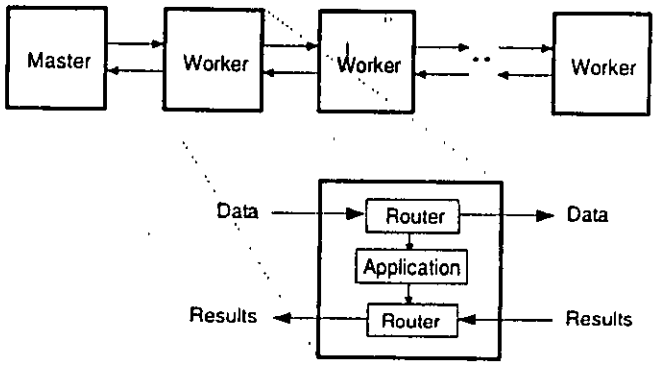


Fig.3 Processor farm arrangement.

Jones and co-workers have investigated this at length and explored ways to reduce the attendant overheads associated with the approach (e.g. [8],[9]). These include task buffering and more fully exploiting the four-link communication provision by mapping the processor farm concept onto a tree-like structure, thereby reducing communication bottlenecks. Dynamic task allocation, of course, affords opportunities for the creation of fault-tolerant systems.

The call for increased numbers of communication links per processor has, perhaps, been most insistent from control engineers striving to design high-reliability systems. For example, Thompson and Fleming [10] make this point in their work on operationally fault-tolerant, transputer-based architectures suitable for gas turbine engine control. The replacement of the fixed four-link architecture in the new T9000 development with multiplexing and "virtual channels" is one of the most eagerly awaited features of this new processor for control engineers.

4. Mapping Strategies

In common with other application domains, there is no standard procedure for converting sequential software into parallel software to run on transputer systems. One attempt to develop a generic approach, based on the processor farm, has already been described above. Another, [11], which exploits the regular nature of many linear control algorithms is dealt with in the next Section. Shaffer [12] describes a method

for converting sequential Fortran code for turbojet engine control to C code to be implemented on a Motorola-based multiprocessor system.

In general, the system architecture should be matched to the control software. Bakkers and van Amerongen [13] demonstrate how parallelism may be extracted from a robotic control problem by identifying various layers such as an interface layer, a safety protection layer, a control law calculation layer, a supervisory layer, etc. This approach is also appropriate for application areas other than robotics.

Many implementations use more than one parallelisation strategy to realise a successful system. *Functional parallelism* is a commonly applied method (e.g. [14]) in which individual tasks are associated with the different control functions to be implemented. Sequential and parallel operations must be identified along with their interdependencies. This approach is susceptible to load balancing problems where unequal task sizes lead to an uneven distribution of computational load across the individual processors in the system.

Some problems have an inherent regular geometrical structure which can be exploited to extract parallelism. This regularity allows data to be distributed uniformly across the processor array, each processor being responsible for a defined spatial area or volume. Neighbouring processors will have to exchange data at intervals. In [15], Ponton and McKinnel provide a good example of *geometric parallelism* and its use in control.

Algorithmic parallelism arises when each processor implements a part of the total control algorithm. In such a decomposition, data now flows between the processing elements, increasing the communication load on each element in the array and communication overheads can severely degrade performance unless care is taken. However, an advantage of this approach is that very little data space is required on each processor. Many control algorithms are regular in nature, e.g. the digital version of the state-space dynamic compensator

$$z(k+1) = Gz(k) + Hy(k)$$
$$u(k) = Cz(k) + Dy(k)$$

(where, u, z and y represent the control, compensator state and measurement vectors respectively) and a variety of techniques have been developed to generate parallel realisations. Two classes of method are appropriate for transputer-based systems: state-space based methods and systolic algorithms. An example of the former class is described in the next Section.

Systolic arrays provide a fine-grained description of the concurrent computation involved in regular, matrix-based control algorithms [16], [17]. Originally intended for implementation on fine-grain, dataflow architectures, Irwin and co-workers [18],[19] have evolved a method for aggregating systolic cells suitable for implementation on the medium-grain architecture of the transputer. First, a systolic algorithm is devised to implement a solution to a particular problem. This will be realised by a number of cells, each with a simple processing rôle, acting in parallel. While a large number of cells may be involved, linked in a regular fashion, only two or three types of cell will be used. This provides a parallel framework which may be suitably grouped and cast onto a transputer network. This establishes a generalised mapping technique which has been successfully applied to Kalman filtering (see Section 6 on Applications).

5. Performance Issues

In view of the importance of matching task granularity to processor granularity to obtain efficient performance, Garcia Nocetti and Fleming [11] have developed an on-line performance analysis environment, EPICAS (Environment for Parallel Implementation of Control Algorithms and Simulation). The *de facto* standard control system design package, MATLAB [20], has been integrated with the Transputer Development System (TDS) (see Fig.4). This environment offers the control engineer a number of software tools for automating the implementation of control algorithms and simulation of systems on transputer-based architectures. The tools are used to map systems onto transputer architectures of different sizes and topologies, and to evaluate strategies by displaying, on-line, task allocation, processor activity and execution time data.
Operating on a control law entered into MATLAB, a solution based on a parallel state-space strategy maximises the parallelism available by modifying the algorithm into a set

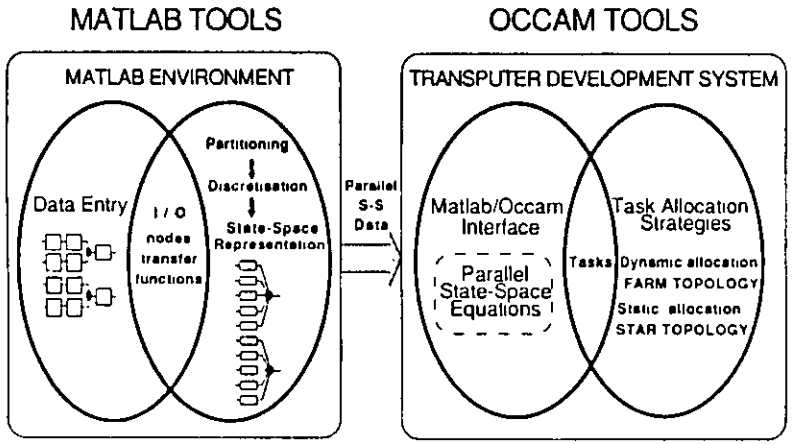


Fig. 4 EPICAS - Environment for Parallel Implementation of Control Algorithms and Simulation

of independent tasks, using a state-space model of the system. The input-output relationship for each input-output pair is obtained and expressed in block-diagonal, state-space form. This is reduced to a set of low-order, state-space equations which may be executed concurrently.

This parallel control law description is transferred into TDS where a toolset, written in occam 2, is used to automate the mapping of control algorithms onto a number of transputer topologies, using both static and dynamic task allocation strategies. Actual execution times, processor activity and task allocation are monitored on-line, enabling direct assessment of various strategies and topologies.

In [11] there is an extensive study of granularity issues with respect to T4 and T8 versions of transputers, where it is clearly established that, unless task sizes are sufficiently large, increasing numbers of transputers affords little improvement. Indeed, a processor

farm example shows that, for certain sizes of system, an increase in the number of processors actually leads to a deterioration in performance.

Maguire [21] has extended the compute/communicate ratio, R/C, where

$$\frac{R}{C} = \frac{\text{length of run-time quantum (secs)}}{\text{length of communications overhead produced by quantum (secs)}}$$

to include information relevant to both the algorithm and the hardware:

$$\frac{R}{C} = \frac{\dfrac{\text{task computational requirements (no. of operations)}}{\text{communication requirements of task (no. of bits)}}}{\dfrac{\text{processor computational performance (flops)}}{\text{processor communication performance (bits/sec)}}}$$

This is a valuable move to express task granularity with respect to processor granularity. He proceeds to obtain the following measures of processor granularity:

T414	0.005
T800	0.075
T9000	0.3125 (estimated).

These figures support the observations [11] that larger task sizes are required to realise efficient performance on T8 series processors and serves to indicate the mapping changes which must be anticipated for the T9 series.

Of course, such measures do not convey the whole story. For example, inter-transputer communication is more efficient when longer messages are sent rather than a large number of small messages. Having introduced, then, the notions of task granularity and processor granularity it is further worth investigating "communications granularity" [22] where communications speed and the handling of message size varies from processor to processor.

6. Applications

This section describes the work done on transputers in a number of important application areas. It will, of necessity, be brief and the reader is referred to the references for more detail.

6.1 Robotics

Robot control functions range through a number of levels: from decision making, path planning, and coordination at the top level to joint angle control at the bottom level. The computational requirements of high speed, high bandwidth, adaptive systems makes this area ripe for the exploitation of transputer technology.

Mirab and Gawthrop [23] review work on the application of parallel processing for calculation of the dynamic equations of robotic manipulators. Jones and co-workers [8],[9] report on the use of transputers in computing the Newton-Euler formulation of the

inverse dynamics. They suggest a granularity mismatch, this particular problem being more suited to a fine-grain architecture. Their work is further aggravated by the use of a processor farm topology for which they propose hardware improvements.

Daniel and Sharkey [24] draw attention specifically to the heavy computational demands of force control and to the need for a controller which is capable of switching easily between different layers in the control hierarchy. It is argued that latency is a key factor in the determination of a suitable transputer-based system architecture and they advocate their 'Virtual Bus' solution.

Hardware and software issues for real-time control of a fast assembly robot are under investigation at Twente University [25]. The proposed control system is made up of several layers. The first consists of the interface hardware and its matching real-time sampling software. The second layer contains the protection software which is essential for maintenance and fault diagnosis of the system during operation. The third layer contains the control algorithm and the necessary communication software for communicating control variables between different parts of the control algorithms.

Arising from this work several new hardware and software components have been developed. These include the LINX backplane, based on specifications for the VME standard with 8 slots for transputer boards and two linkswitch boards, standard transputer sections for use on all I/O boards called TRAS and a real-time language TASC (Transputer Application generator for Sampling applications in Control systems) which is an application generator allowing the control engineer to specify the sampling programme in a high level language. It permits independent control over all aspects of control processes such as timing, communication, control of interface hardware and error handling.

The transputer is employed as the prime architecture in a large mobile robot programme at Oxford University [26] largely because it adapts so easily to a distributed, reconfigurable sensing and control strategy in addition to providing powerful onboard processing capability. A transputer based sensor with local intelligence forms the basis for networks of distributed sensors together with an integrated path for combining the information from different sensors. For control, a distributed architecture overcomes the bottlenecks associated with a centralised controller.

Japanese researchers are also showing an increasing interest in transputers. The Electrotechnical Laboratory, Tsukuba Science City is studying the use of the transputer as a unified processing base for operations ranging from sensing and actuation [27] across to intelligence and manipulation skills. The aim is to produce a virtual environment for teleoperation tasks [28]. Transputers are also being used in force and precision controllers [29] and multiple, mobile robot applications. At Tokyo University, four robots, with on-board embedded transputer processing, are used to manipulate objects cooperatively, with planning and supervisory control being carried out by a central network of transputers.

6.2. Motors and Generators

Variable speed drives are used in applications like machine tool drives, traction, paper and steel mill rollers where rapid changes in torque are required. Although the induction motor has practical advantages over the more commonly used d.c. motor, it requires a more complex control structure since the voltage, current, torque and speed are all interdependent, leading to a highly coupled, nonlinear control problem. The computing load demanded by ac machines has prohibited their application in this domain.

Jones et. al. [30] and Asher and Sumner [14] have investigated whether transputer-based systems can provide sufficient real-time control computing power for the ac induction motor to be an attractive alternative to dc machines. Different control schemes based on the application of functional decomposition are described in [31]. The relatively slow interprocessor communication of the transputer, however, limits performance although its ease of implementation affords the designer valuable insight into the potential of parallelism. It is inferred that a parallel processing scheme is likely to yield a viable solution - perhaps the new generation of transputers with planned order of magnitude faster link communications will fulfil this promise.

Real-time adaptive control of the terminal voltage and speed of a generator using a self-tuning regulator (STR) to adjust the excitation is discussed in [19]. The existing controller, for a laboratory scale turbogenerator, is implemented in C on an IBM PC-AT compatible machine with a numeric coprocessor. With a reduced form of the Generalised Minimum Variance algorithm, the fastest rate at which the control signal can be updated is 20ms. However, to meet industrial standards, this time must be at least halved. Timing results on a real-time occam simulation of the turbogenerator system suggest that this target can be comfortably met with the controller implemented in occam on a single T800 processor. Indeed, the power of the transputer is such that the algorithm itself can be improved by employing a 9-parameter ARMA model which includes noise parameter estimation and a consequent improvement in performance. More recent laboratory test results support these simulation conclusions [32].

6.3. Kalman Filtering

Systolic arrays provide a fine-grained description of the concurrent computation involved in regular, matrix based algorithms like Kalman filters. Maguire and Irwin [18] describe how systolic Kalman filters can be realised on transputer arrays and make comparisons with a heuristic partitioning approach using Gantt charts. The systolic square root covariance filter is found to provide the most efficient implementation because of the close match between the systolic algorithm and the hardware architecture.

Related work [33] on transputer implementation of tracking Kalman filters, where an Extended Kalman filter is used for nonlinear state estimation, involves linearisation by updating a Jacobian matrix. Not unexpectedly, this research demonstrates that exploiting the structure of the system matrices allows a substantial reduction in the iteration time as compared with the more generally applicable systolic mapping approach. Interestingly, the iteration times of the transputer based tracking filters compare favourably with results obtained on the WARP processor quoted more recently in the literature [34].

Atherton et al. [35] employ a parallel processing system to track multiple targets arising from radar measurements. When targets are in the vicinity of one another (for example, when they cross) multiple tracking algorithms must be activated to monitor track development. This potentially very demanding computational problem is best met using parallel processing. An interesting problem here concerns the optimum number of processors required, since the maximum computational load is dependent on the worst-case number of simultaneous target crossings.

6.4. Aerospace

In an early Demonstrator Project, in collaboration with Royal Aerospace Establishment, Bedford, Fleming et al. [36] explored software and hardware mapping

strategies for the implementation of an existing flight control law - the Versatile AutoPilot (VAP). This study inspired the automated control law mapping environment (EPICAS) described earlier. The hardware solution involved four transputers in a "star" master-slave configuration and employed static task allocation. Simple fault scenarios were investigated and occam software solutions proposed. The Project was concluded with successful in-flight testing of the parallel processing flight controller interfaced with VME bus-based hardware onboard the BAe 1-11 test aircraft at RAE, Bedford.

Virk and Tahir [37] examine the problem of real-time, linear quadratic, optimal control of an advanced military aircraft with fast dynamics. The amount of computation involved in linearising the aircraft equations about the current state and control vectors has conventionally forced a number of simplifying assumptions like ignoring the cross-coupling between the longitudinal and lateral dynamics and assuming that the aircraft is time invariant over small time intervals. They formulate two optimal control problems for the linearised aircraft, corresponding to the lateral and longitudinal dynamics respectively, and account for the cross-coupling either explicitly, or implicitly in a modified control effort, and solve two decoupled sub-problems on a transputer network. The algorithm involves linearisation of the equations of motion, integrating Riccati equations in reverse time, computing the controls and updating the state vector. Functional parallelism is used, with 1 transputer responsible for the longitudinal computation and the other handling the lateral calculations. A number of methods for off-loading some of the calculations onto a third processor are also investigated. The parallel autopilots are tested on a transputer based C simulation of the aircraft and it is demonstrated that real-time control can be achieved with a fast linearisation update as required.

6.5. *Fault Tolerant Systems*

Clearly, a parallel processing system has great potential for fault-tolerance. However, it must be recognised that such a system will tend to be less reliable than a uniprocessor system since it generally has more elements. It only becomes more reliable if it can detect a fault and take corrective action, possibly by circumventing the defective sub-system and reconfiguring its software.

Thompson and Fleming [38] use existing fault tolerant techniques to evolve an operationally fault tolerant transputer-based architecture suitable for gas turbine engine control. A system topology, constrained by the dual-lane configuration of gas turbine engine controllers, is devised in a way in which the majority of faults are detected, located and masked by means of a three-way vote, consistent with the conventional triplex approach. They draw attention to limitations of the present transputer generation in devising their scheme.

Fault tolerant flight control involves reconfiguration of the flight control law after actuator failure to maintain flight conditions as close as possible to those of the unfailed system. To implement reconfiguration of the control, it is necessary to compute the control-mixer gain matrices which distribute the forces and moments of the failed surface to the remaining, still functional, surfaces. Virk and Tahir [39] give results to show that real-time performance of such a fault tolerant design can be achieved on a transputer network for an aircraft model supplied by British Aerospace (Military Aircraft Ltd).

The possibilities and limitations offered by transputers for hardware fault tolerance are discussed in [40]. It is shown that the availability of multiple communication links and a hardware scheduler are attractive for this purpose. Fault tolerance for a single processor implementation of a controller is treated, with transient and permanent fault

detection and correction discussed for the CPU, memory and I/O in turn. The addition of a second or third processor in cold, warm, hot or active back-up configurations makes it possible to achieve high availability or high safety in the event of a transputer failure. More complex control algorithms are implemented on multiprocessor systems where the significantly increased probability of failure is offset by the availability of more processors to take over the tasks of the failed processor. While hardware fault tolerance is still based on the use of back-up, additional problems arise and recovery control is more complex. These issues are examined and a flexible, multiprocessor, fault tolerant kernal is described.

Holding et. al. [41] consider software engineering methods for the design of real-time concurrent software. An integrated approach for the design of time-critical and system-critical software systems using Petri nets and temporal logic is described. Petri net models of the software constructs in occam are discussed together with the introduction of fault tolerant techniques for ruggedising the implementation software. The application area for this research is a new generation of flexible machinery in which mechanical complexity is traded for sophistication in control. Thus, mechanical transmissions used to synchronise actuator motions are replaced by sets of independent, electromechanical drives operating under software control.

6.6. *Real-time Simulation*

Ponton and McKinnel [42] report on the application of transputers to the real-time simulation of process plant. The goal is to exploit the benefits of a faster model solution to include not only steady-state conditions but also dynamic simulation. The latter employs complex nonlinear models typically consisting of a set of several thousand mixed differential and algebraic equations. Three different problem areas are discussed and it is interesting to note the approaches to parallelism used in each case.

For real-time simulation of a distillation column, geometrical parallelism is used to partition the distillation column into sections and place each on its own transputer. To make the transition from serial simulation code to parallel as simple as possible a modelling toolkit has been produced which runs on a 12 transputer Meiko surface connected to a Sun Sparcstation. Solution of the nonlinear algebraic equations is the key numerical problem in process plant simulation. A new algorithmic decomposition technique based on decomposing the equation set by "partitioning and tearing" is applied to calculating the pressures and flows of an incompressible fluid in a pipe network consisting of 113 nodes, 123 pipes, with 59 unknown pressures. A number of solution strategies, including a master/slave paradigm, are compared for solving the associated system of linear equations, evaluating the nonlinear function and updating the Jacobian. Finally the mapping technique is extended to cover the dynamic case.

Mapping onto transputer arrays is generally difficult and can involve the evaluation of a number of alternative strategies for a particular application which can be both expensive and time consuming. Ponton and his co-workers have built a model of the parallel application as an extension to the Simula programming language which allows the user to construct a model of a parallel program executing on a parallel computer.

6.7. *Instrumentation*

Non destructive testing (NDT) is used in industry to detect faults in products in order to provide a degree of quality assurance. Reference [43] describes an instrument

in which the parallel processing potential of the transputer is applied to electromagnetic eddy-current NDT of defects in tubular metal samples.

In the present instrument three fault conditions are recognised; a defect free sample, a sample with a defect on the inner wall of the tube and a sample with a defect on the outer wall of the tube. The probabilistic approach employed uses a set of Kalman filters corresponding to the set of likely fault conditions. These are implemented on separate processors, together with an appropriate probability update mechanism.

Using 20MHz T800 transputers it is possible to obtain sample feed rates of 1.9m/s with the current 2 mm resolution of the coil system which surrounds the sample under test. This is approaching the production line rates of metal tubing and hence presents real possibilities for an online quality assurance system.

Transputers have also been reported to provide a flexible and cost effective signal processing base in an instrument for real-time particle flow metering [44]. The main applications for such an instrument include monitoring the particle concentration of effluent gases in industrial processes and control of the flow from spray guns in the electrostatic particle coating process.

7. Conclusions

The transputer has been readily accepted by the control community for embedded parallel control systems and a broad range of successful applications has been reported.

This experience has identified a number of technical limitations for this application domain. The built-in scheduler has only two priority levels, the interrupt handling facility on the basic occam-transputer combination is rudimentary and the number of links per processor limits connectivity.

Acknowledgements

George Irwin wishes to acknowledge the support of the Science and Engineering Research Council, the Royal Signals and Radar Establishment, Malvern, Short Bros. Plc, Belfast and the IFI Institute of Advanced Microelectronics for some of the work reported here. Similarly, Peter Fleming acknowledges the support of the Science and Engineering Research Council, the ESPRIT Parallel Computing Action, Royal Aerospace Establishment, Bedford and Smith's Industries.

References

[1] A.J. Chipperfield, T.P. Crummey and P.J. Fleming, Decision making in CACSD: multiobjective optimisation and parallel processing, *IEE Colloquium Digest on "The Control Factory"*, 1991.

[2] A.E.B. Ruano, *Applications of neural networks to control systems*, Ph.D. dissertation, University College of North Wales, 1992.

[3] H.S. Stone, *High Performance Computer Architectures*, Addison Wesley, 1987.

[4] C.A.R. Hoare, Communicating sequential processes "CSP", *Comm ACM*, Vol. 21, 1978, pp. 666-677.

[5] A.W.P Bakkers, R.M.A. van Rooij and L.James, Design of a real-time operating system (RTOS) for robot control, *Proceedings of the 7th Occam User Group*, IOS Press, 1987, pp. 318-327.

[6] H. Thielemans and E. Verhulst, Implementation issues of Trans-RTXc on the transputer, *Proc. IFAC Workshop on Algorithms and Architectures for Real-Time Control*, Bangor, UK, 1991.

[7] P.H. Welch, Multi-priority schedulers for transputer-based real-time control, *Real-Time Systems with Transputers*, IOS Press, 1990.

[8] D.I. Jones and P.M. Entwistle, Parallel computation of an algorithm in robotic control, *Proc. IEE Int. Conf. Control '88*, 1988, pp. 438-443.

[9] P.N.F. da Fonseca, P.M. Entwistle and D.I. Jones, A transputer based processor farm for real-time control applications, *Applications of Transputers 2*, IOS Press, 1990, pp. 140-147.

[10] H.A. Thompson and P.J. Fleming, Fault tolerant transputer-based controller configurations for gas turbine engines, *IEE Proc.*, Pt. D, Vol. 37, 1990, pp. 253-260.

[11] D.F. Garcia Nocetti and P.J. Fleming, *Parallel processing in digital control*, Springer Verlag, 1992.

[12] P. Shaffer, Experience with implementation of a turbojet engine control program on a multiprocessor, *Proc. American Control Conference*, 1989, pp. 2715-2720.

[13] A.W.P. Bakkers and J. van Amerongen, Transputer based control of mechatronic systems, *Proc. 13th IFAC World Congress*, Tallinn, 1990

[14] G.M. Asher and M.Sumner, Parallelism and the transputer for real-time high performance control of a.c induction motors, *IEE Proc.*, Pt. D, Vol. 137, 1990, pp. 179-188.

[15] J.W. Ponton and R. McKinnel, Nonlinear process simulation and control using transputers, *IEE Proc.*, Pt. D, Vol. 137, 1990, pp. 189-196.

[16] F.M.F. Gaston and G.W. Irwin, Systolic Kalman filtering: an overview, *IEE Proc.*, Pt. D, No. 4, 1990, pp. 235-244.

[17] F.M.F. Gaston and G.W. Irwin, A systolic linear quadratic optimal controller, *Electronics Letters*, Vol. 26, No 14, 1990, pp. 1000-1002.

[18] L.P. Maguire and G.W.Irwin, Transputer implementation of Kalman filters, *IEE Proc.*, Pt. D, 1991, Vol. 138, pp. 355-362.

[19] L.P. Maguire and G.W. Irwin, Parallel adaptive control, *Proc. 1st European Control Conf.*, Vol. 1, Grenoble, France, July 1991, pp. 590-596.

[20] C. Moler, *MATLAB User's Guide*, Department of Computer Science, University of New Mexico, Albuquerque, USA, 1980.

[21] L.P. Maguire, *Parallel architectures for Kalman filtering and self-tuning control*, PhD dissertation, The Queen's University of Belfast, 1991.

[22] G.I. Dodds, *Personal communication*, 1991.

[23] H. Mirab and P.J. Gawthrop, Transputers for robot control, *Proc. 2nd International Transputer Conference*, Antwerp, BIRA, 1989.

[24] R.W. Daniel and P.M. Sharkey, The transputer control of a Puma 560 robot via the virtual bus, *Proceedings IEE, Pt. D*, 1990, Vol. 137, pp 245-252.

[25] A.W.P. Bakkers, J. Meijer, J.C. Musters and H.G. Tillema, Real-time robotic control using transputers, in *Transputers for Real-Time Control*, eds. G.W. Irwin and P.J. Fleming, research Studies Press, Chapter 3, 1992.

[26] J.M. Brady, H. Durrant-Whyte, H. Hu, J.Leonard, P.J. Probert and B.S.Y.Rao, Sensor based control of AGV's, *IEE Computing and Control*, 1989.

[27] T. Ogasawara and G. Dodds, Transputer based simulation of a tele-operated manipulator, *Proc. 9th Annual Conf. of the Robotics Society of Japan*, Japan, 1991.

[28] T. Suehrio et. al., Task coordinate servo system for a manipulator using the transputer, *Proc. Japan Robotic and Mechatronics Conf.*, Japan, 1991, pp 5-9. (in Japanese)

[29] K. Kodairo et. al., A controller system for an ultra precision stage using multi transputers, *Proc. 30th Annual Conf. of the Society of Instrument and Control Engineers*, Japan, 1991, pp 161-162. (in Japanese)

[30] D.I. Jones and P.J. Fleming, Control applications of transputers, in *Parallel Processing in Control - the transputer and other Architectures*, ed. P.J. Fleming, Chapter 7, Peter Peregrinus, 1988.

[31] P.M. Entwistle, Parallel processing for real-time control, *PhD dissertation*, University College of North Wales, 1990.

[32] M.D. Brown, *Transputer implementation of adaptive control for turbogenerator systems*, PhD dissertation, The Queen's University of Belfast, 1991.

[33] R.J. Kee and G.W. Irwin, Transputer implementation of tracking Kalman filters, *Proc. IEE Int. Conf. Control '91*, 1991, Vol. 2, pp 861-866.

[34] R.S. Baheti, D. R. O'Hallaron and H.R. Itzkowitz, Mapping Extended Kalman filters onto linear arrays, *IEEE Trans. on Automatic Control*, Vol. 35, No. 12, 1990, pp 1310-1319.

[35] D.P. Atherton, E. Gul, A Kountzeris and M. Kharbouch, tracking multiple targets using parallel processing, *IEE Proc.*, Pt. D, 1990, Vol. 137, pp 225-234.

[36] F. Garcia Nocetti, H.A. Thompson, M.C.M. de Oliveira, C.M. Jones and P.J. Fleming, Implementation of a transputer based flight controller, *IEE Proc.*, Pt. D, Vol. 137, 1990, pp 130-136.

[37] G.S. Virk and J.M. Tahir, Parallel processing for real-time flight control, in *Transputers for Real-Time Control*, eds. G.W. Irwin and P.J. Fleming, Chapter 4, Research Studies Press, 1992.

[38] H.A. Thompson and P.J. Fleming, Fault tolerant transputer-based controller configurations for gas turbine engines, *IEE Proc.*, Pt. D, Vol. 137, pp 253-260.

[39] G.S. Virk and J.M. Tahir, A fault tolerant flight control system, *Proc. Int. Conf. Control '91*, Vol. 2, 1991, pp 1049- 1055.

[40] R. Cuyvers, R. Lauwereins, C. Caerts and J. Peperstraete, Hardware fault tolerance: possibilities and limitations offered by transputers, in *Transputers for Real-Time Control*, eds. G.W. Irwin and P.J. Fleming, Chapter 8, Research Studies Press, 1992.

[41] D.J. Holding and J.S. Sagoo, A formal approach to the software control of high speed machinery, in *Transputers for Real-time Control*, eds. G.W. Irwin and P.J. Fleming, Research Studies Press, Chapter 9, To appear 1992.

[42] J.W. Ponton, E. Fraga, R. McKinnel and N. Skilling, Simulation of nonlinear chemical processes and control systems using transputers, in *Transputers for Real-Time Control*, eds. G.W. Irwin and P.J. Fleming, Chapter 2, Research Studies Press, 1992.

[43] M.R. Bahramparvar and J.O. Gray, Application of parallel processing techniques to eddy-current NDT instrumentation, *IEE Proc.*, Pt. D, 1990, Vol. 137, pp. 211-224.

[44] E.Mills and B.C.O'Neill, Particle flow instrumentation, in *Applications of transputers 2*, 1990, IOS Press, pp 56-62.

Transputer Applications
M. Jane et al., Eds.
IOS Press, 1992

Transputers in Image Processing

Peter E. Undrill.
Department of BioMedical Physics and Bioengineering,
University of Aberdeen, Foresterhill,
Aberdeen AB9 2ZD, Scotland, UK

Abstract. The advent of the INMOS transputer series of processors enabled the image processing specialist to experience computational power far in excess of that normally provided by conventional computing systems. This review looks at a range of image processing tasks addressed by the transputer using community, identifying some of the benefits and difficulties within their applications, and attempts to look towards opportunities in the future.

1. Introduction

'We may communicate by the written word but our thoughts are guided by images'

2. Fundamental Concepts

Computational image processing can be defined as the operation of mathematical functions on numeric representations of pictorial scenes. In general it is part of an overall process of visual perception, pattern recognition and image understanding. These form the essential components of computer vision. Aspects of each are described in detail in [1,2,3,4,5]. Fortunately image processing is conceptually rather simpler than many of the cognitive processes associated with computer vision and has been the subject of rather more practical and successful effort. Whilst it may be of scientific interest to exploit mathematics in the pursuit of perception, the objective basis of *image processing* is to apply an algorithm to a representation (usually digital) of a visual scene to produce a result which provides added value to:

- Understanding through quantitation.
- Perception through an improvement in a chosen index of quality.
- Efficiency through improved image coding.

2.1. *Image Analysis*

The goals of *image analysis* can take a variety of forms:

- A complete symbolic description of an image at an adequate level of abstraction.
- A list of interesting events, or objects, occurring within the image.
- A description of changes which have occurred between successive recordings of an image.

The first of these definitions allows us some flexibility in the chosen level of abstraction. In image processing we will normally be operating at the lowest level, which names regions, pixels or lines and attributes characteristics. At higher levels we begin to bring in knowledge of the imaged domain.

As an example, Fig. 1 shows an image taken from a high resolution brain scan using magnetic resonance imaging. Applying a method such as that described [4] we could say (right /left orientations often reversed in anatomy):

"There is a region which can be surrounded by a contour C2 and the intensity level in this region is 18 ± 8 % lower than that identified by contour C1"

At a higher level of abstraction we could infer:

"The hydrogen proton density in the region of the caudate nucleus adjacent to the frontal horn of the right lateral ventricle is 18 ± 8 % lower than that of the white matter adjacent to the posterior horn of the left lateral ventricle"

At a higher level still we could begin to relate this to normality, or to proposed pathology, with relative degrees of certainty.

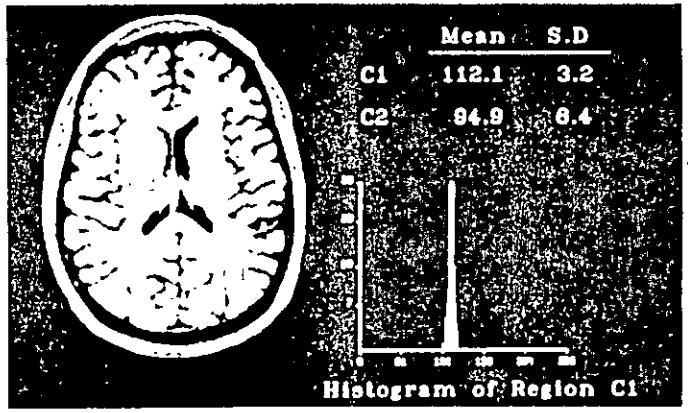


Fig. 1. MRI Brain Image analysed for regional characteristics.

The general structure of the overall imaging system can be described as in Fig. 2.

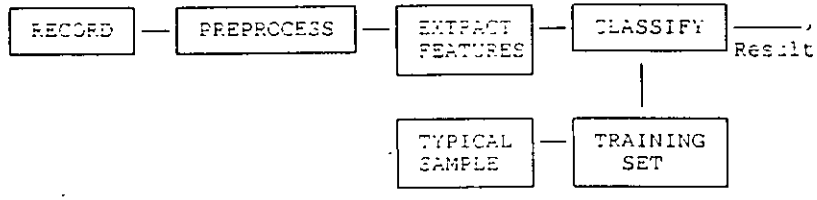


Fig. 2 A General Model of Image Analysis.

Computer vision and image interpretation concentrate on the final stages which assign object classes according to the parameters of extracted features, and deduce hypotheses linking the real world with the imaged world.

2.2. *Image Processing*

Image processing is largely concerned with:

- A pre-processing stage which, for example, might attempt to reduce noise or change / enhance image contrast.
- The segmentation of the image into areas of interest.
- The allocation of features such as shape, texture, orientation and size.

Typical Application areas are:

Inspection	Verification of assembly, quality control in production, directional guidance systems, face and signature recognition and document processing.
Medicine	Tomographic reconstruction, cell recognition, automated screening, image-guided surgery and multi-sensor sensor imaging.
Environment	Remote sensing of land use, environment monitoring and international surveillance.
Entertainment	Virtual reality and image compression for broadcast purposes.

2.3. *Image Formation*

Images are formed either by reflection or transmission, in the first case a typical example is an object illuminated by visible light and recorded with a camera. The *geometry* of the resultant image is governed by the general translation of a 3D object (world coordinates) to a 2D scene (camera coordinates).

Transmission images, on the other hand, result from the passage of radiation through an object. The *structure* of resultant image depends on the internal properties of the object. A typical example is the normal medical X-ray film, producing an image in two dimensions of integrated electron density, thereby discriminating bone from soft tissue. A variant of this, tomographic imaging, produces a cross section (transverse) image and is the result of processing transmission profiles taken at a number of different angles of illumination, which are then built into a three dimensional data space.

The images we deal with may be simple, as in a single printed character, or complex as in an aerial photograph or medical image. The two fundamental characteristics of images are their *spatial* resolution (number of pixels in orthogonal axes) and their *intensity* resolution (number of bits per pixel). How this digital structure is presented to the observer is a function of the visualisation system, not that of the processing system. It is the multi-dimensionality of images, from 2D to 4D (time sequenced 3D), which often leads to data volumes of such magnitude that severe limitations are placed on the form of processing that can be realistically applied within the time available.

As an example, a magnetic resonance imaging system might be used to provide data on the movement of the knee joint. Each image plane is collected at a spatial resolution of 256 x 256 pixels with up to 256 intensity levels. In each data volume there are 128 planes. Eight such data sets are collected, each at a different knee angle. This

examination would take up over 67 MBytes and it needs considerable computing power to do quite simple things with the data in any acceptable time scale.

3. Practical Algorithms

When we examine the image processing literature to establish useful techniques, a number of common themes are evident:

3.1. *Coding*

Simple methods are based on run-length criteria mapped onto the image in a planar (x,y) or vertical (intensity) direction. Other approaches represent the outlines of *sameness* by a chain code. More complex approaches use image transformation, often based on spectral (Fourier) or eigenvector (Karhunen-Loeve) methods [6,7,8], to exploit the redundancy evident in structured images. A fundamental issue is whether the compression results in an exact representation of the original image or is one which, whilst probably achieving a greater compression, results in some loss of image quality on recovery.

3.2 *Normalisation*

Normalisation is the re-mapping of images in space, intensity or time. The most common, and often most contentious process, is that of spatial interpolation [9]. The problem is that all methods enforce some form of filtering as well as geometric change on the image by the creation or deletion of image points. The most popular methods, though not the most favourable, rely on bilinear interpolation or spline fitting.

Intensity remapping, often called histogram equalisation [10,11,12], deals with the problem of image values being badly distributed among the available display levels. The equalisation may distort the original visual presentation of the image, but aid the visibility of otherwise hidden features.

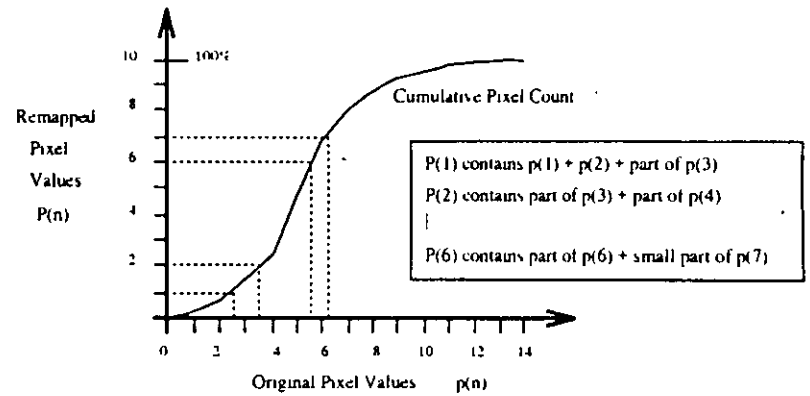


Fig. 3. Intensity Warping : Histogram Equalisation

The most straightforward method uses the curve representing cumulative distribution to remap pixel intensities.In Fig. 3, P(x), $[0 < P \leq 1]$, is formed from the measured points,

then the y axis is separated into n equal bins which are allocated values within the ranges mapped from the original x axis. Difficulties arise in deciding which *specific* pixels to reallocate.

In an application in medical imaging, the original X-ray CT transverse section head scan in Fig. 4(a) is histogram equalised to form the image in Fig. 4(b). Since the areas of interest are seldom uniformly distributed across the full image, regional forms of histogram equalisation, capable of adapting to the image structure, but needing much more computing power can be used to give Figs. 4(c) and 4(d).

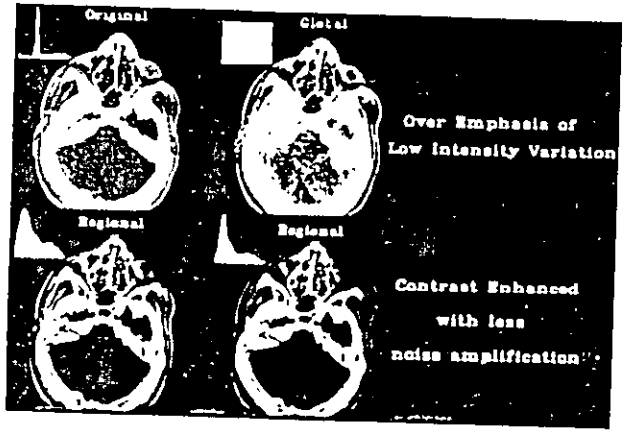


Fig. 4. Histogram Equalisation (a) Original, (b) Global Equalisation (c) Adaptive Equalisation, (d) Contrast Limited Adaptive Equalisation.

Geometric warping [13,14] implies the organised remapping from one set of coordinate spaces to another using control points (Fig. 5), and is usually carried out by reference to the minimisation of a suitable cost function.

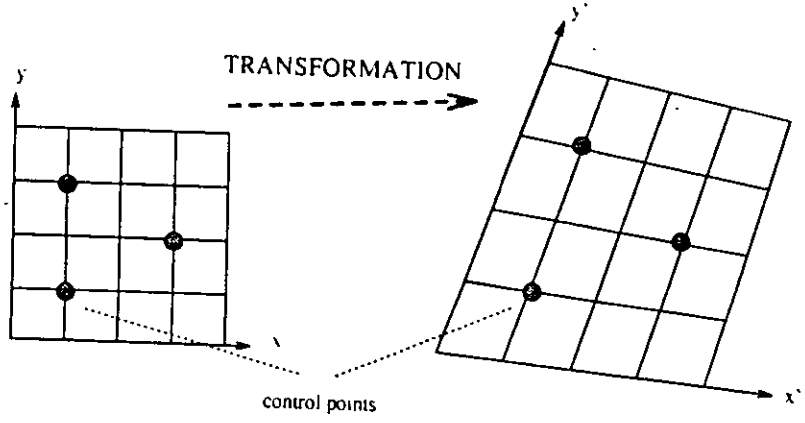


Fig. 5. Spatial Warping : Geometric Correction

This is an increasingly common application, for example, in relating two earth resources images from different satellites or comparing two medical images, one of which might measure functional characteristics such as glucose (energy) uptake and the other anatomical structure with each being derived from independent patient scans.

For linear, first order warping, if p_i and q_i are the coordinates of marker i in related images, the relationship is simply expressed by warping operators.

$$p_i = T [S [R (q_i)]] + N_i$$

where T, S, R are the Translation, Scaling and Rotation matrix operators and N_i is positional noise. As this matrix multiplication does not, in general, commute, least squares minimisation of $\sum(p_i - TSR(q_i))^2$ has to be attempted. Techniques which decouple translation and rotation [15], solving the first by Singular Value Decomposition and the second by movement of the centroid have been found useful.

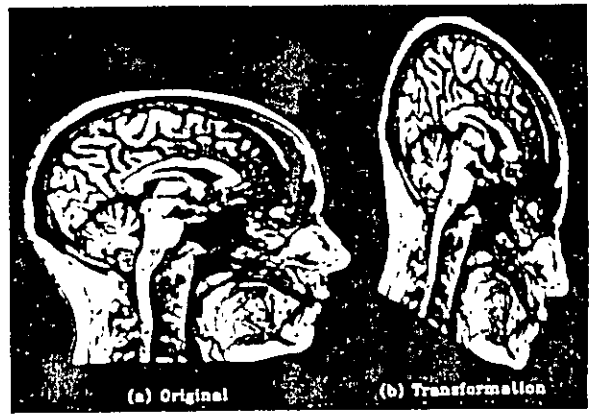


Fig. 6. Geometric Re-mapping of an MRI Brain Image

3.3. Linear Filtering

A powerful method of image computation is to apply a system process P to an input function F(x,y) and generate a transformed output function H(x,y).

$$H(x,y) = P [F(x,y)]$$

Typical transformations are those which attempt to reduce noise by smoothing or enhance intrinsic characteristics such as edges. This category includes the most common image processing function, that of applying *filters*. A system is linear if for two arbitrary functions f(x) and g(x) the superposition property holds.

$$P[a.f(x) + b.g(x)] = a.P[f(x)] + b.P[g(x)]$$

Convolution in the *spatial* domain and the Fourier transform in the *frequency* domain are the most well known linear operations [16,17,18], allowing:

- Selective attenuation of high spatial frequencies to achieve smoothing or noise reduction, or,
- Reduction of the relative proportion of low frequencies thereby enhancing edges and helping to locate regional boundaries via image restoration [19].

In the practical realisation of the Convolution Integral, linear processes apply a mask S(u,v), [size m x m], to all points of an image I(j,k), [size M x M], with the formation of sums of products:

$$\text{Output}(j,k) = \sum_{u=0}^{m-1}\sum_{v=0}^{m-1} I(j-u,k-v) \cdot S(u,v) \quad \text{for } j,k = 0,1,\ldots,M+m-2$$

whereas the Fourier Transform generates spatial frequencies F(u,v) from a periodic sequence of sampled values of f(j,k), j,k = 0,1,.....M-1 according to:

$$F(u,v) = \sum_{j=0}^{M-1}\sum_{k=0}^{M-1} f(j,k) \cdot \exp\left[-i2\pi(u.j/M + v.k/M)\right]$$

Edge detection is often a more complex problem than simple filtering, such as applying the straightforward Sobel mask, might imply. More sophisticated approaches [20], involving Difference of Gaussians techniques, which attempt to optimise detection and edge localisation, while minimising multiple responses are often needed.

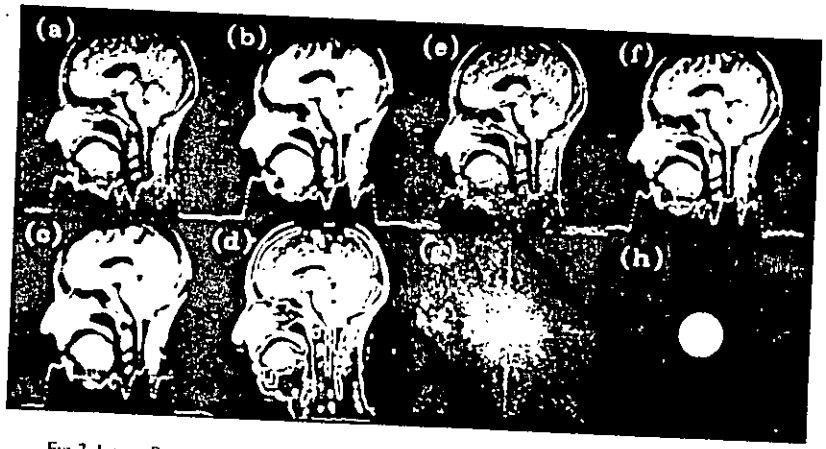


Fig 7 Image Processing Functions (a) original (b) Average, (c) Median, (d) Edge Detection, (e) Sharpening (g) Fourier Spectrum (h) Band-limited Fourier Spectrum, (f) Reconstructed Image, (mid-image profile where appropriate)

3.4. Non-Linear Operations

If the image pixel values in a subset of locations V(i,j), from within image I, constrained by a m-tuple finite set of discrete coordinates are ranked according to:

$$R(i,j) = \{ r_1 \le r_2 \le \ldots \le r_k \le \ldots \le r_m \mid r_k \in V(i,j) \}$$

then standard operations involve computing a new pixel value h(i,j) from a *rank order operation* defined by $h(i,j) = \phi\{R(i,j)\}$.

The most common of these are:

• $h(i,j) = r_1$	Erosion
• $h(i,j) = r_m$	Dilation
• $h(i,j) = r_{(m+1)/2}$	Median
• $h(i,j) = r_m - r_1$	Contour Detection

The desirable property of the median is that it achieves noise reduction (smoothing) without blurring details within an image [21].

Often these morphological operators are applied to binary images and can be thought of as *shape operators* [22,23] Erosion causes a shrinking of the image and dilation an expansion. When combined in sequence they give the actions of Opening and Closing respectively. The first of these tends to eliminate small structures such as sharp peaks or narrow bridges, the second tends to fill in small background areas such as holes or gaps. Fig. 7 shows some of the more common image processing techniques.

3.5. Segmentation

Decomposition or segmentation of an image, or sequence of images, into simpler, more meaningful components is often a complex task. Primitive operations which are used to segment shapes are:

- Thresholding to generate a binary object [24] The threshold level may be *global*, applying across the full extent of the image, *local* where it is governed by a defined region, or *dynamic* in which case it depends on each pixel value examined.
- Edge detection, which once an image has been segmented into a binary object, is a trivial process, whereas contour following [25] by dynamic programming and the linkage of extracted points involves much more computational effort.
- Mathematical classification of shapes by one or more moments of order m,n, which can be made invariant to translation, rotation and scaling, typically:

$$\text{Moment}_{m,n} = \sum_{x=0}^{M-1}\sum_{y=0}^{M-1} x^m.y^n.f(x,y)$$

- Chord length distributions, Fourier descriptors or by decomposition into triangles or polygons [26].
- Computational intensive methods, notably the Hough Transform which effects changes into parametric space such that searching for a cluster of events will indicate the likely size and position of a regular and, more recently, irregular objects [27,28].

In the presence of noise, shape discrimination may be less reliable than region identification (finding the interior space within a contour). Here we are looking for some feature of homogeneity. A popular technique is *region growing* [29] where a region is seeded and grown for as long as a given parameter is satisfied, or *split and merge* methods [30,31] which start with the whole image and progressively split those areas which fail a regional test into four (or eight) equal, smaller regions. At some point the criteria will be met by a number of neighbouring regions which can be combined into

one of irregular shape. The segmented two dimensional structure is usually described by a *quad - tree*, or *oct-tree* in the 3D instance.

3.6. Texture

Intuitively texture is an obvious phenomena though formalisation of texture has been much more difficult. Useful methods have been developed using simple statistics, auto-correlation, Fourier descriptors, co-occurrence matrices, grey level run lengths and fractals [32,33]. The breadth of methods give adequate opportunities for computational experiment.

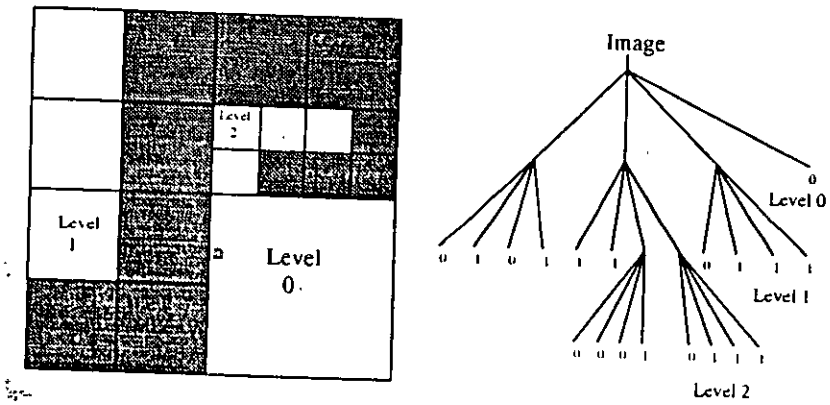


Fig 8. Quad - Tree Representation

4. Parallel Implementation of Image Processing Algorithms

Technology has provided us with abundant amounts of computer memory and the power of single processors is often more than sufficient to deal with *one-dimensional* calculations. The same sufficiency of memory has led to 2- & 3- dimensional scanners and digitisers capable of producing multi - MByte sized image sequences at affordable resource costs. The single processor concept has been less capable of executing many of the types of algorithms outlined above in acceptable time-scales.

The conventional approach has been to increase the speed of serial processors by the use of higher clock-rates, higher component density and faster technology. The concept of parallel processing allows, in principle, the data and algorithm to be distributed over many processors and offers scaleable and order-of-magnitude benefits.

There are, of course, a variety of parallel architectures [34,35]. The implementation instance of MIMD (Multiple Instruction Multiple Data) in the shape of the Transputer is clearly the main interest here although initial considerations might indicate the superiority of the SIMD (Single Instruction Single Data) architecture for general image processing. In practical implementations these SIMD systems involve large numbers of simple processing elements acting in synchronism whilst MIMD systems allow individual, fully specified, processors to undertake separate tasks, communicating with each other as required in a controlled manner.

5.1 Pipeline or Farm Parallelism

If we have a repeated process with little or no requirement for communication during each computation subtask, other than initialisation (which can be a quite considerable overhead in the case of large images), then a *processor farm* may be the best form of parallelism. Each worker in the farm undertakes the same process, one-in-a-sequence of processes. For example the 2D Fourier transform operation which is achieved by equivalent actions on sequential rows and columns.

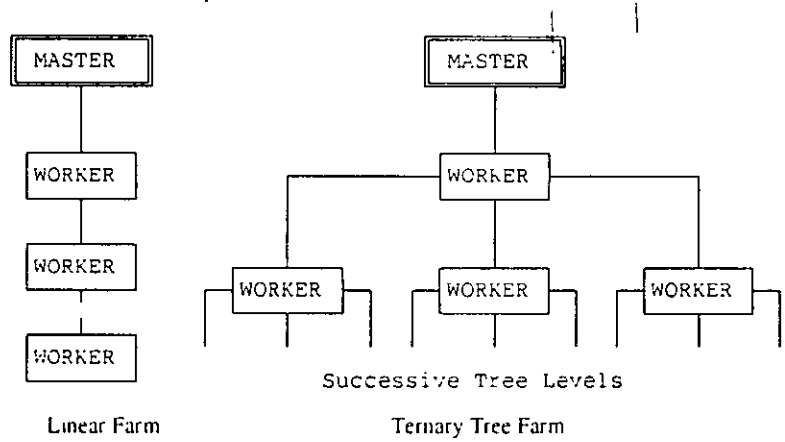


Fig. 9. Processor Farm Topologies.

If the farm involves many processing elements then the data transfer paths inherent in a linear toplogy become extended and can be reduced by rearrangement into a tree structure (binary or ternary branching) as in Fig. 9. A further consideration is that with too long a communication route within the processor farm and a high Communication & Control / Compute ratio, the processors at the end of the line can become under-utilised.

More often than not the actual implementation has to exist at a rather more complex level with the need to incorporate extra multiplexing buffers which attempt to

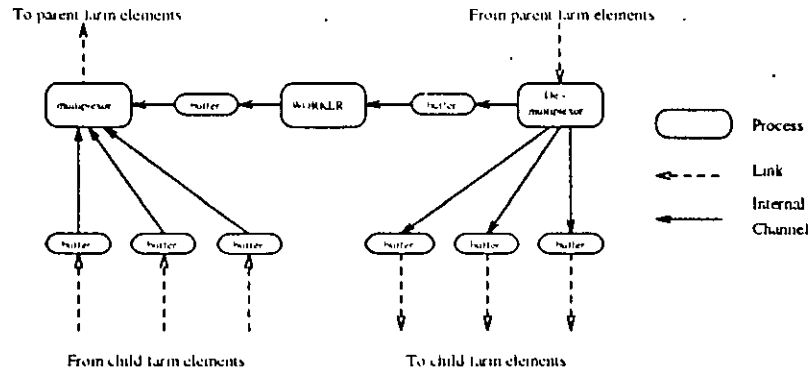


Fig. 10. Effective Ternary Processor Tree with Intermediate Buffers.

Whilst SIMD processing elements are often configured in square grids, transputers can be configured in numerous topologies such as pipelines, rings, binary or tertiary trees, arrays, meshes and hypershapes, allowing, in principle, task and workload dependent reconfiguration. One particularly interesting area this opens up is the prospect of a software *Supervisor* capable of comparing the progress of a task with its required completion time and allocating sufficient power to achieve the desired objective.

Two final considerations are, firstly, that whilst MIMD systems can be *derated* to model SIMD, the converse does not hold, and secondly, that biological systems have, in general, evolved into richly connected MIMD systems with a high degree of redundancy, so perhaps in principle at least we are taking the correct Darwinian path.

The main technical issues which impact on the decision of what form of parallelism to implement are:

- What type of parallelism does the problem intrinsically map into ? Once pattern recognition rather than linear neighbourhood processing is involved then the favourable properties of asynchronism can often be more readily exploited.
- Is there any consideration of cost ? Many SIMD systems have a large initial cost, whilst the scaleability of transputers allows a considered approach to initial experimentation and subsequent expansion. A corollary to this is that transputer systems are more likely to be compact, self standing and transportable, issues which are very important in providing successful applications in the general workplace rather than in the computing laboratory.
- Are there any principled advantages in using MIMD ? Typical answers might be that the mode of working maps more easily into cognitive processes, can produce more robust code with opportunities for redundancy, provides extra degrees of flexibility or poses academic challenges. The transputer and OCCAM have a number of positive features in each of these.
- What form does the working environment take and how long is the time-to-solution for any given task ? Here the situation is not so clear cut. The inherent flexibility of reconfiguration of transputer networks allows even simple changes to have quite profound effects on software stability.

Operation on 256^2 image	SIMD (AMT DAP 510)	MIMD (8 x T800)
3 x 3 Convolution (integer) Fourier Transform (real)	20 msec 600 msec	500 msec 1640msec

Table 1 Comparison of Convolution and Fourier Transforms on SIMD and MIMD processors

As Table 1 shows there is some justification for the initial premise, claiming the superiority of SIMD , when applied to elementary, repetitive algorithms using integer arithmetic, a feature which becomes less obvious as the algorithm complexity and floating point operations take precedence.

5. Practical Considerations of using Transputers in Image Processing

The basic question which has to be considered is the most appropriate form of parallel paradigm for the task at hand:

minimise the idle time of the workers and efficiently route data through to the other workers in the network, as shown in Fig. 10.

5.2 Geometric Parallelism

An alternative model for neighbourhood processing is one which allows data to be distributed across the network (or array of processors) and for each processor to act on subsets of that data. If the type of processing is one which takes an input array and generates a new value, or values, from operations governed by small target areas (masks) which are subsequently placed into an output array, then a simple replication of facing boundaries in the working arrays sent to each processor will suffice. This form of parallelism is often known as *domain decomposition* or *data parallelism*. Edge detection by convolution would be a typical operation.

As described in [36] the worker processors each receive their own working sections of an image and then copy boundaries between themselves in a symmetric manner. For an n x n (n is odd) neighbourhood process, each working sub-image must have a boundary extension of int(n/2).

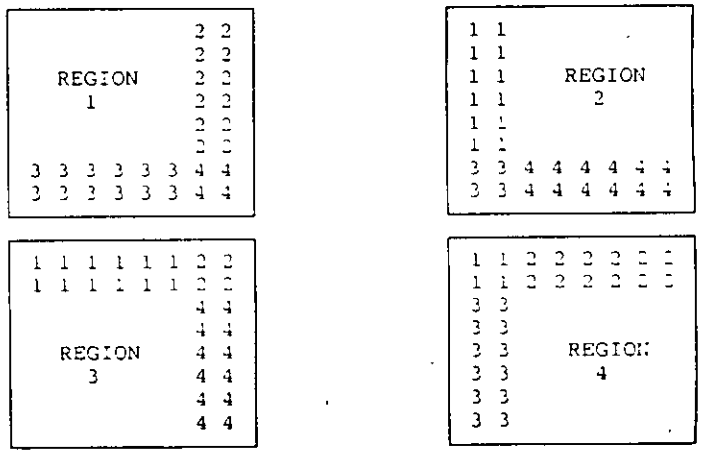


Fig. 11. Partitioning of image segments between four processors.

In the example above a 24 x 24 image is split into equal quartiles and sent to 4 processors, each of which operates on it using a 5 x 5 mask. If the pixels contained in the initial 6 x 6 quartile segments are designated by values 1 ... 4, then the working arrays have to be arranged as in Fig. 11, which shows the partitioning of the original data prior to processing. Whilst Fig. 11 indicates the arrangements for the *facing* edges it is usual to pad out the rows / lines of the partitions which represent the *external boundaries* of the original image with estimates, often zeros, to allow the filter to act up to the natural extent of the data.

If however the working array is being updated for both input *and* output such as in the optimal solution of electric field distribution as given Laplace's Equation using Gauss - Siedel methods, then the communication model has to be adapted to deal with the larger amount of interaction inherent in the problem of updating common borders simultaneously.

5.3 Algorithmic Parallelism

In this approach each processor executes a sub-unit of the *total* algorithm. This can be achieved by examination of any natural coding of the problem either by the simple partitioning of segments of a repeat loop, or by analysing the algorithmic dependencies. In this form the data moves between processing elements and the concept can also be referred to as *data flow parallelism*. One major benefit is that no large individual storage sub-matrices are required. A difficulty is that since there is likely to be a greater degree of algorithmic interaction needed, the danger of deadlocks, inefficient load balancing and communication overheads become significant. Nevertheless, as workers with pure SIMD machines discover, normal neighbourhood processes can often be re-written in vector or matrix terms.

In the example shown in Fig 12, adapted from [37], of averaging an n x n image using an m x m matrix, the master sends m x (n - length horizontal vectors) to the Horizontal Processor and Vertical Processor in turn, each of which returns partial vector sums over m points (with due regard to the ends of the arrays) in one of the two orthogonal directions, to a results processor which then forms a final vector sum and divides throughout by m^2.

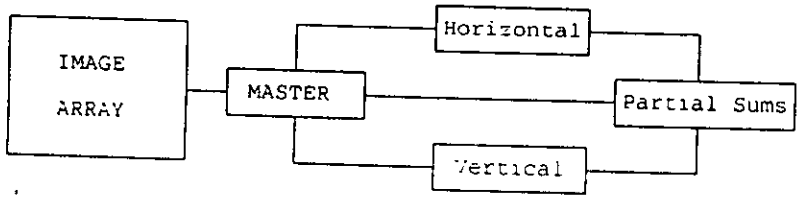


Fig. 12 Convolution by Algorithmic Decomposition.

This is an example where the valency of the transputer and current limitations in the switching / routing mechanisms which are available constrain the number of orthogonal processors that could usefully employed. The choice of appropriate topology, and often a complete redesign of an algorithm rather than the adaptation of an existing method, can help with the first of the above issues. However the size of sub-image data sets, their distribution to each transputer and the swapping of data between processors to resolve actions on image areas which need to be held by more than one transputer leads to practical difficulties which have to be resolved for many of standard image processing actions.

When any *specific* task is considered, these simple topologies or algorithmic decompositions often fail to provide the *single* most efficient approach. Ultimately, the quest for efficiency can only be resolvable by *dynamic* reconfiguration. This flexibility of topology and methodology can be viewed as the major advantage of transputers, or alternatively their prime disadvantage, which in its turn can lead to a lack of robustness, lack of generality, or difficulties of transferability in many implementations. For example, in a given problem, whilst a *divergent* tree might be appropriate for data distribution, a *convergent* tree might be best for data collection. The calculation phase may, at various times, be best described by a balanced linear farm, a pipe line of progressive algorithmic subtasks, an n-stage ring, a regular network of processors, or a one-to-many-connectivity neural network.

These design and management issues of topology and decomposition occur repeatedly in many of the reported applications.

6.0 Effectiveness of Parallelisation

In [38] the essential question of whether an image processing task which is divided amongst N processors can run in $1/n^{th}$ time is addressed. For point, neighbourhood and global domain decomposition, theoretical and practical results where found to agree to within 14%. Algorithmic Parallelism was seen as presenting much more of a challenge. If the 1/n rule is not obeyed then either

- The computation is unbalanced across the processors, because of a design fault or simply because the problem cannot be dealt with by a parallel algorithm, or,
- The processes depend on each other and are waiting for transfer of data or status information, or spend an excessive amount of time passing messages amongst themselves.

6.1 Case Study 1 - Image Reconstruction

An analysis of applying different forms of parallelism to a real problem in image processing has been presented by Byrne [39]. The example chosen is filtered back-projection, a popular form of image reconstruction from projections. The problem itself is relatively straightforward [40]. Image profiles $P_n(i)$ are taken at angles θ_n, equally distributed in a 360° rotation. The reconstructed image can be formed in two steps.

1. The original image profile P_n is corrected for detector sensitivity producing Q_n which is subsequently convolved with a function C to give the filtered profile F_n.
2. F_n is now projected over the image plane giving H(x,y), the reconstructed image.

Whilst a theoretical calculation of speed-up by parallel systems using 2000 processors has been given by Kingswood [41], and a processor farm-based solution to the instance of 90 profiles by Fourier Methods analysed [42]. Byrne [39] investigates the problem of reconstruction from few, approx. 30, profiles, typical of the data collected in nuclear medicine emission tomography, by reference to different parallel paradigms.

Method 1 uses geometric division of the *input* data set between available processors, whilst method 2 uses domain division of the *output* data such that a specific section of the final image is dealt with by each processor, whilst method 3 incorporates a pipeline approach to overall process, with the tasks being farmed out to the network in turn. The three methods were implemented on a system with 10 transputers connected as a linear farm or alternatively as a first-layer ternary tree with a second-layer binary tree.

The work analyses the number and dimension of communication events for each method and then how this is related to network topology. Whilst in method 1, the *whole image* array is needed by each processor, and hence has to be passed back for final addition, method 2 requires that *all the profiles* are available to each processor for its own section of the final image. In method 3 each profile is filtered by the root and back projected as farm processors become available, returning a list of coordinates and values. The results are shown in Table 2 and Fig. 13.

On a *single* processor the whole image reconstruction task takes 1.84 seconds. For method 1, each processor receives $1/^{Nth}$ of the number of profiles and proceeds to add partial sums to positions in its view of the whole array. Each parent sums its children's result with its own and passes its result to its parent. A decrease in performance improvement is seen as the communication time is progressively dominated by the N transmissions of the whole image. However the actual performance increases

with the number of processors used (at least up to 10) with an observed terminal Relative Inefficiency Factor (RIF) of 2.6 when compared with the ideal value.

For method 2 as the number of processors increases, the image size worked upon by each decreases and the final accumulation process simply places, rather than adds sub-images. Since each child needs the full data set, communication of input data is greater than in method 1. However, using this approach all processors carry out the convolution stages, introducing unnecessary computation and resulting in more gradual improvement and an RIF of 4.5 using 8 processors.

In the final method, the improvement flattens out as the ability of the root to supply filtered profiles is reached with an RIF of 2.9 using 6 processors.

Decomposition	Procs.	Target	Actual	RIF
Output Domain	8	0.125	0.54*	4.5
Input Domain	10	0.100	0.34	2.6
Algorithmic	6	0.166	0.27*	2.9

Table 2 Image Reconstruction: Relative Inefficiency Factor (RIF) against Method of Decomposition and Number of Processors [* - limiting value].

For this example, algorithmic decomposition achieves a better speed-up over few processors but then saturates early, output domain decomposition method continues its improvement over more processors, but levels out with a poor speed-up factor, whilst input domain decomposition continues to improve with added processors. Clearly, since most complete *jobs* require a sequence of different types of image processing actions, the choice of approach is not simple to establish.

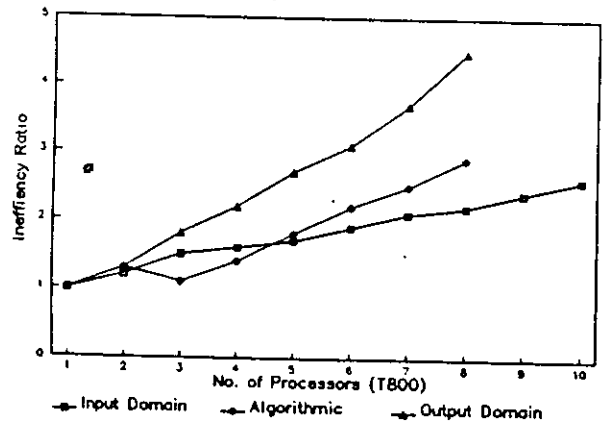


Fig 13. Improvement of Image Reconstruction with Number of Processors and Method of Decomposition

6.2 Case Study 2 - Image Presentation

The effective and timely presentation of 2D projections of 3D volumes is a computationally taxing task. Whilst the geometric transformations may be well established image processing components, especially those related to the optimum

methods of dealing with noise artefacts and sparse data sets remain significant. In an analysis by Lomax [43], multi-transputer systems have been investigated as part of a *practical* implementation of novel 3D medical imaging algorithms. Using up to 12 processors, two major components (tasks) of the image presentation environment were examined:

Task 1 Visible voxel detection, i.e. establishing which parts of a segmented voxel image are visible from a given orientation.

Task 2 Calculating the surface normal, and hence applying a lighting model to emphasise edge strengths.

Whilst the algorithms themselves may not be of interest here, even using the most aggressive techniques on serial processors (circa. 1990), clinically useful data presentation could not be presented within acceptable timescales. Cine sequences could only be obtained by prior selection, and single views, were taking about 20 seconds to produce. Fig 14 shows a sequence of images of segmented brain, visualised by an operator-controlled cutting plane.

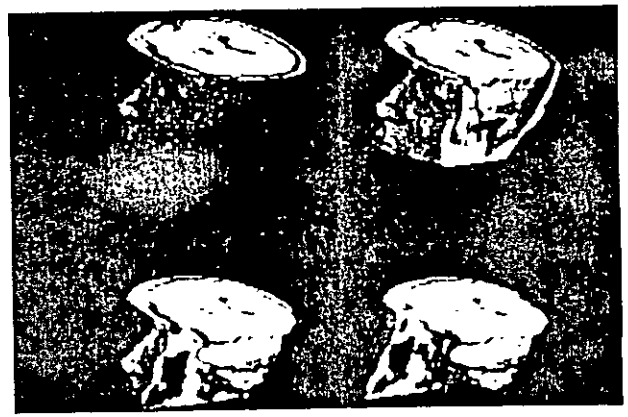


Fig 14 Presentation of the Brain as a Three-Dimensional Structure.

Each of the task units above were found to exhibit few opportunities for algorithmic parallelism. Data parallelism was examined in relation to unary, binary and ternary farm trees, populated by up to 12 worker processors, plus a master and a display controller.

Fig 15(a) shows the relative improvement of the visualisation routines as extra processors are added into the farm, and Fig 15(b) indicates the estimated Active / Passive ratios for each processor and network structure. This parameter reflects the amount of time spent communicating and waiting in relation to real work being executed, and is a factor which measures the latent overhead of the farm structure. When taken into account, a performance increase over the ideal single processor implementation of 7.8 is observed, instancing 12 processors in a ternary tree.

Definitive trends in packet size optimisation were less clear, other than the observation that broad optima existed at between 32 and 128 pixels for task 1, but task 2 needed data packets of at least 512 pixels.

Further analyses were carried out using global buffers [44], rather than local buffers for intra-transputer communications and the variation of performance with data packet size was also investigated. Global buffering was seen to have a more marked effect, offering improvements of up to 20% in algorithms which were shown to have many, small data communication packets and short processing times.

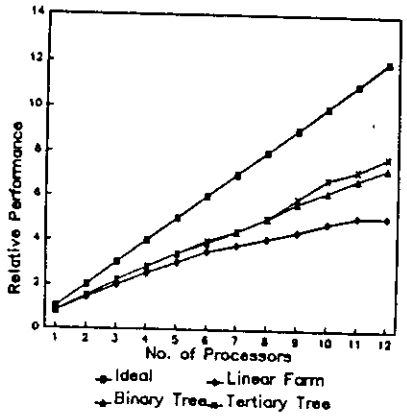


Fig 15(a) Improvement with No. of Processors

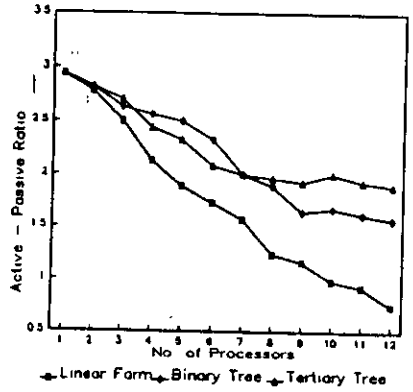


Fig 15(b) Active / Passive Ratios

Table 3 identifies the improvement when using global buffers for each task, assuming a ternary tree farm and a link transmission rate of 10Mhz.

Task	Packet Size	Proc. Time	Proc/Comm Ratio	Gain
Visibility	32 pixels	2 msecs.	12.5 : 1	8.7%
Shading	768 pixels	360 msecs.	2.92 : 1	3.4%

Table 3 Task Improvement from Using Global Buffers

As a result of using the ternary farm with global buffering, plus equal attention given to optimising floating point operations and image coherence, the time to produce an arbitrary selected view from a 128 x 128 x 128 data set has been reduced from 20 seconds / view on a VAX 11/750 to 150 - 250 msecs. using 12 transputers.

7. Review of Applications

The popularity of the transputer in image processing makes the comprehensive appraisal of the community's activities a formidable task. The degree of interest is indicated by the Image Processing Transputer Applications Community Club's over 160 registered members [45]. Another indicator can be formed by assuming that the 1990 and 1991 SERC/DTI Transputer Application Conferences are a representative sample of user interests. Out of a total of 221 papers presented 48 (22%) addresses image processing or image presentation problems. This section attempts to assess and summarise the range of interests.

7.1 Development and Support Environments

Many of the earliest investigations centred around the provision of resources and frameworks for further experimentation. Typical of these were the developments in which general purpose parallel processing environments were produced [46,47,48], or in which the processing of captured images in real-time was seen as an objective which could be more easily achieved by transputer implementation [49]. In a design suggested by Seed [50] the transputer is used to control fast memory, organise I/O controllers and enable custom processors to be incorporated. An alternative approach, which provided multiuser access to transputer facilities, is described by Ross [51] in the design of a Ethernet controller to interface VAX (and more recently PC and SUN) processors to a 20 x T800 network resource. This specific development enabled a number of projects in medical imaging, subsequently reported by Byrne [52] and Lomax [53].

Whilst several manufacturers have image processing software libraries which support transputer based parallelism in one form or other [54] the fundamental issues in designing a descriptor language appropriate to image processing with transputers has been addressed by Crookes et al. [55,56,57] out of the Image Algebra notation, in which the main concepts are that low level expressions can be expressed using *templates* and *images* [58], and that the underlying parallelism should be hidden from the user. Other approaches to software development are described by Dew [59] and Henderson [60].

7.2 Image Assisted Robotics

Image Processing is part of the link between image capture and *intelligent* robotics. At the Artificial Vision Research Unit at Sheffield University [61,62,63] a comprehensive 3D vision system has been developed involving 25 x T800 processors, industry standard MAXbus hardware and a SUN workstation. The first objective is to recover 3D geometry from general 2D scenes, by comparing pairs of Canny edge-filtered images obtained from binocular views. The information derived is matched with items held in a data base of shape-and-size descriptors. A final step allows a robot arm, under the control of a separate PC, to pick up one of the identified objects. Overall processing times from observation to pick-up of 10 seconds are quoted.

A medical application of scene analysis and machine guidance starts with the real-time image as generated from a colonoscope [64]. Using a transputer based dynamic reconfigurable parallel system, the image from inside the human colon is analysed with the objective of keeping the viewing head of the endoscope moving smoothly along the longitudinal axis of the colon. This is achieved by balancing the intensity of the sidewalls and ensuring the centre of the field is always held by the dark zone representing the distant view.

7.2 Recognition of Real-World Objects

A framework for the image classification based on the transputer is introduced by Fairhurst [65] in an application related to the recognition of postcodes. Alternative approaches to the partitioning of the chosen algorithm over sub-groups of processors are reported. Further theoretical analysis and simulated performance figures [66] show how a throughput of up to 20 patterns per second and an error rate of 1.4% can be achieved by a network of 8 transputers.

Progress in the same pattern recognition task is also described by Tregidgo [67]. Here the problem is examined starting from the premise that tree-type topologies can provide the scaleability necessary to allow the hardware to be extended to satisfy

operational demands. The predictive performance of using n-level ternary trees was seen as preferable to the loss of programming flexibility that rigid adherence to fixed topologies necessitates. Using this paradigm, performance simulation shows that processing rates of 10 postcodes per second can be achieved with 40 processors, an order of magnitude less favourable than Fairhurst.

The recognition of vehicle number plates has a number of applications including road pricing, analysis of traffic data, crime investigations and speed enforcement. Robust operation is essential, and whereas the interpretation problem is reasonably simple in the instance of a perpendicular viewpoint under bright viewing conditions, an oblique view of a mud-smeared number plate on dull day with spray from a wet motorway presents a much more difficult problem. Lotufo et al. [68] describe a system using a single T800 which classifies registration letters by shape recognition taking into account *a-priori* knowledge of the dimensional regulations governing UK number plates. An accuracy of 84% is quoted.

Face recognition is another complex task which is being attempted by a number of workers using a wide range of architectures. The extent of the general problem is clear from the changes of ageing, facial hair and spectacles that can affect of us. Lades [69] implements a system where facial images are characterised using Fourier transform methods into regionally dependent spatial frequencies and orientations. These features are represented by a connected graph, which can then compared with a data base of candidate faces. Using 31 x T800 processors, an individual face can be selected from 80 candidates in 30 seconds.

The theme of transferring existing code into a parallel form in the same language is reported by Morrow [70] as part of a recognition problem. The application was a large industrial FORTRAN program which segmented infra-red images of helicopters into component objects. The main conclusions drawn are that most difficulties stemmed from the different dialects of FORTRAN, that linear speedup was more successful with low-level image processing operations and that it helps to have the advice of the program originator. The overall success suggests that there is benefit in contemplating the *dusty deck* approach to generating parallel code.

7.3 Resolution of Stereoscopy

The inverse of 3D visualisation is the interpretation of 3D structure from 2D views. This is something that the human vision system general does very well, but is a complex computational task. By considering the inverse of the scaling, rotation and translation matrices on objects with well defined vertices, a captured image can be matched by exhaustive selection to a shape candidate [71] with a typical detection time of 2 seconds for a object having 8 vertices (parallelapiped) using 8 transputers. By incorporating knowledge into the algorithm, improvements of at least an order of magnitude are observed.

In most image processing applications we are dealing with images which are below 10^6 bytes. In the case of remote sensing, images of at least 4K pixels squared are commonplace. Holden [72] describes the problem of comparing two 2D views, as perceived by satellite imagery, to determine surface topology, a task which takes up to 6 days on a conventional system. Using a 48 processor Parsys Supernode system, the original code (> 3000 lines of C) was executed in one fortieth of the time taken on a SUN Sparc 1+ system. Image sections were processed using geometric parallelism and the authors state that very little recoding was necessary in the transfer.

7.4 Inspection

Detecting the corruption, including subversive alteration, of text between two notionally equal printed copies [73] can be achieved by producing a difference image which is then analysed for connectivity and spatial content. Using five T414 transputers an improvement over a IBM PC/AT from 29 to 1.4 seconds is reported at an 87% success rate.

Industrial inspection is reported by Netherwood [74] with the objective of confirming the likely viability of surface mount and solder joints. The technique chosen to outline regions is, once again, the Canny operator producing skeletal images which can be identified using shape descriptors and allow segmented areas of the original image to be measured in terms of a quality, such as texture, which can then be related to viability.

Another application is the inspection of labels [75]. As customers we feel annoyed if product labels are wrongly positioned, folded, torn or skewed. In the system described, using four transputers, checking rates of up to 20 labels per second have been recorded using prior knowledge of the dimensional characteristics of a satisfactorily placed label.

In a 'down-to-earth' application [76], an image of rock-blast debris is analysed with the objective of measuring the fragment size, so that blast engineers have time to adjust the design of subsequent blasts. Essentially the problem is one of segmenting the image into fragment classes and working out their dimensions and extent of overlap. Using 4 transputers, analysis time is reduced from the previous tens of minutes, appropriate to manual methods, to just 2 minutes.

7.5. Determination of Optic Flow and Dimensional Changes

The analysis of a series of real-time images, often taken from video cameras, can be used to answer questions such as how many objects pass by within a given time interval. The acquisition of more than one perspective view can reinforce the selection of objects from within a known candidate space. In another traffic related problem, that of calculating vehicle movement density within a given section of road space [77], those characteristics of objects which are likely to be owned by moving vehicles, such as road wheels or headlights are identified and lead to a vehicle count.

A rapid method of computation which identifies regions of high curvature (corners), then maps their movement across the image frame has been used [78] to quantitate absolute and relative movement of single and multiple objects. Using 8 x T800 processors, decision times of 2.7 seconds are achieved in an experiment involving the analysis of images of two vehicles, moving at different speeds, in parallel directions across the field of view.

Algorithms which may be successful in good viewing conditions often perform less well in poor, but not infrequent, operational conditions, for example, images of aircraft descending through mist. In this instance, ambiguities can be better resolved by using probabilistic relaxation algorithms [79], which have been tested on a four transputer host and will be implemented on a target system containing 20 processors. Another aspect, that of whether information from Red/Green/Blue video signals taken separately can help edge detection in a changing visual scene has also been considered [80]. A general scene is pre-processed to establish its information content which then can be used to derive the kernel size for a Canny filter. By comparison with a monochrome version of the same scene the conclusion is that the combination of evidence is beneficial.

When applied to large objects, such as the Humber Bridge, [81], sequential studies of images have been found to be superior to accelerometer measurements which cannot deal adequately with the low frequencies involved. Using a Quintek Harlequin board, 4 transputers and non-predictive searching, objects can be tracked at 83 pixels/sec movement at update rates of 3 frames per sec. This performance can be improved up to a theoretical limit of 400 pixels/sec if a prediction model is employed, although movements which are different from the prediction need careful management. By viewing fixed markers on the bridge's central span, lateral oscillations of 200 cm. and time dependent drift in excess of 100 cm. have been measured.

The study of sequential retinal images from ophthalmology studies is described by Byrne [52]. The objective is to compare similar regions of the retinal surface, using strong similarity features in successive images, which may have been captured over a period of a few minutes or be gathered over a many months. The underpinning method is one of cross correlation using strong features. Each of the processes of thresholding, the application of a Canny filter, object selection and matching by translation and rotation are described using single processor and 8 - processor instances. Efficiencies ranging from 95% to 14% are reported, dependent on the image processing task. One of the main obstacles to efficiency in this type of application is seen to be the need to distribute whole images to each of the worker processes and the difficulties of tracking linear features, in this case retinal arteries between processors, (a generic problem which is returned to in section 8.2). Using two or more strong features, such as bifurcations, matching of rotation was found to be correct to 0.2° and the estimation of translation was almost always correct.

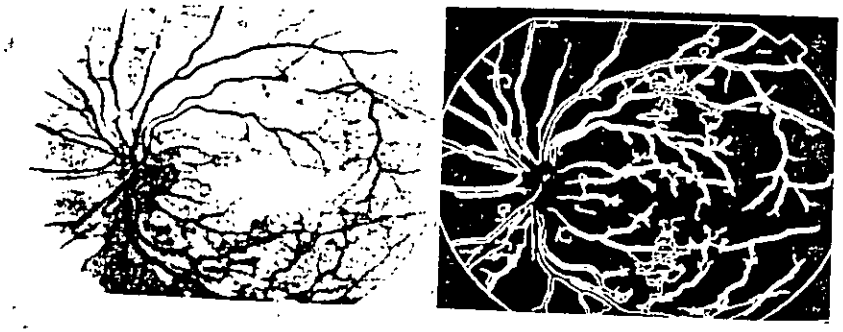


Fig. 16 Fluorescien Angiogram showing major blood vessels and its corresponding binary edge image for use as a correlation search area

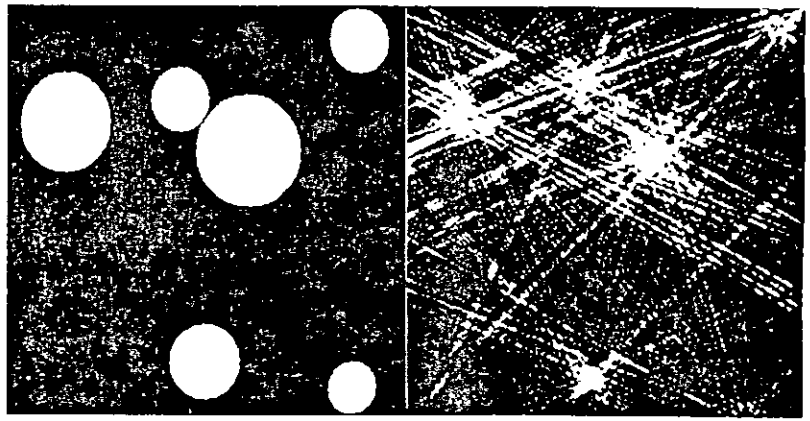
7.6 Transforms

Computationally intensive image transforms have always been seen as a challenging area for multi-processor implementation. As an example of *spatial* transformation, in the production of electric potential distribution within the brain starting from the conventional electroencephalograph [82], the computational load of interpolation by the *four nearest neighbour* method needed to produce a 128 x 128 image is carried out using four processors, arriving at the resultant image in a little over 1 second. In a medical

imaging problem dealing with *intensity* transformation [83], fluoroscopy images (X-ray without film) are corrected for scatter and glare in order to improve contrast and apparent resolution. With five transputers, the process times are reduced from over 20 minutes on a IBM PC/AT to 6 seconds.

The Hough Transform was earlier described as a method of clustering objects with similarities into common regions of parametric space [84] In essence the transform produces an alternative image space out of original, which is then searched for local maxima. Lotufo et al. [85] implemented line detection algorithm on up to 11 processors acting on a 256 x 256 image exhibiting two strong lines, achieving detection within 557 msecs. This theme was subsequently developed by Costa [86] to operate in binary arithmetic. Using a four T800 processor system, an effective improvement per processor of 38% was achieved. In the implementation on circle detection shown in Fig. 17, [87], rather slower operation of 3.14 sec. to detect position and radius were reported with 720 edge points and 6 processors using a non-optimised partitioning of the algorithm. This performance was improved by more than an order of magnitude by Eghtesadi [88] using a mixed processor environment of transputers and Texas TMS320C25 devices.

The relative immunity to noise of the Hough transform has been used by Yang [89] in an attempt to establish the orientation of an object moving across the field of view of a CCD camera. Using an array of nine transputers, the detection algorithm takes 33 msec. The experimental objective of a field rate of 60 Hz could be achieved by predicting the next position of the object from its previous motion.



(a) (b)

Fig 17. A selection of Circles (a) and their Hough Transform (b)

The Radon transform is one of the fundamental processes in the reconstruction of images from their profiles. Hall [90] describes its application to Synthetic Aperture Radar images by a Fourier Transform method, an alternative to the filtered back-projection approach mentioned earlier. Using a single T414 a transform was achieved in 4 minutes, with the objective of 4 seconds being obtained using 25 x T800 devices, organised into two ternary sub-trees. Further developments, reported in [91] review the prospects of dynamic switching of the topology to optimise efficiency and to direct processors to *where the effort is needed* in order to complete a task on time. In an alternative examination of image reconstruction using a Fourier back-projection interpretation of

Radon's method, Kelieff [42] uses a Meiko Computing Surface configured as a linear farm to reconstruct 90 projections into 256 x 256 image space in 7.14 sec., figures which when normalised for computational load compare well with those of Byrne [39].

Similar work on the reconstruction of nuclear medicine positron emission tomography (PET) images has resulted in the comparison of grids versus trees using input and output data decomposition [92,93]. In these studies, trees are once again found to be superior. However when the problems of PET reconstruction in *real time* are analysed, systems requiring at least 200 transputers of the T800 form are considered necessary [94]. In another medical imaging application, electrical impedance tomography holds out a prospect of non-invasive imaging of internal organs, by measuring their resistivity, without a large capital outlay on scanner equipment, albeit at present at a rather low resolution. The objective of reconstruction within an acceptable timescale with a PC host as data collector and display has been aided by the 20:1 speedup reported from using a single T800 transputer [95].

A extension into full 3D reconstruction, rather than by a series of 2D slices is reported by Zapata [96]. Applying the technique to the volume imaging of the Escherichia Coli micro organism, a 3D Fast Fourier Transform is produced from the prefiltered profile data and the resulting data set subjected to a low pass filter to reduce residual high frequencies (noise). Using up to 8 processors arranged as a 3 dimensional hypercube applied to objects of 32 x 32 x 32 pixels, total process times are reduced from 311 sec. to 44 sec., a speed up factor of 7.1.

7.7. *Representations*

As mentioned earlier the size of image data sets can present a problem, sometimes capable of resolution by quadtree (or octree in the case of volume data) representation, which although resulting in a more compact data structure introduces difficulties in subsequent analysis [97]. In the specific task of selecting variable orientation slices from a 256 x 256 x 256 image space, the Millar Polyhedron method [98,99], working on four transputers, extracts slices in 15 seconds as opposed to greater than 1 minute using data sets containing much smaller numbers image planes at 256 x 256 pixel resolution.

Segmentation of Synthetic Aperture Radar images is difficult since these are typical of images having a low signal to noise ratio. Starting from small regions (4 x 4 pixels) Cook [100] determines similarity measures which allow region merging. Using a Meiko system with 9 transputers, each processor grows a region until the merging criterion can no longer be satisfied within its section of the image. The whole image is then composed by the master processor, resulting in segmentation by regional characteristics.

Other approaches use straightforward data compression. The problem of safeguarding old photographic records from the Welch collection in the Ulster Museum, whilst still allowing public access has been addressed by Crookes [101]. Apart from heritage issues, having digitised the photographs to an appropriate resolution, the next natural step is to allow the viewer to *browse* using such standard techniques as zooming, contrast enhancement, spatial frequency alteration and edge detection. With 16 transputers an interactive image enhancement facility has been developed.

The design of image coding system aiming to transit video images in near real time across a Local Area Network is described by Chong [102,103,104]. Transputers are seen to give the flexibility necessary to respond to different codecs. The CCITT H261 recommendation is implemented on a network of 18 transputers, achieving a coding rate of 3 frames per second. An alternative method, based on fractals, is shown by Griffiths [105] to p e image texture, but with the possible introduction of blocking artefacts.

Starting with original images having 8 bit depth resolution, compressed images of 1.1 bits per pixel are reported without any noticeable degradation in image quality.

7.8 *Real Scene Simulation*

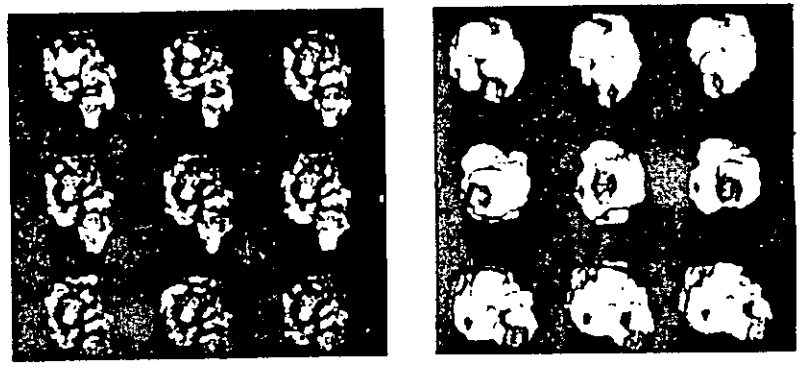
Simulating the real world is important in many aspects of training, whether in the existing game-like flight simulator software or in the simulation of surgical interventions. The development of Virtual Reality is an exciting prospect both for leisure and professional skill enhancement.

The design of a driving simulator has been presented by Zimmerman [106] using 25 x T800 processors with the objective of producing realistic images of scenes described by 600 polygons at 25 frames per second. The scan-line and depth buffering techniques for landscapes pose conventional problems, whilst simulating accidents and collisions with static and dynamic road-occupying objects at varying densities was seen as a more difficult task.

Image Presentation for the advertising industry has to produce images of very high quality, although not always in real-time. Conventional photo-realistic composition methods are used [107] within the framework of the RenderWoman Software, running on 8 x T800 processors. This system is quoted as achieving, at higher quality, in 2 to 3 weeks what has previously taken 3 to 4 months.

7.9 *Volume Rendering*

The use of transputer systems for volume imaging and ray tracing has proved a popular theme [108,109,110], the driving force being to achieve *interaction* with the rendered scene. Lomax [43,111] describes the use of a *single* transputer system in regular use for the diagnosis of cardiac motion disorders, particularly in the identification of infarcted tissue and aneurisms, whilst Cannon [112] generates 3D cardiac images from X-ray Computed Tomography. In Lomax's work, the original images are of blood pool distribution within the cardiac chambers as determined from time-gated Single Photon Emission Tomography, which can be displayed as the usual surface shaded views.



(a) (b)

Fig. 18 Phase Wavefront Flow (a), in red which starts in the top left-hand image and flows p of the ventricle before reappearing from the posterior surface of the atrium, and Wobbling sequ from cardiac blood pool images (tip of left ventricle has blue anti-phase shade and points gener ward)

The unique aspect of this implementation is the ability to view the images as parametric surfaces, indicating extent and phase of cardiac motion and to visualise the manner in which the contractile wave front, representing electric stimulus moves over the ventricular surfaces, Fig 18. Another unusual feature is that is performed on a transputer system interfaced to a simple BBC B microcomputer.

The CARVUPP (Computer Assisted Radiological Visualisation Using Parallel Processing) system is described as having similar objectives, i.e. interactive viewing, but with the addition of being able to alter the rendering characteristics. The implementation uses a 16 processor Meiko M10 system. An initial presentation [113] generates images from a 44 slice, 128 x 128 data set in 5 seconds using all processors. A greater degree of algorithmic detail is given in [114] where the application is extended to 256 x 256 images presented in pro-rata times. Whilst this, and many other systems, are seen as experimental, one highly acclaimed implementation in the surgical planning of facial bone replacement is in regular use [115,116,117]. The system allows the surgeon to plan corrective actions and produces the milling plan for the prosthesis which is to replace the excised or missing bone.

As a final example, in industry, the projected use of the Meiko M10 for volume visualisation of simulation of air-flow over aerofoil surfaces is reported by Payne [118], where the objective of replacing existing Fortran VAX code is considered. With 16 processors it is hoped to obtain rendered views at 5 - 10 frames per second.

7.10 Analysis of Texture

Novel methods of recognising features by their textural qualities are described by Smart [119,120]. The application is in the analysis of electromicrograph images of clayey soils. For 20 years this has been a manual process taking up to 15 hours per image. The images are examined for consistent domains, which are then analysed for directional trends. The initial approach concentrated on calculating intensity gradients within circular, elliptical or rectangular kernels to determine the existence of similar domains within each micrograph. The work was carried out on PC-based and Meiko transputer systems. In a recent development [121] the Hough transform has been used as a successful alternative in establishing directional characteristics of domains. Future work from the same team will examine Hough-based methods as domain detectors themselves.

7.11 Knowledge Processing

Whilst the majority of image processing can be said to be a precursor to a decision process, the representation of those processes in a formalised, competitive manner, using transputer processors as in a Blackboard architecture is described by Brown and Fisher [122]. This would appear to be an ideal use of MIMD in the emulation of the organised contributions of *experts* to generalised problem solving. The chosen test domain here is Canny edge detection, where each of the stages in the algorithm are treated as a knowledge unit. Using 64 transputers a speed-up of 21.2 times is achieved, although when the power / performance ratio is examined, the overhead of the Blackboard is considerable. The objective, however, is to enable evolution of classes algorithms which will more readily adapt to computational tasks.

8. Future Prospects

The opportunities which the new generation of T9000 transputers [123,124] offer stem largely from their order of magnitude improvement in processing and communication

speeds. The history of computing would suggest that improvements in power, however substantive, fail to satisfy requirements over more than just the short term. That the transputer has been taken up by the image processing community (inter alia) is undeniable, what is questionable is whether it is the correct type of processor for image processing tasks.

In a straightforward example, we can show that for image rotation using four nearest neighbour interpolation, a SUN IPX is equal to at least 8 x T800 processors in terms of execution speeds. It is also much quicker to code. By the time the T9000 is easily available the serial processor technology will have improved yet again. We have seen that there are classes of tasks, and natural forms of implementation, for which the data distribution bottle-neck and communication overheads reduce quite considerably the maximum achievable performance. Two approaches which would help to alleviate this problem and maintain the viability of transputer technology into the future are:

- Introducing additional alternative processors into the computational architecture,
- Addressing the problem of communication overhead between processors.

8.1 Additional Processors

Signal and image processing applications can often utilise the high performance advantages which are to be gained from Digital Signal Processing (DSP) methods. In a typical examination of the performance of an FFT algorithm by Dodge [125], saturation effects were seen to set in at 5 or more processors, and an attached processor solution proposed.

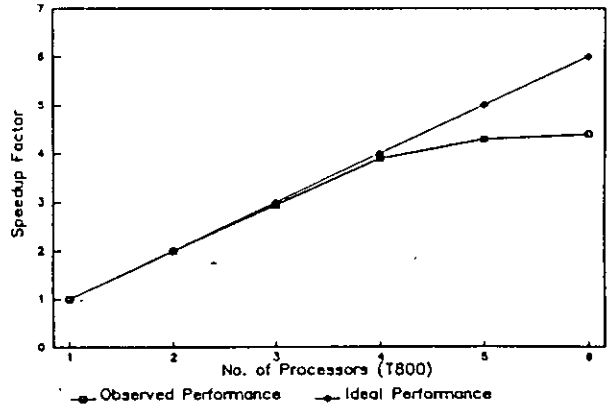


Fig. 19. FFT performance increase with number of processors.

The chosen hardware for the design of a composite processing architecture is the Plessey chipset (PDSP family) capable of allowing a *butterfly* operation every 100 nanosecs. Using a circuit such as the one in Fig. 20, the projected process rate given in Table 1 will be achieved. A T425 processor organises main memory and controls the DSP functions. Using switched memory banks, the DSP chip can be kept busy for image sizes of 256 x 256 and greater, and the internal precision is such that one dimensional transforms of 2000K points could, in principle, be achieved. The use of formal logic [126] was seen as a substantial advantage in the design of the circuit.

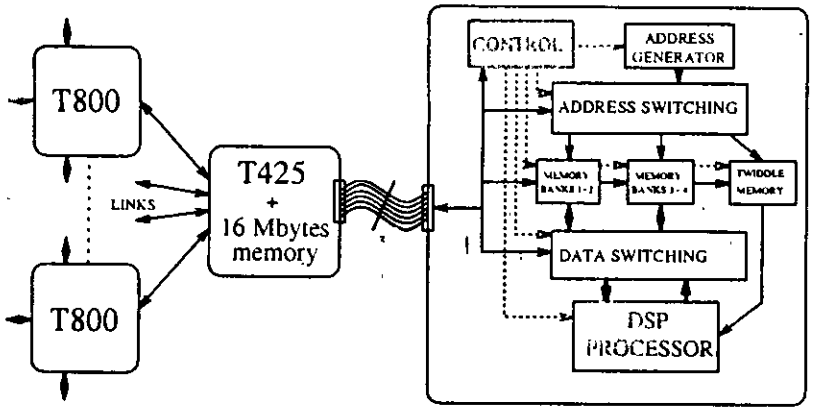


Fig. 20. FFT Accelerator System Diagram

8.2 Addressing the Communication Bottle-neck

Using data domain parallelism, it is desirable that processors start work on their own segments of data as soon as it is available regardless of the amount of data still needing to be distributed across the network. With increasingly large images, we observe initialisation stages which are highly significant when compared with the total processing time. A companion problem is that of passing results data back across the chosen topology when the task-load of each processor is dependent on the delivered data and *cannot* be reasonably anticipated. The edge following example quoted by Byrne [77] is a typical example, but the general problems of contour following or optic flow will also demonstrate these difficulties. For each processor to begin its tracking algorithm immediately it receives its region, it is necessary to support both *local* and *global* communication.

The ability to return results from anywhere in the network in a *fire-at-will* mode would reduce termination delays and also increase efficiency. Whilst the solution to a specific problem can be usually achieved by a complicated network of communicating processors as in Fig. 21, the work needed is a undesirable overhead for the *problem solving* as opposed to the *computing* community.

The underlying need is for an automatic mechanism for the general routing of messages across a network, not simply to the next nearest neighbour. The lack of a system kernel facility to support this kind of programming is one of the reasons transputer implementation of MIMD is often perceived as difficult and why expected performance increases are not always achieved. The use of an operating system, such as Helios or TransIDRIS, which supports virtual communication goes some way to resolving the problem, but at the expense of overheads and losing the simplicity of the OCCAM code.

The problems themselves are not a consequence of the CSP model, but rather of the implementation on to the available hardware. A better solution than the Operating System approach would be to remove the need for the programmer to handle the inter-processor routing restrictions imposed by the valency limitations of the transputer. This can be made possible by the implementation of *virtual* channels which allow the

unrestricted communication across intrinsically deadlock free networks whilst retaining the purity and convenience of OCCAM.

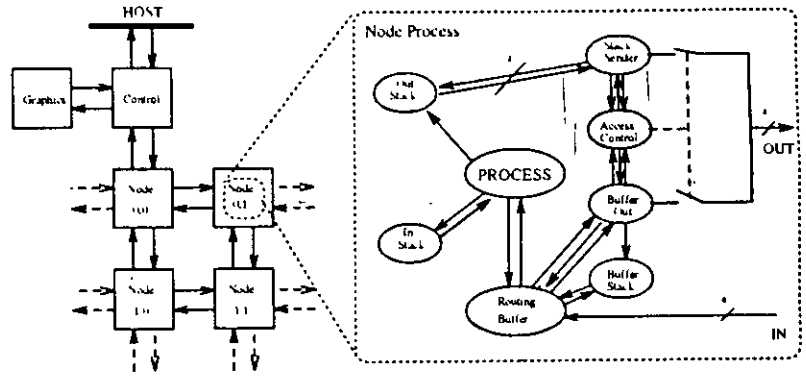


Fig 21. Schematic Approach to Global Message Passing

One of the perceived advantages for an image processing problem would be the removal of the need to arbitrate link access, as well as the beneficial consequences of control processes being considerably simplified by the use of dedicated channels. Processors can be asked to operate in a much less constrained manner. Another advantage would be the ability to place specialist peripheral processors, such as a graphics processors, in a network location which minimises communication traffic.

The T9000 design incorporates this philosophy within its hardware, although it may not be available on the initial release of the processor. Another approach which is being followed is the use of the simulation software. The Virtual Channel Router (VCR) [127,128], based on the virtual OCCAM compiler / configurer may well achieve the same objectives, and is the subject of on-going research investigations which aim to optimise placement strategies and improve the collection of run-time performance data as a necessary precursor to achieving the dynamic allocation of processors to tasks of variable and unpredictable complexity.

9. Conclusion

Searching through the Science Citation Index for journal references is less productive than initially anticipated. Most of the published material is in Conference Proceedings, predominantly those organised by the SERC/DTI Transputer Initiative and the Occam User Group and these submissions have increased steadily. Over the last four years an average of only 10 - 15 papers in learned journals bear the term *transputer* and a similar number include *parallel* and *image*. Only 5 or so per annum include the term *MIMD*. These numbers have remained remarkably constant over the period.

From a simple appraisal of the two most recent major UK conferences sponsored by the SERC/DTI Transputer Initiative, TA'90 and TA'91, the number *industrial* or *commercial* organisations publishing their applications of image processing using

transputers is limited [6 out of 19 in 1990 and 2 out of 29 in 1991]. Some comments are relevant here. Firstly, some of the work will have been done as part of an industrial contract to academic institutions, secondly conference and publications in scientific literature probably do not reflect the true extent of industrial usage, and finally, at present, the power / cost ratio of the transputer family is not particularly attractive. When coupled with the perceived difficulty of translating existing algorithms, not only into parallel code, as well as into an environment which places restrictions on the flexibility of message passing, a limited involvement in *industrial solutions* has been apparent [129]. This is in marked contrast with the wide range of transputer assisted image processing hardware which is available, and the enthusiastic take-up by the academic sector. It is hoped that the improvements to the technology reported earlier will do much to alter this situation.

The SERC/DTI Transputer Initiative, in a response to several of these issues set up three Transputer Application Community Clubs (TACCs) of which the image processing group (IPTACC) has played a significant role in coordinating conferences and producing surveys of image processing hardware and software. On the educational side it has organised practical workshops at conferences and sponsored seminars on image processing topics.

In a review of the transputer, past and present, [130], future applications are likely to have a significant bearing on image processing and visualisation. Typical opportunities are identified as being in hand held navigation devices, terrain mapping, virtual reality and long-range radar, with a concentration on embedded applications in a micro-packaging environment which is atypical of many of today's visualisation devices. Once we know how to produce, and interact with, high quality images in hand-held boxes, the new transputer products should be extremely attractive from the hardware viewpoint.

Nevertheless we should ask whether there is more to their cost-effective uptake than hardware viability. The transputer started life as a physically small, embedded processor, able be programmed to support a wide variety of special tasks. It metamorphosised into a wide spectrum of PC and workstation accelerators when it became clear that the (then) popular PC had rather limited processing power. What we really need is a general parallel paradigm into which the transputer can easily fit.

Is the problem of the take-up of any novel hardware essentially one of software support [131,132]? Many forms of novel hardware have evolved rapidly in the past five years, the transputer is a typical example of what happened at the beginning of that period. The audience for high performance computing is not the computer scientist , but the experimental scientist, the engineer and other technical programmers whose tasks exceed the capacities of the fastest sequential systems. A typical quote is:

> *"Nobody wants parallelism, what we all want is performance!"*

It is the fact that going to parallelism is the only way to achieve this that makes it a necessity [133]. Parallelism has been with us in some form since the mid-1970s. The reasons for a sparsity of *production level* environments depends on the viewpoint of the individual user.

Researcher — Many modern problems require complex application to run in reasonable time. Users want to further their science, but we still need technical specialists, and that makes for a slow development cycle.

Programmer — Whilst academic successes in parallelisation of code of 5,000 - 10,000 lines for a particular machine produced by a single group are reported, commercial program suites are often much larger and generated by different development groups to an overall design plan. Often the opportunities for parallelism are limited and a new sequential processor changes the viability of the whole exercise.

Manager — Generally interested only in the development cycle and the impact on the learning curve of using the software and getting it into service.

Parallel software is inherently more complex because of its increased dimensionality. We have seen examples above where constraining the degree of flexibility radically reduces the speedup. In general if we increase the speed of a serial processor by a given factor, our program runs that much faster. Even if we did not have to think about communication and architectures, we would still have the multidimensional problems.

What is needed are some standards for parallel computing across a broad scientific front. In some areas, notably numerical algorithms (BLAS), this may have been achieved. With the increased dimensionality of programming there needs to be new methods of visualising data, performance and algorithms, perhaps multidimensional imaging and graphics can help here. Unless there is stability of architectures, and the transputer scores well in this respect, *and* an investment in software, hardware will get more complex and more wastefully used.

We also need to pay attention to the user environment. Even in the atypical case of a University computing community a recent survey showed that only 7% of users were doing developmental programming work. Perhaps what we need is more attention to software templates rather than programming languages.

There remains the question of what is the most appropriate architecture. We have seen the problems which the local memory aspect of transputers brings to image processing, but there are also instances where distributed local memory is beneficial. Nevertheless, large shareable memory spaces would often make life a bit simpler for the image processing specialist. Perhaps there is a need for heterogeneous computing [134], which is then developed into a homogeneous whole from the users' viewpoint.

Parallel computing, in one or more paradigms, will continue to grow rapidly in the future, and image processing, probably as a precursor to computer vision will be at the forefront of those disciplines requiring the power parallel computing provides. In order to survive, transputer methodology and its software support must be able to:

- Handle the prospect of very large numbers of processors with the consequent difficulties of indeterminacy and the need to access shared memory. Eventually data transfer and computation will become a single operation.
- Compete against increasingly cost-effective fine grained technology which has a intrinsic attraction for image processing,
- Provide *programmers' assistants*, capable of teasing out what a programmer wants to do and thereby optimise on parallelism and data flow.
- Provide confidence building robustness, satisfy user expectations, adhere to and help to establish standards, and finally,
- Appeal to the non-academic as well as the academic world in respect of its flexibility, reliability and long-standing support.

These are the issues which the new technology must address and resolve.

Acknowledgements

This review owes much to many colleagues. In particular, to fellow members of the SERC/DTI Transputer Initiative IPTACC Executive especially to Mike Jane, Terry Mawby and Peter Smart of the Secretariat, to Phil Ross, Chris Dodge, John Byrne, Tony Lomax and Raymond Hutcheon from the Department of BioMedical Physics, University of Aberdeen, and to the SERC, MRC, Professor John Mallard, the University of Aberdeen and the Grampian Health Board for their encouragement and support via equipment, resources and research studentships.

Inevitably there will be many workers in transputer based projects in image processing who have not been mentioned, some of whom may have achieved objectives and results far in advance of those reported on here. To all of these there are sincere apologies.

References

[1] Ballard D.H. and Brown C.M., "Computer Vision" Prentice Hall, Englewood Cliffs, NJ, 1982.

[2] Pratt W.K., "Digital Image Processing" Wiley, New York, 1978

[3] Rosenfeld A. and Kak A.C., "Digital Picture Processing", Vols 1 & 2, Academic Press, New York, 1982.

[4] Niemann H., "Pattern Analysis and Understanding" Springer, Berlin, 1990.

[5] Wyszecki G. and Stiles W.S., "Color Science - Concepts and Methods, Quantitative Data and Formulae" 2nd Edition, Wiley, New York, 1982.

[6] Jayant N.S and Noll P., "Digital Coding of Waveforms", Prentice Hall, NJ, 1984.

[7] Wintz P A., "Transform picture coding", *Proc IEEE*, 60 (1972) 809.

[8] Kunt M, Benard M and Leonardi R., "Recent results in high compression coding", *IEEE Trans* CAS - 34 (1987) 1306 - 1336

[9] Wolberg G., "Digital image warping", IEEE Computer Soc. Press, Los Alamitos, 1990, pp 117 - 162.

[10] Rosenfeld A. and Kak A C., "Digital Picture Processing", Vol 1, Academic Press, New York, 1982, pp 231 - 237

[11] Hummel R A., "Histogram equalisation techniques", *Comput. Graphics Vis. and Image Proc.* 4 (1975) 209.

[12] Pizer S M. Amburn E.P. Austin J.D. et al., "Adaptive histogram equalisation and its variations" *Comput. Graphics Vis. and Image Proc.* 39 (1987) 355 - 368.

[13] Barnea D.I and Silverman H.F., "A class of algorithms for fast digital image registration", *IEEE Trans. Comput.*, C-24 (1975) 935

[14] Wolberg G., "Digital Image Warping", IEEE Computer Soc. Press, Los Alamitos, 1990, pp 41 - 92.

[15] Arun K.S., Huang T.S. and Bostein S.D., "Least squares fitting of two 3D point sets", *IEEE Trans Patt. Anal. Mach. Int*, PAMI-9, (1987) 698 - 700.

[16] Oppenheim A.V., "Applications of Digital Signal Processing", Prentice Hall, Englewood Cliffs, NJ, 1978

[17] Nussbaumer H.J., "Fast fourier transform and convolution algorithms", 2nd Edition, Springer Ser. Inf Sci., Vol 2, Springer Berlin, 1981.

[18] Brigham E.O., "The fast fourier transform and its applications", Prentice Hall, Englewood Cliffs, NJ, 1988

[19] Andrews H.C. and Hunt B.R., "Digital Image Restoration", Prentice Hall, Englewood Cliffs, NJ, 1975.

[20] Canny J.F., "A computational approach to edge detection", *IEEE Trans. Patt Anal. Mach. Int.*, PAMI-8 (1986) 679.

[21] Pitas I. and Venetsanopoulous A.N., "Non-linear order statistics filters for image filtering and edge detection", *Signal Processing*, 10 (1986) 395.

[22] Serra J., "Image Analysis and Mathematical Morphology", Academic Press, London, 1982.

[23] Haralick, R.M., Sternberg S.R. and Zhuang X., "Image analysis using mathematical morphology", *IEEE Trans Comput.*, C-18, (1987) 733.

[24] Sohoa D., "A survey of thresholding techniques", *Comput. Graphics Vis. and Image Proc.*, 41, (1988) 233 - 260.

[25] Elliot H. and Srinivasan L., "An application of dynamic programming to sequential boundary estimation", *Comput. Graphics Vis. and Image Proc.*, 17 (1981) 291

[26] Pavlidis T., "A review of algorithms for shape analysis", *Comput. Graphics Vis. and Image Proc.*, 7 (1978) 243

[27] Duda O. and Hart P.E., "Use of the Hough transform to detect lines and curves in pictures", *Commun. ACM* 11 (1972).

[28] Ballard D.H., "Generalising the Hough transform to detect arbitrary shape", *Pattern Recognition* 13 (1981) 11.

[29] Zucker S.W., "Region Growing. Childhood and adolescence", *Comput. Graphics Vis. and Image Proc.*, 5 (1976) 382.

[30] Ohlander R., Price K. and Reddy D.R., "Picture segmentation using a recursive region splitting method", *Comput. Graphics Vis and Image Proc.*, 8 (1978) 313.

[31] Horowitz S.L. and Pavlidis T., "Picture segmentation by a tree traversal algorithm", *J. Assoc Comput. Mach*, 23 (1976) 368,.

[32] Haralick R A., "Statistical and structural approaches to texture", In: *Proc 4th Int'l Joint Conf on Pattern Recognition*, Kyoto, 1978, pp 45 - 69

[33] Hawkins J K., "Textural properties for pattern recognition" In Picture Processing and Psychopictorics, Ed B S. Lipkin & A Rosenfeld, Academic Press, New York, 1970, pp 347 - 370

[34] Flynn M.F., "Some computer organisations and their effectiveness", *IEEE Trans. Comput* C-21 (1972) 948 - 960

[35] Sharp J A., "An Introduction to Distributed and Parallel Computing", Blackwell Scientific Press, Oxford, 1987 pp 32 -52

[36] Harp G., Baker S. and Weber H., "Image Processing", In: Transputer Applications, Ed G Harp, Pitman London, 1989, pp 170 - 203

[37] Hey A., "Scientific Applications" In: Transputer Applications, Ed G. Harp, Pitman, London, 1989, pp 204 - 220

[38] Browne R F. and Hodgson R.M., "Mapping image processing operations onto transputer networks", *Microprocessors and Microsystems* 13, 3 (1989) 203 - 211

[39] Byrne J.P., "Investigation of parallel computing techniques in clinical image processing", Ph.D. Thesis, University of Aberdeen, 1992.

[40] Gordon G T., "Image Reconstruction From Projections, the fundamentals of computerised tomography", Academic Press, New York, 1970.

[41] Kingswood N., Dagless E L., Belchamber R.M., Betteridge D., Lilley T. and Robertson J.D M., "Image Reconstruction using the transputer", *IEE Proc.* 133, No 3, 139 - 144 (1989).

[42] Kellett G. and Durrani T S., "Parallel Algorithms for Tomographic Image Reconstruction", In: *Proc. 3rd Intl. Conf. on Image Processing and its Applications*, IEE Press, London 1989, pp 178 - 181.

[43] Lomax A.J., "Methodologies for the 3D display of Volumetric data and implementation on a parallel processing system", Ph.D Thesis, University of Aberdeen, 1992.

[44] Jones, G., "Efficient multiple buffering in occam", *Occam User Newsletter*, 11 (1989) 36.

[45] Image Processing Transputer Applications Community Club (IPTACC), Directory, *SERC/DTI Initiative in the Engineering Applications of Transputers*, Ed. T. Mawby, June 1991.

[46] Cook R.S , Geerstenberger, J., Lawrence C. and Neamtu H., "Applications of a parallel image processor", In: *Proc 1st WoTUG Conf., 'Transputing 91'*. Ed Welch, Styles, Kunii & Bakers, IOS Press Amsterdam, 1991,pp 15.

[47] Harp J G. and Weber H.C., "Image processing on the reconfigurable transputer processor", In: *Proc 7th OUG Technical Meeting*, IOS Press, Amsterdam, 1990

[48] Hirsch E., Paillou, P. , Muller, C. and Guggenbach V., "A versatile parallel computer architecture for machine vision", In: *Proc 1st WoTUG Conf., 'Transputing 91'*. Ed. Welch, Styles, Kunii & Bakers, IOS Press Amsterdam, 1991, pp 828

[49] Edwards J. and Sillitoe, I., "The design of a real time 3-D vision system for object identification", In: *Proc. 12th OUG Technical Meeting*. Ed. S.Turner, 1990, pp 198.

[50] Seed N L., Houghton A.D., Lander M.J. and Goodenough N.J., " An enhanced transputer module for real time image" In: *Proc. 3rd Intl. Conf. on Image Processing and its Applications*, IEE Press, London 1989, pp 131 - 135.

[51] Ross P.G.B., Lomax A.J. and Undrill P E., "Medical image processing using transputer based hardware", In: *Proc. 3rd Intl. Conf. on Image Processing and its Applications*, IEE Press, London 1989, pp 171 - 177.

[52] Byrne J.P., Ross P.G.B., Undrill P E. and Phillips R.P., "Feature based retinal image registration using transputers", In: *Applications of Transputers 3*, Ed. T.S. Durrani, W.A. Sandham, J.J. Soraghan & S.M. Forbes, IOS Press, Amsterdam, 1991, pp 687 - 692.

[53] Lomax A J., Ross P.G.B. and Undrill P.E., "Parallel processing techniques for the interactive display of volumetric data sets", In: *Applications of Transputers 3*, Ed. T.S Durrani, W.A. Sandham, J J. Soraghan & S M. Forbes, IOS Press, Amsterdam, 1991, pp 136 - 141.

[54] Smart P , "Commercial Software for Image Processing by Transputer", *SERC/DTI Transputer Initiative Mailshot*, Ed. T. Mawby, April 1991, pp 72 - 76.

[55] Crookes D., Morrow P.J and McParland P , "IAL , A parallel image processing language", *IEEE Proc. I*, 137, No. 3, (1989) 176 - 182.

[56] Morrow P.J., Crookes D. and Kirkpatrick, P C , Milligan P and Scott N.S., "A comparison of two notations for programming image processing applications on transputers" In: Developments using OCCAM, *Proc. 8th* Technical Occam Meeting Ed. J.Kerridge, 1988, pp 1 - 9.

[57] Crookes D., Morrow P.J., Sharif B, and McClatchey, I , "An environment for developing concurrent software for transputer based image processing", *Microprocessing and Microprogramming*, 27 (1989) 417 - 422.

[58] Morrow P.J. and Perrott R.H., "The design and implementation of low-level processing algorithms", In: *Parallel Architectures and Computer Vision*, Oxford Science Publications, 1990, pp 243 - 2650.

[59] Dew P.M, Wang H. and Webb J.A., "Apply : Machine independent image processing language and its implementation on a Meiko Computing Surface", In:*Proc 5th Alvey Vision Conf.*, 1989, p 309.

[60] Henderson T.B., Symanski J.J. and Bromley K , "Software development on the video transputer array", In: *Proc 1st NATUG Conf.*, IOS Press, Amsterdam, 1990.

[61] Rygol M.R., Pollard S, Brown C. and Kay J., "MARVIN & TINA: A multi-processor 3D vision system,In: *Applications of Transputers 2*. Ed. D.J. Pritchard and C.J.Scott, IOS Press, Amsterdam, 1990, pp 218 -225

[62] Rygol M R., Pollard S, Brown C.,"A Multiprocessor 3D Vision system for pick and place", In:*Proc. British Machine Vision Club, University of Sheffield Press*, (1990) pp 169 -174.

[63] Brown C. and Rygol M., "Marvin: Multiprocessor architecture for vision",In. *Proc. 10th Occam User Group Meeting*, IOS Press, Amsterdam 1989.

[64] Khan G.N. and Gillies D , "Transputer based implementation of as parallel machine vision method for endoscope navigation", In: *Applications of Transputers 3*, Ed T.S. Du ..., W.A. Sandham, J.J. Soraghan & S M. Forbes, IOS Press, Amsterdam 1991 pp 699 - 70..

[65] Fairhurst M.C., Abdel Wahab H.M.S. and Brittan P.S J, "Parallel implementation of image classifier architectures using transputer arrays", In: *Proc. 3rd Intl. Conf. on Image Processing and its Applications*, IEE Press, London 1989 pp 136 - 140

[66] Fairhurst M.C., Brittan P. and Cowley K D , "Transputer based models for the parallel implementation of image classification algorithms", In *Applications of Transputers 3*. Ed. T.S. Durrani, W A Sandham, J.J Soraghan & S.M. Forbes, IOS Press, Amsterdam 1991 pp 559 - 564.

[67] Tregidgo R.W S and Downton A.C., "Scalable parallelism for embedded vision applications: The generalised tree pipeline", In: *Applications of Transputers 3*, Ed T S Durrani, W.A. Sandham, J.J. Soraghan & S M. Forbes, IOS Press, Amsterdam 1991, pp 383 -387.

[68] Lotuto R.A., Morgan A D , Johnson A.S. and Thomas B.T , "A transputer based automatic number-plate recognition system",In: *Applications of Transputers 2*, Ed. D.J. Pritchard and C.J Scott, IOS Press, Amsterdam 1990.

[69] Lades M, Vorbruggen J C and Wurtz R P , "Recognising faces with a transputer farm",In :*Applications of Transputers 3*, Ed. T S Durrani, W.A. Sandham, J.J. Soraghan & S.M. Forbes, IOS Press, Amsterdam 1991 pp 148 -153.

[70] Morrow P.J. and Crookes D , "Parallelising an Image segmentation and analysis system for infrared images", In: *Applications of Transputers 3*, Ed T.S. Durrani, W.A. Sandham, J.J. Soraghan & S M. Forbes, IOS Press, Amsterdam 1991 pp 377 - 382.

[71] Lin W. and Fraser D.A , "Identification of 3D objects from 2D objects",In: *Applications of Transputers 2*. Ed D J. Pritchard and C.J Scott, IOS Press, Amsterdam 1990 pp 203 - 208.

[72] Holden M., Zennerly M J. and Muller J-P , "Using a transputer array parallel stereo matching of SPOT satellite images", In *Applications of Transputers 3*, Ed. T.S. Durrani, W A Sandham, J.J. Soraghan & S.M. Forbes, IOS Press, Amsterdam 1991 pp 154 - 159

[73] Curtis K M and Bourindane A "A parallel processing engine for n-tuple pattern recognition",In: *Applications of Transputers 2*, Ed D J Pritchard and C J Scott, IOS Press, Amsterdam 1990 pp 233 - 239

[74] Netherwood P., Barnwell P. and Forte P , "An automated visual inspection system for surface mount assemblies and solder joints" In: *Applications of Transputers 3*, Ed T.S. Durrani, W.A. Sandham, J J. Soraghan & S M. Forbes, IOS Press, Amsterdam 1991 pp 530 - 535

[75] Mirmehdi M , "Product label inspection using transputers", In. *Applications of Transputers 2*, Ed. D.J. Pritchard and C.J.Scott, IOS Press, Amsterdam 1990 pp 408 - 416.

[76] Yeo K.P , Cheung C.C., Ord A and Brown W.A., "Determination of Rock Fragments using a transputer array", In: *Applications of Transputers 3*, Ed. T.S Durrani, W.A. Sandham, J.J: Soraghan & S M. Forbes, IOS Press, Amsterdam 1991 pp 142 - 147

[77] Ali A.T. and Dagless E.L., "Automatic traffic monitoring using transputer image processing system", In *Applications of Transputers 2*, Ed. D.J. Pritchard and C.J.Scott, IOS Press, Amsterdam 1990 pp 209 - 210

[78] Wang H Brady M. and Page, I., "A fast implementation for computing optic flow and its implementation on a transputer array" In: *Proc. British Machine Vision Club*, University of Sheffield Press, 1990 pp 175 - 179.

[79] Chalabi N.A. and Durrani T S., "Transputer applications to the image flow field" In: *Applications of Transputers 3*, Ed T S Durrani, W A Sandham, J.J. Soraghan & S M. Forbes IOS Press, Amsterdam 1991 pp 394 - 399

[80] Ellis T.J., Hung T.W.R and Omarouayache S., "Structural elements in colour images: A parallel approach", In: *Applications of Transputers 3*, Ed. T.S. Durrani, W A. Sandham, J J Soraghan & S M. Forbes, IOS Press, Amsterdam 1991 pp 536 - 537

[81] Stephen G.A., Taylor C.A. and Dagless E.L, "Real time analysis for dynamic displacement measurement", In: *Applications of Transputers 2*, Ed. D.J. Pritchard and C.J.Scott, IOS Press, Amsterdam 1990 pp 211 - 217.

[82] McAllister H.G, Ayre J. and McCullagh P.J., "Topographic brain electrical potential mapping", In: *Applications of Transputers 3*, Ed. T.S. Durrani, W.A. Sandham, J.J. Soraghan & S.M. Forbes, IOS Press, Amsterdam 1991 pp 524 - 529.

[83] Doan D., Hulskamp J. and Maher K., "An application of transputers in digital fluoroscopy", In: *Applications of Transputers 3*, Ed. T.S. Durrani, W.A. Sandham, J.J. Soraghan & S M. Forbes, IOS Press, Amsterdam 1991 pp 693 - 697.

[84] Cao X, Deravi, F and Rodd G., "Parallel implementation of the tuned Hough transform on transputer networks", In. *Applications of Transputers 1*', IOS press, Amsterdam, 1989.

[85] Lotuto R.A , Dagless E.L., Milford D.J., Morgan A.D., Morrissey J.F. and Thomas B.T., "Hough transform for transputer arrays", In: *Proc. 3rd Intl. Conf. on Image Processing and its Applications*, IEE Press, London 1989, pp 122 - 130.

[86] Costa L-da-F., and Sandler M.B., "Implementation of the binary Hough transform in a pipelined multi-transputer architecture", In: *Applications of Transputers 2*, Ed. D.J. Pritchard and C.J.Scott, IOS Press, Amsterdam 1990 pp 150 -155.

[87] Taylor D R., "Implementing the Hough transform in parallel using a transputer network", M.Sc thesis, University of Aberdeen, 1989.

[88] Eghtesadi S. and Sandler M., "Implementation of the Hough Transform for intermediate low level vision on a transputer network", *Microprocessors and Microsystems* 13, No. 3, (1989) 212 - 218

[89] Yang O and Hodgson D.C., "Real-time motion tracking using the Hough Transform", In: *Applications of Transputers 3*, Ed T.S. Durrani, W.A. Sandham, J.J. Soraghan & S.M. Forbes, IOS Press, Amsterdam 1991 pp 548 - 553

[90] Hall G., Terrel T.J., Senior J.M. and Murphy L.M , "A new discrete radon transform enhancing linear features in noisy images", In' *Proc. 3rd Intl Conf on Image Processing and its Applications*, IEE Press, London 1989, pp 187 - 191.

[91] Hall G , Terrel T.J., "Implementation of the Radon transform using a dynamically switched transputer network, In: *Applications of Transputers 2*, Ed. D J. Pritchard and C.J.Scott, IOS Press, Amsterdam 1990, pp 156 - 163.

[92] Barrett S., Ballini D , Del Guerra, A., "Use of a transputer system for fast 3D reconstruction in 3D PET", *IEEE Trans Nucl Sci* NS - 37 (1990) 812 - 816

[93] Wilkinson N , Atkins, M.S. and Rogers J.G , "A tomograph parallel processing data acquisition system", *IEEE Trans. Nucl. Sci.* , NS - 36. (1990) 1047 - 1051.

[94] Atkins, M.S., Murray D. and Harrop, R., "Use of transputers in a 3D Positron Emission Tomograph", *IEEE Trans. Med. Imag.* MI-10, 3, (1991) 276 - 283, .

[95] Zadehkoochak M., Hames, T.K., Blott B H. and George R F., "A transputer implemented algorithm for electrical impedance tomography", *Clin. Phys. & Physiol. Meas.* 11, 3, (1990) 223 - 230.

[96] Zapata E.L., Benavides I., Bruguera J.D. and Carazo J.M , "Image reconstruction on transputer networks", In: *Applications of Transputers 2*, Ed. D.J. Pritchard and C.J.Scott, IOS Press, Amsterdam 1990, pp -164 -171.

[97] Mukai R., "Parallel processing of quadtree images", In *Proc. 3rd Japan OUG Intl. Conf.*, IOS Press, 1990.

[98] Hull M.E.C., Frazer J.H. and Millar R.J , "Transputer based manipulation of computed tomography images",pp 146 - 149.

[99] Hull M.E C., Frazer J.H and Millar R.J., "Octree modelling of computed tomography images", *IEE Proc I*, 137, 3, (1989) 118 - 122.

[100] Cook, R. and Sandys-Renton J., "A parallel segmentation algorithm (Merge using Moments) fror SAR images", In. *Applications of Transputers 3*, Ed. T.S. Durrani, W.A. Sandham, J.J. Soraghan & S.M. Forbes, IOS Press, Amsterdam 1991 pp 311 316.

[101] Crookes D., Morrow P.J. and Philip G., "The development of a transputer based image data base", In: *Applications of Transputers 2*, Ed. D.J. Pritchard and C.J.Scott, IOS Press, Amsterdam 1990 pp 189 - 195.

[102] Cong M N, Rudberg B., Soraghan J J , Durrani T.S. and Marshall S., "Transputer based image codec for local area network", In: *Applications of Transputers 3*, Ed T S. Durrani, W.A. Sandham, J.J. Soraghan & S.M. Forbes, IOS Press, Amsterdam 1991 pp 542 - 547.

[103] Chong, M N., Soraghan J.J. and Durrani, T.S., "Parallel implementation and analysis of adaptive transform coding algorithms", In. *Proc IEEE Intl. Conf. in Acoustics, Speech and Signal Processing (ICASSP 90)*, IEEE Press, 1990, pp 997 - 1000.

[104] Chong M.N, Soraghan J.J. and Durrani T S., "Pipelined functional algorithms, data partitioning for adaptive transform coding algorithms", *IEE Colloquium 'Parallel Architectures for Image Processing Applications'*, IEE Press, London, 1991.

[105] Griffiths J W R., Chia L T. and Lu G.J., "Image compression using iterated function systems", In: *Applications of Transputers 3*, Ed. T.S. Durrani, W A Sandham, J.J. Soraghan & S.M. Forbes, IOS Press, Amsterdam 1991 pp 554 - 557.

[106] Zimmerman P., "Visual simulation by means of a transputer network for a driving simulator", In: *Applications of Transputers 2*, Ed. D.J. Pritchard and C.J.Scott, IOS Press, Amsterdam 1990 pp 13 - 24

[107] Tricot C , and Fest J-M , "Image synthesis for television on a Volvox transputer based machine", In. *Applications of Transputers 2*, Ed D.J. Pritchard and C.J Scott, IOS Press, Amsterdam 1990 pp 29 - 33

[108] Hulskamp W., Elgerhuizen, P M ,Langenkamp A.A.J and van Lieshout, P.L.J., "Visualisation of 3D empirical data: The Voxel Processor", In. *Proc 10th OUG Technical Meeting*, Ed Bakkers, IOS Press, 1989, p 82

[109] Kawai T , Ohnishi, M., Abeki J. and Ohnishi H., "Ray tracing for parallel image generation", In: *Proc 4th Australian and OUG Conf* , Ed Bossomier, Hintz and Hulskamp, IOS Press, Amsterdam, 1991 , pp 89.

[110] Purgathofer W and Zeiller M , "Configuring transputers for ray tracing", In *Applications of Transputers 1*, IOS Press, Amsterdam, 1990

[111] Lomax A J and Undrill P.E., "Interactive 3 dimensional and dynamic display of gated blood pool tomograms" In. '3D Imaging Techniques for Medicine', *IEE Digest* 1991/083 pp 7/1 - 7/4.

[112] Cannon S R. and Allan S.J., "A parallel processing sub-system for the generation of 3D cardiac images from CT", In *Proc 3rd* NATUG Conf 'Transputer Research and Applications 3', Ed. S. Wagner, IOS Press, 1990, pp 75

[113] Tyrell J., Farzad Y., Riley M and Winterbottom N , "CARVUP Computer Assisted Radiological Visualisation Using Parallel processing",In *Applications of Transputers 2*, Ed. D J. Pritchard and C.J.Scott, IOS Press, Amsterdam 1990, pp 172 181.

[114] Tyrell J., Farzad Y , Riley M. and Winterbottom N., "CARVUP Computer Assisted Radiological Visualisation Using Parallel processing", In 3D Imaging in Medicine', Eds K H. Hohne, H Fuchs and S M. Pizer, Nato ASI Series F', Vol. 60, 1990 pp 363 - 375

[115] Tan A C., Richards R and Linney A.D , "3D medical graphics - Using the T800 transputer,In: *'Developments in using OCCAM', Proc. 8th OUG*, IOS, Amsterdam 1988 pp 83 - 89

[116] Linney A D., Grinrod S R , Arridge S.R. and Moss J.P , "Three dimensional visualisation of computerised tomography and laser scan data for the simulation of maxillo-facial surgery" *Med Informatics* 14 (1989) 109 - 121 .

[117] Tan A C, and Richards, R , "Developing the MGI Workstation - a multi transputer based medical graphics system"In: *Proc 1st WoTUG Conf., 'Transputing 91'*. Ed. Welch, Styles, Kunii & Bakers, IOS Press Amsterdam, 1990 ,pp 801.

[118] Payne B J , "3D Parallel visualisation", In: *Applications of Transputers 2*, Ed. D.J. Pritchard and C.J.Scott, IOS Press, Amsterdam 1990 pp 182 - 188.

[119] Smart P. and Leng X., "Textural analysis by transputer", In: *Applications of Transputers 3*, Ed. T.S. Durrani, W.A. Sandham, J.J. Soraghan & S.M. Forbes, IOS Press, Amsterdam 1991 pp 240 - 247.

[120] Smart P. and Leng X., "Improved methods of Textural analysis", In. *Applications of Transputers 3*, Ed. T.S. Durrani, W.A. Sandham, J.J. Soraghan & S M. Forbes, IOS Press, Amsterdam 1991 pp 323 - 328.

[121] Costas L da F. Leng X, Smart P and Sandler M.B., "Analysis of clay microstructure by transputers", In: *Applications of Transputers 3*, Ed. T.S. Durrani, W A. Sandham, J.J. Soraghan & S.M Forbes, IOS Press, Amsterdam 1991 pp 317 - 322.

[122] Brown M.D. and Fisher R.B., "A distributed blackboard system for vision applications" In: *Proc. British Machine Vision Club*, University of Sheffield Press, 1990 pp 163 - 168.

[123] "The T9000 Transputer Products Overview Manual", Inmos Databook Series, 1991.

[124] Dyson C., "Inmos H1 architecture revealed", New Electronics, 23, 8 (1990) 21 -24 .

[125] Dodge C.J., Ross P.G.B. and Undrill P.E., "A transform accelerator for a transputer system", In: *Proc Parallel Hardware for Signal and Image Processing*", British Computer Society (in press)

[126] Dodge C.J., Ross P.G.B., Allen A.R. and Undrill P.E., "Formal methods in the design of an FFT accelerator for a transputer based image processing system", *Medical & Biol. Eng. and Comput.*, 29, Supp. Pt. 1, (1990) 91.

[127] Debbage M., Hill M. and Nicole D., "The virtual channel router" (Release Notes) Transputer Technology Solutions, University of Southampton 1991

[128] Pountain D , "Virtual Channels: The next generation of transputer", *Byte (European & World Edition)* 15, (1990) 4, 3 -12,.

[129] New Scientist, 130, (1991) 1766, 29.

[130] Whitby-Stevens, C., "Transputers past and present", *IEE Micro* 10, 6, (1991) 16

[131] Pancake, C M , "Software support for parallel computing : where are we headed", *Comm. ACM* 34 (1991) 11, 63 - 54.

[132] Hack J.J., "On the promise of general purpose parallel computing", *Parallel Computing*, 10, 3, (1989) 261 - 275.

[133] Simon H.D , "Are highly parallel systems ready for prime time", *Intl J. Supercomput. Appl.* 4,(1990) 90.

[134] Nudd, G R , Atherton T J., Howarth R.M, et al., "Wpm: A multiple-simd architecture for image processing", In: *Proc. 3rd Intl. Conf. on Image Processing and its Applications*, IEE Press, London, 1989, pp 161 - 165.

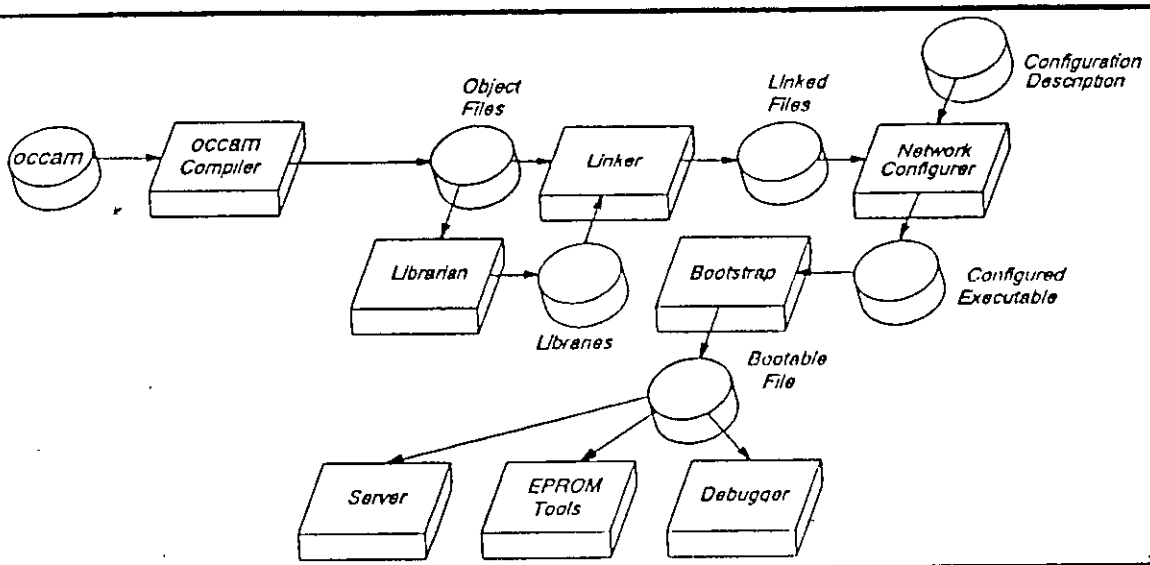
APENDICE

OCCAM 2 Toolset

IMS D7305
IMS D4305
occam 2 Toolset

Translech Parallel Systems Corp.
20 Thornwood Drive
Ithaca, NY 14850-1263
Tel: (607) 257-6502
Fax: (607) 257-3980

Product Information



KEY FEATURES

- occam compiler for INMOS transputers
- Excellent compile time diagnostics
- Optimization of object code
- Tools to support separate compilation (linker, librarian)
- Tools for creating and loading multiprocessor programs
- Support for all first generation INMOS transputers (T2xx, T4xx and T8xx)
- Breakpointing and post-mortem symbolic debugger for multiprocessor programs
- Automatic make file generator
- T425 transputer simulator
- Support for assembler inserts in occam
- Support for EPROM programming
- Automatic routing of channels through a transputer network
- Listings of where variables and functions reside in memory

KEY FEATURES

- Support for placement of code and data in user specified memory locations
- Small runtime overhead
- Consistent tools across PC and Sun 4 hosts
- Support for mixing occam with C code compiled with the D7314 and D4314 toolsets
- Support for dynamically loading programs and functions
- Scientific libraries
- Compatibility with INMOS INQUEST advanced development environment

DESCRIPTION

The INMOS Professional occam 2 Toolsets provide a complete high quality software development environment for all T2, T4 and T8 series transputers. The toolsets enable parallel programs to be built for single transputers or multi-transputer networks comprising a mixture of transputer types. Networks of processors are supported by a software through-routing configurer that greatly simplifies the placement of code and communication paths in complex applications. Improved embedded application support is provided by both configuration and symbol map utilities.

SGS-THOMSON MICROELECTRONICS
INMOS is a member of the SGS–THOMSON Microelectronics Group
42 1600 01
March 1993

Contents

1 Introduction

The INMOS Professional occam 2 Toolsets provide a complete high quality software development environment for all T2, T4 and T8 series transputers. The toolset enables parallel programs to be built for single transputers or multi-transputer networks comprising one or more transputer types. Networks of processors are supported by a software through-routing configurer that greatly simplifies the placement of code and communication paths in complex applications. Improved embedded application support is provided by both configuration and symbol map utilities.

1.1 Applications

- Single processor embedded systems
- Multiprocessor embedded systems
- Porting of existing software and packages
- Massively parallel applications
- Evaluation of concurrent algorithms
- Low cost single chip applications
- Low level device control applications
- Scientific programming
- Education in concurrent programming

2 Product Overview

The INMOS occam 2 Toolsets provide complete occam cross-development systems for transputer targets. They can be used to build parallel programs for single transputers and for multi-transputer networks consisting of arbitrary mixtures of T2xx, T4xx and T8xx transputer types. Programs developed with the toolset are both source and binary compatible across all host development machines. The INMOS occam 2 Toolset is available for the following development platforms:

IMS D7305 occam 2 Toolset for IBM under MSDOS (Single user licence)

IMS D4305 occam 2 Toolset for Sun 4 under SunOS (Multi-user licence)

The occam programming language is a high-level language which supports the design and implementation of concurrent systems on networks of processors. The occam language is described in the following publication:

occam 2 Reference Manual, published by Prentice-Hall, ISBN 0-13-629312-3.

2.1 Mapping occam Programs Onto Transputer Networks

The occam programming model consists of parallel processes communicating through channels. Channels connect pairs of processes and allow data to be passed between them. Each process can be built from a number of parallel processes, so that an entire software system can be described as a process hierarchy. Each channel has a `PROTOCOL` which determines the types of messages that may flow across it. This model is consistent with many modern software design methods.

Figure 2.1 shows a collection of four processes communicating over channels. The **`multiplexor`** process communicates with a host computer and hands out work to be done to one of three **`worker`** processes. Results from the workers are then returned to the host by the multiplexor. The following examples show how this collection of processes can be described in occam and how the occam program can be mapped onto a network of processors.

The occam example in figure 2.2 illustrates how to program the collection of parallel processes in figure 2.1. It assumes that the **`multiplexor`** and **`worker`** processes have been compiled and the code has been placed in the transputer code files **`mux.tco`** and **`worker.tco`**.

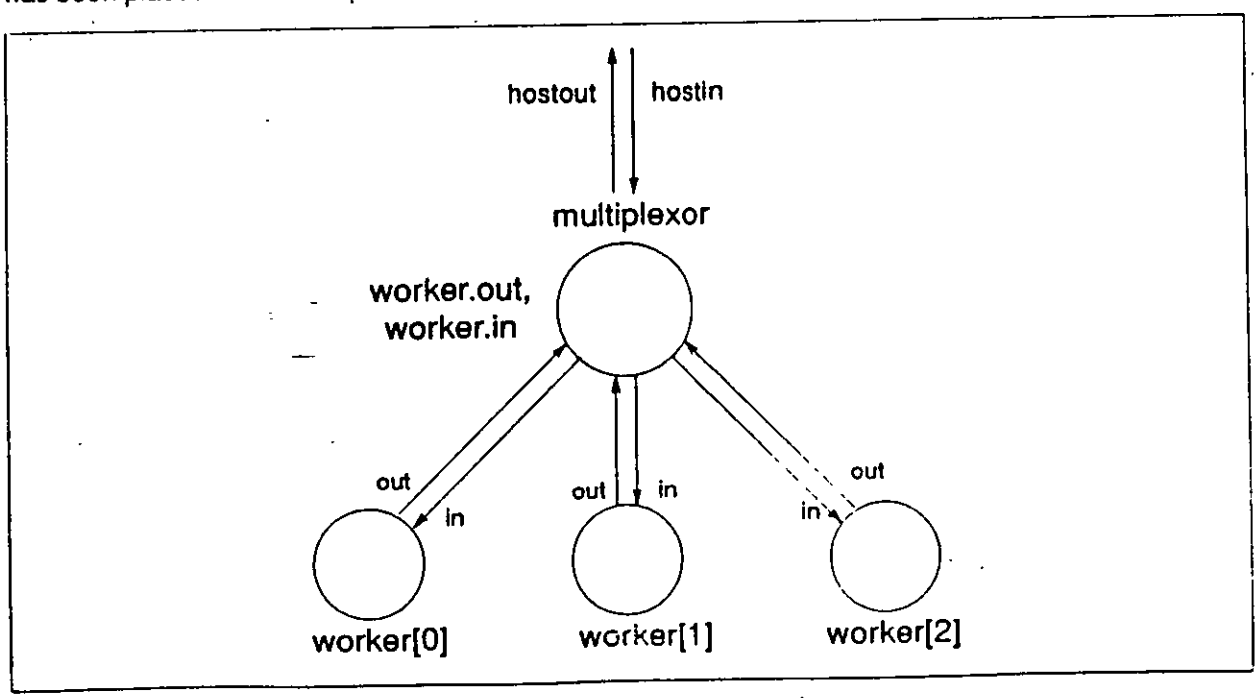


Figure 2.1 Software network

```
#INCLUDE "hostio.inc" -- contains SP protocol definition
#USE "mux.tco"
#USE "worker.tco"
PROC example (CHAN OF SP hostinput, hostoutput, []INT memory)
  [3]CHAN OF SP worker.in, worker.out:
  PAR
    multiplexor(hostinput, hostoutput, worker.out, worker.in)
    PAR i = 0 FOR 3
      worker(worker.in[i], worker.out[i])
:
```

Figure 2.2 Parallel processes in occam

The occam compiler in the toolset can be used to compile the program shown in Figure 2.2 and produce a code file to run on a single transputer. Alternatively it may be desirable to distribute the program over a network of processors. The program can be mapped onto a network using the configuration tools. A *configuration language* is used to describe the transputer network, and the mapping of the occam program onto the transputer network.

The following three examples illustrate how to configure the example program for a transputer network. In all cases we assume the `multiplexor` and `worker` processes in the software network have been compiled and linked into files `mux.lku` and `worker.lku` respectively. Three hardware networks are considered: the single processor network shown in figure 2.3 and the four-processor networks shown in figures 2.5 and 2.7.

Figure 2.4 shows the configuration text for mapping the software network in Figure 2.1 onto the hardware shown in Figure 2.3: a single T805 with 1 Mbyte of memory, connected to the host by link 0. In this example all the processes run on the root transputer. It might be useful to test this configuration before moving to a four-processor network.

Figure 2.6 shows the configuration text for mapping the software network on to the hardware shown in figure 2.5; four T805s, each with 1 Mbyte of memory. In this example the `multiplexor` process runs on the root transputer while the individual `worker` processes run on each of the other transputers.

In figures 2.5 and 2.6, processes on different processors communicate with each other over transputer links. Each channel pair is mapped onto a single link. If the network is as shown in figure 2.7, then the channels cannot be mapped onto the hardware so easily. In this case the configurer can automatically add communication routing processes onto the transputers so that the user processes can be mapped onto the processors as required. Figure 2.8 shows the configuration file for this network.

Each configuration example includes a `NETWORK` description section which describes how a particular network is connected. Since many applications can be run on identical networks it may be appropriate to include this definition from a separate file. The `NETWORK` description is followed by a `CONFIG` section which is virtually identical to the parallel process structure shown in Figure 2.2. The parallel process structure has been extended with `PROCESSOR` directives indicating which processor is to run each process. The description of the software network is almost identical in all cases; only the allocation of the `worker` processes is different. If a `MAPPING` section is used, it can be arranged that the software is identical in all cases.

For each multiprocessor system architecture, the connection of multiprocessor and `worker` processes is the same; only the hardware description is changed.

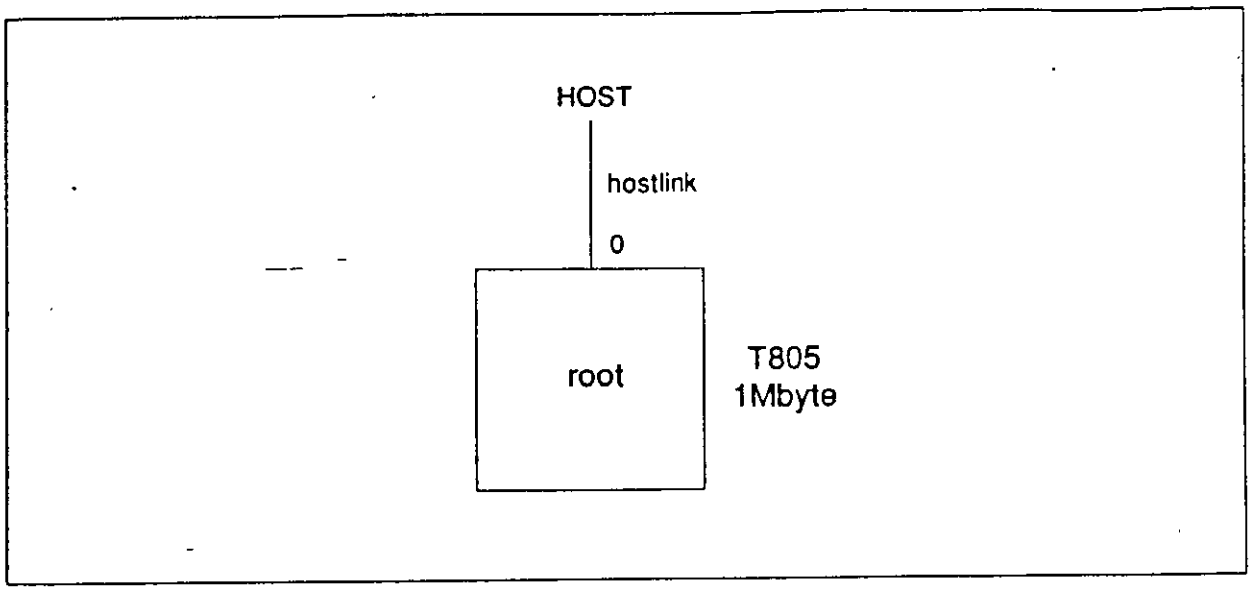


Figure 2.3 Hardware network 1

```
VAL M IS 1024*1024:
NODE root:
ARC hostlink:
NETWORK ONE.PROCESSOR
  DO
    SET root(type, memsize := "T805", 1*M)
    CONNECT root[link][0] TO HOST WITH hostlink
:
#INCLUDE "hostio.inc"
#USE "mux.lku"
#USE "worker.lku"
CONFIG
  [3]CHAN OF SP worker.in, worker.out:
  CHAN OF SP hostinput, hostoutput:
  PLACE hostinput, hostoutput ON hostlink:
  PAR
    PROCESSOR root
      multiplexor(hostinput, hostoutput, worker.out, worker.in)
    PAR i = 0 FOR 3
      PROCESSOR root
        worker(worker.in[i], worker.out[i])
:
```

Figure 2.4 Software configuration 1

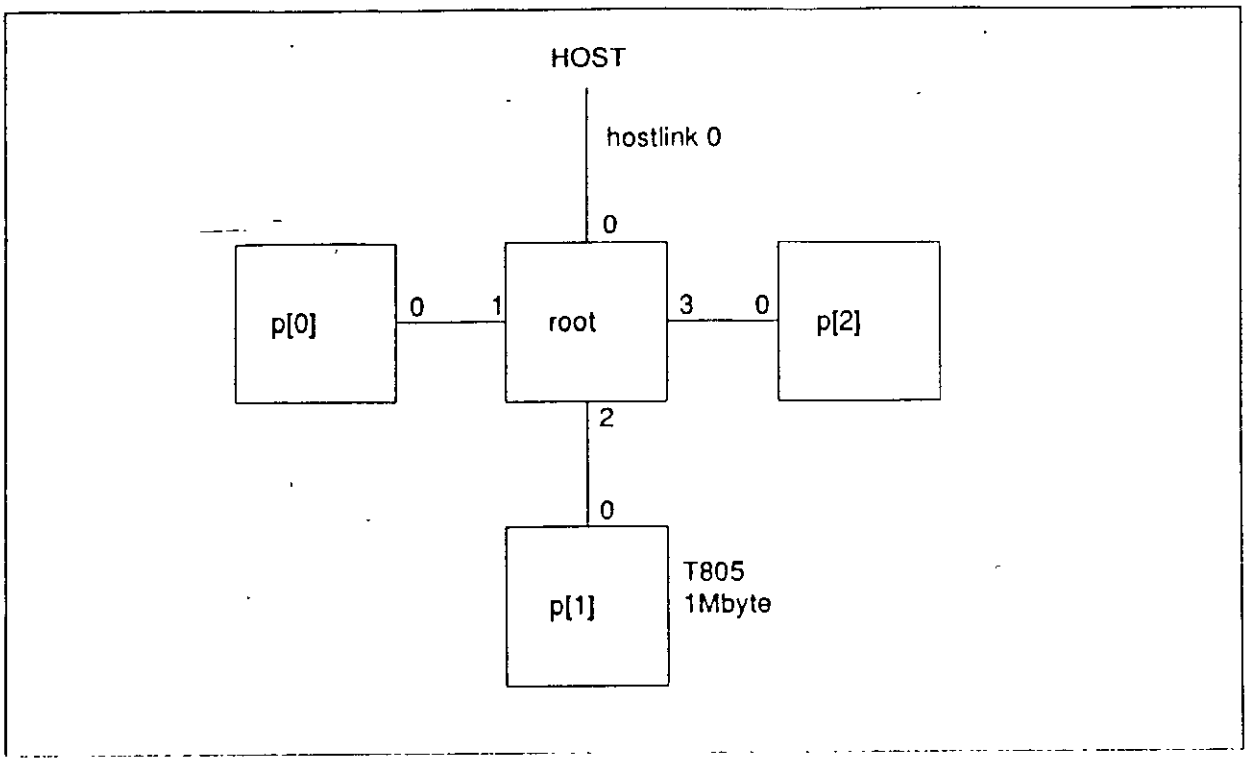


Figure 2.5 Hardware configuration 2

```
VAL M IS 1024*1024:
NODE root:
[3]NODE p:
ARC hostlink:
NETWORK FOUR.PROCESSOR
  DO
    SET root(type, memsize := "T805", 1*M)
    CONNECT root[link][0] TO HOST WITH hostlink
    DO i= 0 FOR 3
      DO
        SET p[i](type, memsize := "T805", 1*M)
        CONNECT root[link][i+1] TO p[i][link][0]
:
#INCLUDE "hostio.inc"
#USE "mux.lku"
#USE "worker.lku"
CONFIG
  CHAN OF SP hostinput, hostoutput:
  PLACE hostinput, hostoutput ON hostlink:
  [3]CHAN OF SP worker.in, worker.out:
  PAR
    PROCESSOR root
      multiplexor(hostinput, hostoutput, worker.out, worker.in)
    PAR i = 0 FOR 3
      PROCESSOR p[i]
        worker(worker.in[i], worker.out[i])
:
```

Figure 2.6 Software configuration 2

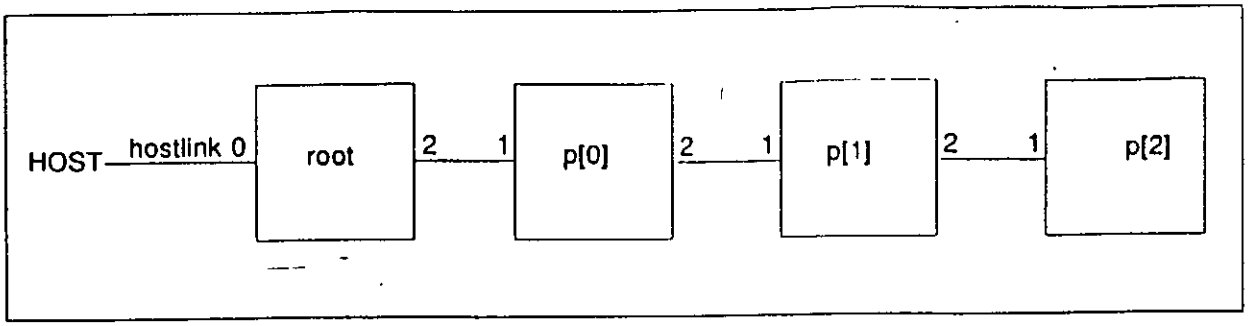


Figure 2.7 Hardware configuration 3

```
VAL M IS 1024*1024:
NODE root:
[3]NODE p:
ARC hostlink:
NETWORK FOUR.PROCESSOR
  DO
    SET root(type, memsize := "T805", 1*M)
    CONNECT root[link][0] TO HOST WITH hostlink
    DO i= 0 FOR 3
      DO
        SET p[i](type, memsize := "T805", 1*M)
         IF
           (i = 0)
            CONNECT p[i][link][1] TO root[link][2]
           TRUE
            CONNECT p[i][link][1] TO p[i-1][link][2]
:
#INCLUDE "hostio.inc"
#USE "mux.lku"
#USE "worker.lku"
CONFIG
  CHAN OF SP hostinput, hostoutput:
  PLACE hostinput, hostoutput ON hostlink:
  [3]CHAN OF SP worker.in, worker.out:
  PAR
    PROCESSOR root
      multiplexor(hostinput, hostoutput, worker.out, worker.in)
    PAR i = 0 FOR 3
      PROCESSOR p[i]
        worker(worker.in[i], worker.out[i])
:
```

Figure 2.8 Software Configuration 3

2.2 occam 2 development system

The occam 2 compiler supports the full occam 2 language as defined in the occam 2 reference manual. In addition to type checking the compiler will validate programs to ensure that variables and communication channels are being used correctly in parallel components of a program. This detects many simple programming errors at compile time. The language provides support for low-level programming, including the allocation of variables to specific memory addresses, and the inclusion of transputer assembly code.

The compiler will generate code for the full range of transputer types; IMS M212, T212, T222, T225, T400, T414, T425, T426, T800, T801, and T805. Since the different processor types share a common core of instructions it is possible to create compiled code which will run on a range of processors.

A program may be compiled in one of several 'error modes' which determine the behavior of the program when a run-time error occurs. A mode which supports the use of the post-mortem debugger may be chosen; in this mode the network will come to a halt when a run-time error occurs. A compiler option allows all error checking to be removed from time-critical or proven parts of a program, and unchecked routines can be called from within a checked system.

The processor target, error mode and other compilation options are specified by command line switches.

Collections of procedures and functions can be compiled separately with the occam 2 compiler. Separately compiled units may be collected together into libraries. The linker is used to combine separately compiled units into a program to run as a self contained unit (process). The linker supports selective loading of library units, and the linking of components written in other programming languages including INMOS ANSI C.

The toolset supports the use of **make**, a program building tool available under UNIX and other operating systems. The input to **make** is a 'Makefile' which describes how a program is to be built from its parts, and **make** rebuilds those parts of the program which have changed. In the occam toolset a tool called **imakef** is provided which will generate a Makefile automatically from the occam program text. This guarantees that the Makefile is consistent with the program sources.

To create a multi-transputer program, the mapping of the processes to processors is described in the configuration description. This description is processed by a pair of tools called the *configurer* and the *collector*. The configurer checks the description, and passes information to the collector on how the program code and data should be mapped onto the network. The collector creates the bootstrap and routing information necessary to load the entire network, and stores this, along with the compiled code, in a program code file. The program code file can be down-loaded to one transputer in a network, and the program will automatically boot all the processors in the network and distribute the application code to them.

A server program on the host, called the **iserver**, can be used to load programs on to transputer networks. Once loaded, the programs start automatically. The server supports access to the host terminal and file system from the transputer network.

A compiled and configured program may be run on a network under the control of the interactive debugger supplied with the occam 2 toolset. One processor in the network is used to run this debugger. Breakpoints may be set in the program on any processor in the network. Processes stopped at breakpoints may be examined and continued.

INMOS has also produced an advanced development environment called INQUEST, providing windowing based tools for debugging and profiling applications. The INQUEST debugger supports advanced operation including breakpoints, watchpoints, process halting and continuation, multiple windows and command language operation. The INQUEST debugger runs on the host and therefore does not need an extra transputer in the network.

As an alternative to loading the program code from a host, the program code may be placed into an EPROM. The program code for a whole network of processors may be booted and loaded from a single processor with a ROM. The program code file may be converted to an industry standard format for programming EPROMs, using the EPROM tools in the toolset.

If the transputer network halts because of a run-time error, or if the user interrupts the server, the symbolic network debugger may be used, in post-mortem mode, to investigate the state of the network in terms of the program source text.

A transputer network may consist of any combination of processor types; a network containing a number of different types is called a *mixed network*. The configuration tools, EPROM support tools and debugging tools all support mixed networks.

2.2.1 Libraries

The occam toolsets provide a wide range of occam libraries, including mathematical functions, string operations and input/output functions. The libraries support similar functions across the full range of transputer types, making it easy to port software between transputer types. Sources of most of the libraries are provided, for adaptation if required.

The libraries provided are listed below.

occam compiler library

This is the basic occam run-time library. It includes: multiple length arithmetic functions; floating point functions; IEEE arithmetic functions; 2D block moves; bit manipulation; cyclic redundancy checks;code execution; arithmetic instructions. The compiler will automatically reference these functions if they are required.

`snglmath.lib, dblmath.lib`

Single and double length mathematics functions (including trigonometric functions). These libraries use floating point arithmetic and will produce identical results on all processors. The occam source code is also provided.

`tbmaths.lib`

Mathematical functions optimized for the T414, T425 and T400 processors. These functions provide slightly different results to the maths library above but within the accuracy limits of the function specifications. The occam source code is also provided.

`string.lib`

String manipulation procedures. The occam source code is also provided.

`hostio.lib`

Procedures to support access to the host terminal and file system through the iserver. The occam source code is also provided.

`streamio.lib`

Procedures to read and write using input and output streams. The occam source code is also provided.

`msdos.lib`

Procedures to access certain DOS specific functions through the file server. The functions are specific to the IBM PC. The occam source code is also provided.

`crc.lib`

Procedures to calculate the cyclic redundancy check (CRC) values of strings.

`convert.lib`

Procedures to convert between strings and numeric values.

`xlink.lib`

Procedures implementing link communications which can recover after a communication failure.

`debug.lib`

Procedures supporting the insertion of debugging messages and assertions within a program, for use with the interactive debugger.

2.2.2 Mixed language programs

It is often appropriate in the development of a large system to use a mixture of programming languages. For example, it may be necessary to preserve an investment in existing C code, while using occam to express the concurrency and communication within the system.

The occam programming model provides an excellent vehicle for building mixed language systems where the components built in each language can be clearly defined with a simple interface. The occam toolset can be used to bind these components together and distribute them over a network of transputers.

The occam toolset supports calling C functions directly from occam. It is possible to call C functions which require static data or heap; occam procedures are provided to set up occam arrays for use as static or heap area for collections of C functions.

The associated INMOS ANSI C toolset allows occam procedures and (single valued) occam functions to be called from C just like other C functions. In addition occam programs can be mixed with C programs at the configuration level, providing a message based interface between C and occam code.

2.2.3 Support for embedded applications

The toolset has been designed to support the development of embedded applications. The features include the ability to place code and data at particular places in memory, and being able to locate where functions and data areas reside in memory.

2.2.4 Placing code and data

The highest level occam processes in a system are visible to the configurer and identified in the configuration language; these are known as configuration processes. Each processor contains one or more configuration processes.

A configuration process consists of its code, workspace and vectorspace. The configurer allocates each of these separate segments a place in the memory of the processor. By default, the configurer assumes that the memory space is contiguous from the lowest memory address for the number of bytes defined by the `memory` attribute of the processor.

The default memory space can be changed by reserving a section of memory adjacent to the lowest memory address for a processor. The configurer will not then allocate any segment of the application to that reserved area of memory. Thus the on-chip memory of a transputer can be reserved for the application's use.

In addition, particular segments of an application can be allocated to particular places in memory, provided only that they are placed outside the default memory space used by the configurer. For example, the workspace of a process can be placed in fast on-chip memory and the code of another, less important, process can be put into slow memory at a high address. These allocations are under the control of the user by means of extra attributes in the processor definitions in the configuration file.

The compiler, linker and collector each will optionally produce a listing of how the various parts of an application are mapped into the segments of memory. A tool is provided that can read all these map files and produce a summary of the whole application, giving the locations of all the functions, workspace areas and vectorspace areas.

2.2.5 Bootstraps

The source code of the standard bootstraps are provided. The user can then write bootstraps that are tailored to a specific application by using the standard ones as templates.

If the application is spread over several processors in a network, then it is possible to modify that part of the bootstrapping process which loads remote processors in order to perform special initializations on those processors before they receive any application code.

Note: If the user wishes to modify the initial or primary bootstraps, the ANSI-C toolset (D4314 or D7314) is required for its C-compiler/assembler.

2.2.6 Debugging

The occam toolset provides three debugging tools: an interactive symbolic debugger, a post-mortem symbolic debugger, and an interactive T425 simulator.

Interactive symbolic debugging

The interactive debugger provides source level interactive debugging of a program running on a network of processors. The debugger supports breakpoints at the occam or C source code level. Breakpoints may be set on any processor in the network. The state of any halted process in the network can be examined symbolically. Variables – both scalar and arrays – may be read or written symbolically. The call stack can be examined to determine the nesting of procedure calls and parallel process instantiation. Channels can be inspected. If another process is waiting on a channel it is possible to inspect the state of the waiting process, even if it is on another processor. The debugger will automatically switch between occam and C when debugging a mixed language program.

The interactive debugger also provides low-level examination of memory on any processor in the network.

Post–mortem symbolic debugging

The post-mortem debugger can be used to examine the state of a transputer network symbolically. The debugger works with exactly the same code as will run in the final product; there is no additional code inserted to support debugging. This allows debugging in those circumstances where the program works under simulation or interactive debugging, but fails when run normally. After a program has halted or been interrupted by the developer, the state of the network can be preserved so that the post-mortem debugger can be run. The post-mortem debugger will support direct analysis of the network, or allow the state of the network to be saved in a dump file for later analysis. The post-mortem debugger supports the same symbolic and machine level browsing functions as the interactive debugger.

Both the interactive debugger and the post-mortem debugger run on a transputer. When doing interactive debugging, a processor must be allocated to the debugger, in addition to the target network. When doing post-mortem debugging, the state of one of the processors must be saved before running the post-mortem debugger on it.

T425 simulation

The T425 simulator is a simulation of the IMS T425 processor connected to a host running the **Iserver**. It allows transputer programs to be executed on the host machine and supports machine level debugging of transputer code.

The machine level debugging support includes: breakpoints and single stepping at machine code level, browsing memory in different forms including disassembled machine code, access to registers and processor queues.

The transputer simulator can be used to run transputer programs on the PC or Sun 4. Code for a transputer network can usually be re-configured for a single processor (as shown in section 2), allowing the simulator to be used to try out multi-processor programs as well as single processor programs.

A batch mode is provided for running test suites.

2.2.7 Optimized code generation

The occam compiler implements a wide range of code optimization techniques:

Constant folding. The compiler evaluates all constant expressions at compile time.

Workspace allocation. Frequently used variables are placed at small offsets in workspace, thus reducing the size of the instructions needed to access them, and hence increasing the speed of execution.

Dead-code elimination. Code that cannot be reached during the execution of the program is removed.

Peephole optimization. Code sequences are selected that are the fastest for the operation.

Instruction scheduling. Where possible the compiler exploits the internal concurrency of the transputer. In particular integer and floating point operations can be overlapped to exploit the parallel execution of the integer processor and floating point processor on the T805 series.

Constant caching. Some constants have their load time reduced by placing them in a constant table.

CASE statements. The compiler can generate a number of different code sequences to cover the dense ranges within the total range.

Inline code Procedures and functions can optionally be implemented as inline code.

The compiler, linker and configurer provide features which allow programmers to make good use of the transputer's on-chip RAM.

2.2.8 Assembler Inserts

The occam toolset provides an assembler insert facility. The assembler insert facility supports

- Access to full instruction set of the transputer
- Symbolic access to occam variables
- Pseudo-operations to load multiple values into registers
- Ability to load results of occam expressions to registers
- Labels and jumps
- Directives to access particular workspace values

2.2.9 Improvements over previous releases

The D7305 and D4305 occam toolsets represent a considerable improvement over the D7205 and D4205 toolset releases. A summary is provided below.

- Compatible with the new INMOS ANSI C toolset release (IMS D7314, D4314): identical object format and many tools in common.
- Improved mixing of C and occam.
- Faster execution.
- Automatic support of software through-routing and multiplexing of channels across a network.
- Listing of memory use for every processor of a network.
- Improvements in runtime libraries.
- Placement of code, workspace and vectorspace at user-specified locations.
- Reservation of on-chip RAM for user's application.
- Compatibility with the INMOS INQUEST advanced development environment.

3 occam Toolset Product Components

3.1 Documentation

- Delivery manual
- Release notes
- User guide
- Toolset reference manual
- Language and libraries reference manual
- Performance note
- Toolset handbook
- occam 2 language reference manual
- Tutorial introduction to occam

3.2 Software Tools

- **oc, ilink, ilibr**: occam compiler, linker and librarian
- **imakef, ilist**: Makefile generator and binary lister program
- **imap**: memory map tool
- **occonf, icollect**: configuration tools
- **isim, idebug, iskip, idump**: debugging tools
- **iserver**: INMOS host server program
- **ieprom, iemit**: EPROM and memory interface programming tools

3.3 Software libraries

- occam compiler libraries
- **snglmath, dblmath**: mathematics functions (includes sin, cos, etc.)
- **tbmaths**: mathematics functions optimized to run on T414, T425 and T400
- **string**: string manipulation procedures
- **hostio**: file and terminal I/o procedures
- **streamio**: file and terminal stream i/o procedures
- **msdos**: access to certain MS-DOS calls
- **crc**: CRC calculation procedures
- **convert**: string-number conversion procedures
- **xlink**: extraordinary link handling procedures

- **debug**: debugging procedures.

3.4 Source code

- occam example programs
- Source of makefile generator
- Source of server program
- occam library sources

4 Product Variants

4.1 IMS D7305 IBM PC version

All tools are provided in a form which will run on the host machine. The exceptions to this rule are the target debugging tools which will only run on the transputer.

4.1.1 Operating requirements

The following configuration is required:

- IBM PC with 386 or 486 processor and 4 Mbytes memory.
- DOS 5.0 or later
- 10 Mbyte of free disk space

The interactive symbolic debugger requires an additional transputer with at least 2 Mbyte external memory. The simulator does not require this additional transputer.

Access to a transputer network can be supplied by an IMS B008 Motherboard and transputer modules or by using the IMS B300 Ethernet/transputer gateway in conjunction with the IMS S707 software package.

The other target debugging tools **`iskip`** and **`idump`** also require access to the target transputer network.

4.1.2 Distribution media

Software is distributed on two media systems (both of which are supplied in the product):

- 1.2 Mbytes (96TPI) 5.25 inch IBM format floppy disks
- 1.44 Mbytes 3.5 inch IBM format floppy disks.

4.2 IMS D4305 Sun 4 version

All tools are provided in a form which will run on the host machine. The exceptions to this rule are the target debugging tools which will only run on a transputer.

The IMS D4305 software is run in conjunction with a 'floating' licence manager. For each product purchased, up to four users are able to use the toolset concurrently at any one customer site. The tools can be run on any Sun 4 machine that is part of a network connected to a single machine where the licence manager is installed. Further information about the licence manager is included in the product Delivery Manual.

4.2.1 Operating requirements

For hosted cross-development and debugging the following configuration is required:

- A Sun 4 workstation or server
- SunOS 4.1.1 or later
- 10 Mbytes of free disk space.

For loading target systems and remote debugging, one of the following is required:

- IMS B014 or IMS B016 VME/transputer interface board used with the IMS S514 driver package.

- Ethernet connection to an IMS B300 and the IMS S507 software package.

The interactive symbolic debugger requires an additional transputer with at least 2 Mbyte external memory. The simulator does not require this additional transputer.

4.2.2 Distribution media

Sun 4 software is distributed on DC600A data cartridges 60 Mbyte, QIC-24, tar format.

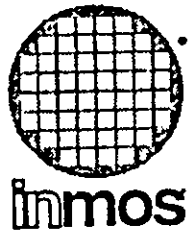
5 Support

INMOS transputer, board-level and toolset development products are sold and supported worldwide through SGS-THOMSON Sales Offices, Regional Technology Centers, and authorized distributors.

A registration form is provided with each product. Return of this registration form will ensure you are informed about future product updates and covered by INMOS software warranty. Software problem report forms are included with the software.

6 Ordering Information

Description	Order number
Professional occam 2 Toolset for 386 PC single user licence.	IMS D7305
Professional occam 2 Toolset for Sun 4 four-user licence.	IMS D4305



occam 2 toolset handbook

INMOS Limited

72 TDS 199 01

March 1991

INMOS document number: 72 TDS 199 01

Contents

The occam 2 Toolset

Tool	Description
`icollect`	The code collector which creates bootable files from linked units or configuration binary files.
`icvemit`	Memory interface file convertor. Converts files produced by `iemi` (a previous version of `iemit`) to a format suitable for use with `ieprom` and `iemit`.
`icvlink`	The TCOFF file convertor which converts LFF object files to TCOFF format.
`idebug`	The network debugger which provides post-mortem and interactive debugging of transputer programs.
`idump`	The memory dumper for saving the program image on the root transputer.
`iemit`	The transputer memory configurer for evaluating and defining memory configurations.
`ieprom`	The EPROM program formatter tool which generates transputer bootable code for input to ROM programmers.
`ilibr`	The toolset librarian which builds libraries of compiled code.
`ilink`	The toolset linker which links separately compiled code into a single file.
`ilist`	The binary lister which displays information from toolset object files.
`imakef`	The Makefile generator which generates Makefiles for input to MAKE programs.
`iserver`	The host file server which loads programs onto transputer hardware and provides host communication.
`isim`	The T425 simulator. Simulates program execution on an IMS T425 transputer.
`iskip`	The skip loader tool which loads over the root transputer.
`oc`	The occam 2 compiler. Generates object code for specific transputer targets.
`occonf`	The configurer which creates configuration binary files from configuration descriptions.

Default file extensions

Extension	Description
.btl	Bootable code file. Created by **icollect**.
.btr	Executable code minus bootstrap information used for input to **ieprom**. Created by **icollect**.
.cfb	Configuration binary file. Created by **occonf** or **icollect**.
.clu	Configuration object file. Created by **occonf**.
.dmp	Core-dump file created by **idump** or network-dump file created by **idebug** or **isim**.
.inc	Include file. Input to **oc** and **occonf**.
.lku	Linked unit. Created by **ilink**.
.lbb	Library build file. Command file for **ilibr**.
.lib	Library file. Created by **ilibr**.
.liu	Library usage file. Created by **imakef**.
.lnk	Linker indirect file. Command file for **ilink**.
.occ	occam 2 source files.
.pgm	Configuration description source file.
.rsc	Dynamically loadable code file. Created by **icollect**.
.tco	Compiled code file. Created by **oc**.

Tools

icollect – code collector

Generates bootable code files for single and multiple transputer programs from linked units and configuration binary files respectively. Also used to create non-bootable single transputer programs for dynamic loading or booting from ROM.

Syntax: `icollect` *filename* {*options*}

where: *filename* is a linked unit (`.lku`) or configuration binary file (`.cfb`).

Note: If the input file is a linked unit (single transputer mode) then the '`T`' option must be specified.

Options:

`B` *filename*	Specifies external bootstrap loader program.
`C` *filename*	Specifies debug data file.
`D`	Disables the debug data file for single transputer programs.
`E`	Changes setting of Halt On Error flag.
`I`	Displays progress information.
`K`	Creates non-bootstrapped code.
`L`	Loads the tool and terminates.
`M` *memorysize*	Specifies memory size on root processor.
`O` *filename*	Specifies output file.
`P` *filename*	Specifies a memory map output file.
`RA`	Creates RAM-loadable code.
`RO`	Creates ROM-loadable code.
`RS` *romsize*	Specifies size of ROM on the root processor.
`S` *stacksize*	Specifies extra runtime stack size for single transputer programs.

Options:

`T`	Creates bootable code for a single transputer.
`XM`	Loads transputer-hosted tool for multiple execution.
`XO`	Loads transputer-hosted tool for single execution.
`Y`	Disables interactive debugging with `idebug` for single transputer programs.

icvemit – memory interface convertor

Converts memory configuration files produced by **iemi** (a previous version of **iemit**) to a file format suitable for use with **iemit** and **ieprom**.

Syntax: **icvemit** *filename* {*options*}

where: *filename* is the input file to be converted.

Options:

I	Displays progress information.
O *filename*	Specifies output file.

icvlink – object code convertor

Converts object code into TCOFF format.

Syntax: **icvlink** *filename* {*options*}

where: *filename* is the object file to be converted.

Options:

D	Creates multiple modules from TA and TC modules. For libraries only.
I	Displays progress information.
L	Loads the tool and terminates.
O *filename*	Specifies output file.
P	Creates T8 modules from TA and TC modules.
XM	Loads transputer-hosted tool for multiple execution.
XO	Loads transputer-hosted tool for single execution.

idebug – network debugger

Provides post-mortem debugging and breakpoint debugging.

Syntax: `idebug` *filename* {*options*}

where: *filename* is a bootable file.

Options:

B *linknumber*	Breakpoint debug a network.
M *linknumber*	Postmortem debug a previous breakpoint session.
T *linknumber*	Postmortem debug a program not using the root processor.
R *filename*	Postmortem debug a program using the root transputer.
N *filename*	Postmortem debug from a network dump file.
C *type*	Specify processor type instead of class for non-configured programs.
D	Dummy debugging session.
A	Assert subsystem analyse on the network.
S	Ignore subsystem error status when breakpoint debugging.
I	Display debugger version number.

idump – memory dumper

Writes the root transputer's memory to a file. Used in debugging programs that use the root transputer.

Syntax: `idump` *filename memsize* {*offset length*}

where: *filename* is the bootable program file

memsize is the amount of memory measured in bytes to be dumped to the file

offset is a byte offset from the start of memory

length is the number of bytes of memory starting at *offset* to be written to the file in addition to *memsize*.

idebug – network debugger

Provides post-mortem debugging and breakpoint debugging.

Syntax: `idebug` *filename* {*options*}

where: *filename* is a bootable file.

Options:

B *linknumber*	Breakpoint debug a network.
M *linknumber*	Postmortem debug a previous breakpoint session.
T *linknumber*	Postmortem debug a program not using the root processor.
R *filename*	Postmortem debug a program using the root transputer.
N *filename*	Postmortem debug from a network dump file.
C *type*	Specify processor type instead of class for non-configured programs.
D	Dummy debugging session.
A	Assert subsystem analyse on the network.
S	Ignore subsystem error status when breakpoint debugging.
I	Display debugger version number.

idump – memory dumper

Writes the root transputer's memory to a file. Used in debugging programs that use the root transputer.

Syntax: `idump` *filename memsize* {*offset length*}

where: *filename* is the bootable program file

memsize is the amount of memory measured in bytes to be dumped to the file

offset is a byte offset from the start of memory

length is the number of bytes of memory starting at *offset* to be written to the file in addition to *memsize*.

ilibr – librarian

Builds libraries of code from separate files.

Syntax: `ilibr` {*filenames*} {*options*}

where: *filenames* is a list of compiled modules.

Options:

F *filename*	Specifies library indirect file.
I	Displays progress information.
L	Loads the tool and terminates.
O *filename*	Specifies output file.
XM	Loads transputer-hosted tool for multiple execution.
XO	Loads transputer-hosted tool for single execution.

ilink – linker

Links object files together, resolving external references, to create a single linked unit.

Syntax: `ilink` {*filenames*} {*options*}

where: *filenames* is a list of compiled modules or libraries.

Options:

TA	Link for class TA (T400/T414/T425/T800/T801/T805).
TB	Link for class TB (T400/T414/T425).
T212	Link for T212.
T2	Link for T212/T222/M212.
T222	Link for T222.
T225	Link for T225.
T3	Link for T225.
T400	Link for T400.
T414	Link for T414. Default.
T4	Link for T414.
T425	Link for T425.
T5	Link for T425/T400.
T800	Link for T800.
T8	Link for T800.
T801	Link for T801.
T805	Link for T805.
T9	Link for T801/T805.
H	Generates HALT mode output. Default.
S	Generates STOP mode output.
X	Generates UNIVERSAL mode output.

Options:

T	Specifies TCOFF format. Default.
LB	Specifies LFF format. For use with the iboot and iconf tools, supported by the D705, D605 and D505 toolsets.
LC	Specifies LFF format. As above, but only for use with the iconf tool.
EX	Allows extraction of modules without linking them.
F *filename*	Specifies linker indirect file.
I	Displays progress information.
KB *memorysize*	Specifies maximum internal code buffer size.
L	Loads the tool and terminates.
ME *entryname*	Specifies main entry point.
MO *filename*	Generates module information file.
O *filename*	Specifies output file.
U	Allows unresolved references.
XM	Loads transputer-hosted tool for multiple execution.
XO	Loads transputer-hosted tool for single execution.
Y	Disables interactive debugging for occam code.

ilist – binary lister

Decodes and displays information from object files and bootable files.

Syntax: ilist *filename* {*options*}

where: *filename* is an object or bootable file.

Options:

A	Displays all available symbolic information.
C	Displays code in hexadecimal.
E	Displays all exported names.
H	Displays specified file(s) in hexadecimal format.
I	Displays full progress information.
L	Loads the tool and terminates.
M	Displays module data.
N	Displays library index data.
O *filename*	Specifies output file.
P	Displays procedural interfaces.
R *reference*	Displays library module(s) with the specified reference.
T	Displays full listing.
W	Identifies file type. Default.
X	Displays all external references.
XM	Loads transputer-hosted tool for multiple execution.
XO	Loads transputer-hosted tool for single execution.

imakef – Makefile generator

Creates Makefiles for toolset compilations.

Syntax: `imakef` {*filenames*} {*options*}

where: *filenames* is a list of target object or bootable files.

Options:

C	For C modules only. Specifies input from a linker indirect file.
D	Disables debugging data.
I	Displays full progress information.
L	Loads the tool and terminates.
NI	Do not include dependencies on ISEARCH.
O *filename*	Specifies output file.
R	Incorporates a delete rule.
XM	Loads transputer-hosted tool for multiple execution.
XO	Loads transputer-hosted tool for single execution.
Y	Disables interactive (breakpoint) debugging in the target file.

iserver – server/loader

Loads programs onto transputers and transputer boards and serves host communications.

Syntax: `iserver` *bootablefile* {*options*}

where: *bootablefile* is a bootable file.

Options:

SA	Analyses root transputer and peeks 8K of memory.
SB *filename*	Loads specified program.
SC *filename*	Copies specified file to root transputer link
SE	Monitors transputer error flag.
SI	Displays progress information.
SL *name*	Specifies device name or link address.
SP *n*	Specifies KBytes of memory peeked on Analyse.
SR	Resets root transputer and subsystem on the link.
SS	Serves the link, that is, starts up the host communications.
Option SB is equivalent to invoking the combination: SR SS SI SC *filename*.	

isim – T425 simulator

Simulates the execution of a program on the IMS T425.

Syntax: `isim` *filename programparameters* {*options*}

where: *filename* is a bootable file.

Options:

B	Batch mode operation.
BQ	Batch Quiet mode. No progress information.
BV	Batch Verify mode.
I	Displays full progress information.
L	Loads the tool and terminates.
N	No more options for the simulator.
XM	Loads transputer-hosted tool for multiple execution.
XO	Loads transputer-hosted tool for single execution.

iskip – skip loader

Allows programs to be loaded onto transputer networks beyond the root transputer.

Syntax: `iskip` *linknumber* {*options*}

where: *linknumber* is the root transputer link to which the target network is connected.

Options:

E	Monitors subsystem error status; terminates when it becomes set.
R	Reset subsystem but not the root transputer.
I	Displays detailed progress information.

oc – occam 2 compiler

Compiles occam 2 source code.

Syntax: oc *filename* {*options*}

where: *filename* is an occam source file.

Options:

TA	Compile for class TA (T400/T414/T425/T800/T801/T805).
TB	Compile for class TB (T400/T414/T425).
T212	Compile for T212.
T2	Compile for T212/T222/M212.
T222	Compile for T222.
T225	Compile for T225.
T3	Compile for T225.
T400	Compile for T400.
T414	Compile for T414. Default.
T4	Compile for T414.
T425	Compile for T425.
T5	Compile for T425/T400.
T800	Compile for T800.
T8	Compile for T800.
T801	Compile for T801.
T805	Compile for T805.
T9	Compile for T801/T805.
H	Generates HALT mode output. Default.
S	Generates STOP mode output.
X	Generates UNIVERSAL mode output.
G	Enables use of ASM or GUY code for a restricted set of transputer instructions.
W	Enables use of ASM or GUY code for the full set of transputer instructions.

Options:

K	Disables insertion of run-time range checks.
U	Disables insertion of run-time error checks.
NA	Disables insertion of run-time checks for ASSERTs.
L	Loads the tool and terminates.
XM	Loads transputer-hosted tool for multiple execution.
XO	Loads transputer-hosted tool for single execution.
A	Disables alias and usage checking.
B	Displays messages in brief.
C	Only performs check.
D	Generates minimal debugging information.
E	Disables use of the compiler libraries
I	Displays progress information.
N	Disables usage checking
O *outputfile*	Specifies output file.
R *filename*	Redirects error messages to a file.
V	Disables the generation of code with a separate vector space requirement.
Y	Disables interactive debugging with idebug.
NWP	Disables warning messages of unused parameters.
NWU	Disables warning messages of unused variables or routines.
WD	Provides a warning when a name is descoped.
WO	Provides a warning when a run-time alias check is generated.

occonf – configurer

Generates configuration binary files from configuration descriptions.

Syntax: occonf *filename* {*options*}

where: *filename* is a configuration description (.pgm) source file.

Options:

B	Displays messages in brief.
C	Only performs check.
I	Displays progress information.
L	Loads the tool and terminates.
O *outputfile*	Specifies output file.
R *filename*	Redirects error messages to a file.
V	Disables the generation of code with a separate vector space requirement.
Y	Disables interactive debugging with idebug.
RA	Creates RAM-loadable code.
RO	Creates ROM-loadable code.
H	Generates HALT mode output. Default.
S	Generates STOP mode output.
X	Generates UNIVERSAL mode output.
K	Disables insertion of run-time range checks.
U	Disables insertion of run-time error checks.
NA	Disables insertion of run-time checks for ASSERTs.

Options:

G	Enables use of ASM or GUY code for a restricted set of transputer instructions.
W	Enables use of ASM or GUY code for the full set of transputer instructions.
RE	Enables memory lay-out re-ordering.
NWP	Disables warning messages of unused parameters.
NWU	Disables warning messages of unused variables or routines.
WD	Provides a warning when a name is descoped.
WO	Provides a warning when a run-time alias is check is generated.
XM	Loads transputer-hosted tool for multiple execution.
XO	Loads transputer-hosted tool for single execution.

Debugger commands

Debugger symbolic functions

Function	Description
INSPECT	Inspect symbol.
CHANNEL	Locate to process waiting on channel.
TOP	Locate to last error or location.
RETRACE	Retrace last operation.
RELOCATE	Locate back to last location line.
INFO	Display extra information.
SEARCH	Search for string.
MONITOR	Change to Monitor page.
BACKTRACE	Locate to procedure or function call.
HELP	Display function keys.
GET ADDRESS	Display address of source line.
CHANGE FILE	Display different file.
ENTER FILE	Display included file.
EXIT FILE	Display enclosing file.
GOTO LINE	Go to specific line in file.
TOP OF FILE	Go to first line in file.
BOTTOM OF FILE	Go to last line in file.
TOGGLE HEX	Display C variables in Hex.
TOGGLE BREAK ‡	Set/clear breakpoint.
MODIFY ‡	Change variable in memory.
RESUME ‡	Resume program from breakpoint.
CONTINUE FROM ‡	Resume program from current line.
INTERRUPT ‡	Return to Monitor page.
MODIFY ‡	Change variable in memory.
FINISH	Quit.
‡ Breakpoint mode only	

Debugger Monitor page commands

Key	Meaning	Description
A	ASCII	View memory in ASCII.
B‡	Breakpoints	Enter breakpoint debugging.
C	Compare	Compare actual code with expected code.
D	Disassemble	Display transputer instructions.
E	Next Error	Go to next processor with error flag set.
F	Select file	Select source file for display.
G	Goto process	Enter source level debugging for a process.
H	Hex	View memory in hexadecimal.
I	Inspect	View memory in any type.
J‡	Jump	Start or resume program.
K	Processor names	Display processor names.
L	Links	Display processes waiting on links or Event pin.
M	Memory map	Display transputer's memory map
N	Network dump	Dump network memory.
O	Specify process	Resume symbolic debugging.
P	Processor	Change processor.
Q	Quit	Quit debugger.
R	Run queues	Display active process queues.
T	Timer queues	Display timer queues.
U‡	Update	Update Monitor page with new register values.
V	Process names	Display process names.
W‡	Write	Write to memory.
X	Exit	Change to symbolic mode.
Y‡	Postmortem	Change to post-mortem debugging.
?	Help	Display help information.
‡ Breakpoint mode only		

Debugger Monitor page commands (contd)

Function	Description
RETRACE	Redo last operation.
RELOCATE	Locate to last location line.
LINE UP	Next line.
LINE DOWN	Previous line.
PAGE UP	Previous page.
PAGE DOWN	Next page.
HELP	Display help information.
REFRESH	Re-draw the screen.
TOP	Find last instruction executed.
↑	Scroll display.
↓	Scroll display.
←	Scroll processor left.
→	Scroll processor right.

Simulator commands

Key	Meaning	Description
A	ASCII	View memory in ASCII.
B	Breakpoints	Breakpoint menu.
D	Disassemble	Display transputer instructions.
G	Go	Run/resume program.
H	Hex	View memory in hexadecimal.
I	Inspect	Display memory in occam type.
J	Jump	Resume (Jump into) program.
L	Links	Display processes waiting on links or Event pin.
M	Memory map	Display transputer memory map.
N	Network dump	Create core dump file.
P	Program boot	Simulate a program load.
Q	Quit	Quit simulator.
R	Run queues	Display active process queues.
S	Single step	Single step one transputer instruction.
T	Timer queues	Display timer queues.
U	Assign register	Assign value to register.
?	Help	Display help information.
? ‡	Query	Display registers and queue pointers.
. ‡	Where	Display next `Iptr`.
↑		Scroll display.
↓		Scroll display.
HELP		Display help information.
REFRESH		Redraw the screen.
FINISH		Quit simulator.
‡ Batch mode commands.		

Libraries

User libraries

Library	Description
`hostio.lib`	Host file server library.
`streamio.lib`	Stream i/o library.
`snglmath.lib`	Single length maths library.
`dblmath.lib`	Double length maths library.
`tbmaths.lib`	T400/T414/T425 optimised maths functions.
`string.lib`	String handling library.
`convert.lib`	Type conversion library.
`xlink.lib`	Extraordinary link handling library.
`crc.lib`	Block CRC library.
`debug.lib`	Debugging support library.
`callc.lib`	Mixed languages support library.
`msdos.lib`	DOS specific hostio library.

Include files

File	Contents
`hostio.inc`	Host file server constants.
`streamio.inc`	Stream i/o constants.
`mathvals.inc`	Maths and trigonometric constants.
`linkaddr.inc`	Transputer link addresses.
`ticks.inc`	Rates of the two transputer clocks.
`msdos.inc`	DOS specific hostio library constants.

Compiler libraries

File	Processor type
`occam2.lib`	T212/T222/T225/M212
`occama.lib`	T400/T414/T425/TA/TB
`occam8.lib`	T800/T801/T805
`occamutl.lib`	All.
`virtual.lib`	All.

The compiler libraries support all error modes. `occamutl.lib` contains routines called by some of the other compiler libraries. `virtual.lib` supports interactive debugging.

Bit manipulation functions

Result	Function	Parameter Specifiers
`INT`	`BITCOUNT`	`VAL INT Word, CountIn`
`INT`	`BITREVNBITS`	`VAL INT x, n`
`INT`	`BITREVWORD`	`VAL INT x`

2D block moves

Procedure	Parameter Specifiers
CLIP2D	VAL []BYTE Source, VAL INT sx, sy, []BYTE Dest, VAL INT dx, dy, width, length
DRAW2D	VAL []BYTE Source, VAL INT sx, sy, []BYTE Dest, VAL INT dx, dy, width, length
MOVE2D	VAL []BYTE Source, VAL INT sx, sy, []BYTE Dest, VAL INT dx, dy, width, length

CRC functions

Result	Function	Parameter Specifiers
INT	CRCWORD	VAL INT data, CRCIn, generator
INT	CRCBYTE	VAL INT data, CRCIn, generator

Supplementary arithmetic support functions

Result(s)	Function	Parameter specifiers
INT	FRACMUL	VAL INT x, y
INT, INT, INT	UNPACKSN	VAL INT x
INT	ROUNDSN	VAL INT Yexp, Yfrac, Yguard

Hostio library

#USE "hostio.lib"

Procedure	Parameter Specifiers
so.ask	CHAN OF SP fs, ts, VAL []BYTE prompt, replies, VAL BOOL display.possible.replies, VAL BOOL echo.reply, INT reply.number
so.buffer	CHAN OF SP fs, ts, from.user, to.user, CHAN OF BOOL stopper
so.close	CHAN OF SP fs, ts, VAL INT32 streamid, BYTE result
so.commandline	CHAN OF SP fs, ts, VAL BYTE all, INT length, []BYTE string, BYTE result
so.core	CHAN OF SP fs, ts, VAL INT32 offset, INT bytes.read, []BYTE data, BYTE result
so.date.to.ascii	VAL [so.date.len]INT date, VAL BOOL long.years, VAL BOOL days.first, [so.time.string.len]BYTE string
so.eof	CHAN OF SP fs, ts, VAL INT32 streamid, BYTE result
so.exit	CHAN OF SP fs, ts, VAL INT32 status
so.ferror	CHAN OF SP fs, ts, VAL INT32 streamid, INT32 error.no, INT length, []BYTE message, BYTE result
so.flush	CHAN OF SP fs, ts, VAL INT32 streamid, BYTE result

Procedure	Parameter Specifiers
so.fwrite.char	CHAN OF SP fs, ts, VAL INT32 streamid, VAL BYTE char, BYTE result
so.fwrite.hex.int	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT n, width, BYTE result
so.fwrite.hex.int32	CHAN OF SP fs, ts, VAL INT32 streamid, n, VAL INT width, BYTE result
so.fwrite.hex.int64	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT64 n, VAL INT width, BYTE result
so.fwrite.int	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT n, field, BYTE result
so.fwrite.int32	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT32 n, VAL INT field, BYTE result
so.fwrite.int64	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT64 n, VAL INT field, BYTE result
so.fwrite.nl	CHAN OF SP fs, ts, VAL INT32 streamid, BYTE result
so.fwrite.real32	CHAN OF SP fs, ts, VAL INT32 streamid, VAL REAL32 r, VAL INT Ip, Dp, BYTE result
so.fwrite.real64	CHAN OF SP fs, ts, VAL INT32 streamid, VAL REAL64 r, VAL INT Ip, Dp, BYTE result
so.fwrite.string	CHAN OF SP fs, ts, VAL INT32 streamid, VAL []BYTE string, BYTE result
so.fwrite.string.nl	CHAN OF SP fs, ts, VAL INT32 streamid, VAL []BYTE string, BYTE result

Procedure	Parameter Specifiers
so.getenv	CHAN OF SP fs, ts, VAL []BYTE name, INT length, []BYTE value, BYTE result
so.getkey	CHAN OF SP fs, ts, BYTE key, result
so.gets	CHAN OF SP fs, ts, VAL INT32 streamid, INT length, []BYTE data, BYTE result
so.multiplexor	CHAN OF SP fs, ts, []CHAN OF SP from.user, to.user, CHAN OF BOOL stopper
so.open	CHAN OF SP fs, ts, VAL []BYTE name, VAL BYTE type, mode, INT32 streamid, BYTE result
so.open.temp	CHAN OF SP fs, ts, VAL BYTE type, [so.temp.filename.length] BYTE filename, INT32 streamid, BYTE result
so.overlapped.buffer	CHAN OF SP fs, ts, from.user, to.user, CHAN OF BOOL stopper
so.overlapped.multiplexor	CHAN OF SP fs, ts, []CHAN OF SP from.user, to.user, CHAN OF BOOL stopper, []INT queue
so.overlapped.pri.multiplexor	CHAN OF SP fs, ts, []CHAN OF SP from.user, to.user, CHAN OF BOOL stopper, []INT queue

Procedure	Parameter Specifiers
so.parse.command.line	CHAN OF SP fs, ts, VAL []BYTE option.strings, VAL []INT option.parameters.required, []BOOL option.exists, [][2]INT option.parameters, INT error.len, []BYTE line
so.pollkey	CHAN OF SP fs, ts, BYTE key, result
so.popen.read	CHAN OF SP fs, ts, VAL []BYTE filename, VAL []BYTE path.variable.name, VAL BYTE open.type, INT full.len, []BYTE full.name, INT32 streamid, BYTE result
so.pri.multiplexor	CHAN OF SP fs, ts, []CHAN OF SP from.user, to.user, CHAN OF BOOL stopper
so.puts	CHAN OF SP fs, ts, VAL INT32 streamid, VAL []BYTE data, BYTE result
so.read	CHAN OF SP fs, ts, VAL INT32 streamid, INT length, []BYTE data
so.read.echo.any.int	CHAN OF SP fs, ts, INT n, BOOL error
so.read.echo.hex.int	CHAN OF SP fs, ts, INT n, BOOL error
so.read.echo.hex.int32	CHAN OF SP fs, ts, INT32 n, BOOL error
so.read.echo.hex.int64	CHAN OF SP fs, ts, INT64 n, BOOL error
so.read.echo.int	CHAN OF SP fs, ts, INT n, BOOL error
so.read.echo.int32	CHAN OF SP fs, ts, INT32 n, BOOL error

Procedure	Parameter Specifiers
so.read.echo.int64	CHAN OF SP fs, ts, INT64 n, BOOL error
so.read.echo.line	CHAN OF SP fs, ts, INT len, []BYTE line, BYTE result
so.read.echo.real32	CHAN OF SP fs, ts, REAL32 n, BOOL error
so.read.echo.real64	CHAN OF SP fs, ts, REAL64 n, BOOL error
so.read.line	CHAN OF SP fs, ts, INT len, []BYTE line, BYTE result
so.remove	CHAN OF SP fs, ts, VAL []BYTE name, BYTE result
so.rename	CHAN OF SP fs, ts, VAL []BYTE oldname, newname, BYTE result
so.seek	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT32 offset, origin, BYTE result
so.system	CHAN OF SP fs, ts, VAL []BYTE command, INT32 status, BYTE result
so.tell	CHAN OF SP fs, ts, VAL INT32 streamid, INT32 position, BYTE result
so.test.exists	CHAN OF SP fs, ts, VAL []BYTE filename, BOOL exists
so.time	CHAN OF SP fs, ts, INT32 localtime, UTCtime
so.time.to.ascii	VAL INT32 time, VAL BOOL long.years, VAL BOOL days.first [so.time.string.len]BYTE string
so.time.to.date	VAL INT32 input.time, [so.date.len]INT date

Procedure	Parameter Specifiers
so.today.ascii	CHAN OF SP fs, ts, VAL BOOL long.years, days.first, [so.time.string.len]BYTE string
so.today.date	CHAN OF SP fs, ts, [so.date.len]INT date
so.version	CHAN OF SP fs, BYTE version, host, os, board
so.write	CHAN OF SP fs, ts, VAL INT32 streamid, VAL []BYTE data, INT length
so.write.char	CHAN OF SP fs, ts, VAL BYTE char
so.write.hex.int	CHAN OF SP fs, ts, VAL INT n, width
so.write.hex.int32	CHAN OF SP fs, ts, VAL INT32 n, VAL INT width
so.write.hex.int64	CHAN OF SP fs, ts, VAL INT64 n, VAL INT width
so.write.int	CHAN OF SP fs, ts, VAL INT n, field
so.write.int32	CHAN OF SP fs, ts, VAL INT32 n, VAL INT field
so.write.int64	CHAN OF SP fs, ts, VAL INT64 n, VAL INT field
so.write.nl	CHAN OF SP fs, ts
so.write.real32	CHAN OF SP fs, ts, VAL REAL32 r, VAL INT Ip, Dp
so.write.real64	CHAN OF SP fs, ts, VAL REAL64 r, VAL INT Ip, Dp
so.write.string	CHAN OF SP fs, ts, VAL []BYTE string
so.write.string.nl	CHAN OF SP fs, ts, VAL []BYTE string

Streamio library

#USE "streamio.lib"

Procedure	Parameter Specifiers
ks.keystream.sink	CHAN OF KS keys
ks.keystream.to.scrstream	CHAN OF KS keyboard, CHAN OF SS scrn
ks.read.char	CHAN OF KS source, INT char
ks.read.int	CHAN OF KS source, INT number, char
ks.read.int64	CHAN OF KS source, INT64 number, INT char
ks.read.line	CHAN OF KS source, INT len, []BYTE line, INT char
ks.read.real32	CHAN OF KS source, REAL32 number, INT char
ks.read.real64	CHAN OF KS source, REAL64 number, INT char
so.keystream.from.file	CHAN OF SP fs, ts, CHAN OF KS keys.out, VAL []BYTE filename, BYTE result
so.keystream.from.kbd	CHAN OF SP fs, ts, CHAN OF KS keys.out, CHAN OF BOOL stopper, VAL INT ticks.per.poll
so.keystream.from.stdin	CHAN OF SP fs, ts, CHAN OF KS keys.out, BYTE result
so.scrstream.to.ANSI	CHAN OF SP fs, ts, CHAN OF SS scrn

Procedure	Parameter Specifiers
so.scrstream.to.file	CHAN OF SP fs, ts, CHAN OF SS scrn, VAL []BYTE filename, BYTE result
so.scrstream.to.stdout	CHAN OF SP fs, ts, CHAN OF SS scrn, BYTE result
so.scrstream.to.TVI920	CHAN OF SP fs, ts, CHAN OF SS scrn
ss.beep	CHAN OF SS scrn
ss.clear.eol	CHAN OF SS scrn
ss.clear.eos	CHAN OF SS scrn
ss.del.line	CHAN OF SS scrn
ss.delete.chr	CHAN OF SS scrn
ss.delete.chl	CHAN OF SS scrn
ss.down	CHAN OF SS scrn
ss.goto.xy	CHAN OF SS scrn, VAL INT x, y
ss.ins.line	CHAN OF SS scrn
ss.insert.char	CHAN OF SS scrn, VAL BYTE ch
ss.left	CHAN OF SS scrn
ss.right	CHAN OF SS scrn
ss.scrstream.copy	CHAN OF SS scrn.in, scrn.out
ss.scrstream.fan.out	CHAN OF SS scrn, screen.out1, screen.out2
ss.scrstream.from.array	CHAN OF SS scrn, VAL []BYTE buffer
ss.scrstream.multiplexor	[]CHAN OF SS screen.in, CHAN OF SS screen.out, CHAN OF INT stopper
ss.scrstream.sink	CHAN OF SS scrn

Procedure	Parameter Specifiers
ss.scrstream.to.array	CHAN OF SS scrn, []BYTE buffer
ss.up	CHAN OF SS scrn
ss.write.char	CHAN OF SS scrn, VAL BYTE char
ss.write.endstream	CHAN OF SS scrn
ss.write.hex.int	CHAN OF SS scrn, VAL INT number, field
ss.write.hex.int64	CHAN OF SS scrn, VAL INT64 number, VAL INT field
ss.write.int	CHAN OF SS scrn, VAL INT number, field
ss.write.int64	CHAN OF SS scrn, VAL INT64 number, VAL INT field
ss.write.nl	CHAN OF SS scrn
ss.write.string	CHAN OF SS scrn, VAL []BYTE str
ss.write.real32	CHAN OF SS scrn, VAL REAL32 number, VAL INT Ip, Dp
ss.write.real64	CHAN OF SS scrn, VAL REAL64 number, VAL INT Ip, Dp
ss.write.text.line	CHAN OF SS scrn, VAL []BYTE str

Single length maths library `#USE "snglmath.lib"`

Result(s)	Function	Parameter specifiers
REAL32	ACOS	VAL REAL32 X
REAL32	ALOG	VAL REAL32 X
REAL32	ALOG10	VAL REAL32 X
REAL32	ASIN	VAL REAL32 X
REAL32	ATAN	VAL REAL32 X
REAL32	ATAN2	VAL REAL32 X, VAL REAL32 Y
REAL32	COS	VAL REAL32 X
REAL32	COSH	VAL REAL32 X
REAL32	EXP	VAL REAL32 X
REAL32	POWER	VAL REAL32 X, VAL REAL32 Y
REAL32, INT32	RAN	VAL INT32 X
REAL32	SIN	VAL REAL32 X
REAL32	SINH	VAL REAL32 X
REAL32	TAN	VAL REAL32 X
REAL32	TANH	VAL REAL32 X

Double length maths library `#USE "dblmath.lib"`

Result(s)	Function	Parameter specifiers
REAL64	DACOS	VAL REAL64 X
REAL64	DALOG	VAL REAL64 X
REAL64	DALOG10	VAL REAL64 X
REAL64	DASIN	VAL REAL64 X
REAL64	DATAN	VAL REAL64 X
REAL64	DATAN2	VAL REAL64 X, VAL REAL64 Y
REAL64	DCOS	VAL REAL64 X
REAL64	DCOSH	VAL REAL64 X
REAL64	DEXP	VAL REAL64 X
REAL64	DPOWER	VAL REAL64 X, VAL REAL64 Y
REAL64, INT64	DRAN	VAL INT64 X
REAL64	DSIN	VAL REAL64 X
REAL64	DSINH	VAL REAL64 X
REAL64	DTAN	VAL REAL64 X
REAL64	DTANH	VAL REAL64 X

T400/T414/T425 maths library `#USE "tbmaths.lib"`

Contains the same functions as `snglmath.lib` and `dblmath.lib`, but optimised for the IMS T400, IMS T414 and IMS T425 procesoors.

String library

#USE "string.lib"

Result	Function	Parameter Specifiers
INT	char.pos	VAL BYTE search, VAL []BYTE str
INT	compare.strings	VAL []BYTE str1, str2
BOOL	eqstr	VAL []BYTE s1,s2
BOOL	is.digit	VAL BYTE char
BOOL	is.hex.digit	VAL BYTE char
BOOL	is.id.char	VAL BYTE char
BOOL	is.in.range	VAL BYTE char, bottom, top
BOOL	is.lower	VAL BYTE char
BOOL	is.upper	VAL BYTE char
INT, BYTE	search.match	VAL []BYTE possibles, str
INT, BYTE	search.no.match	VAL []BYTE possibles, str
INT	string.pos	VAL []BYTE search, str

Procedure	Parameter Specifiers
append.char	INT len, []BYTE str, VAL BYTE char
append.hex.int	INT len, []BYTE str, VAL INT number, field
append.hex.int64	INT len, []BYTE str, VAL INT64 number, VAL INT width
append.int	INT len, []BYTE str, VAL INT number, field
append.int64	INT len, []BYTE str, VAL INT64 number, VAL INT field
append.real32	INT len, []BYTE str, VAL REAL32 number, VAL INT Ip, Dp
append.real64	INT len, []BYTE str, VAL REAL64 number, VAL INT Ip, Dp
append.text	INT len, []BYTE str, VAL []BYTE text
delete.string	INT len, []BYTE str, VAL INT start, size, BOOL not.done
insert.string	VAL []BYTE new.str, INT len, []BYTE str, VAL INT start, BOOL not.done
next.int.from.line	VAL []BYTE line, INT ptr, number, BOOL ok
next.word.from.line	VAL []BYTE line, INT ptr, INT len, []BYTE word, BOOL ok
str.shift	[]BYTE str, VAL INT start, len, shift, BOOL not.done
to.lower.case	[]BYTE str
to.upper.case	[]BYTE str

Type conversion library

#USE "convert.lib"

Procedure	Parameter Specifiers
BOOLTOSTRING	INT len, []BYTE string, VAL BOOL b
HEXTOSTRING	INT len, []BYTE string, VAL INT n
HEX16TOSTRING	INT len, []BYTE string, VAL INT16 n
HEX32TOSTRING	INT len, []BYTE string, VAL INT32 n
HEX64TOSTRING	INT len, []BYTE string, VAL INT64 n
INTTOSTRING	INT len, []BYTE string, VAL INT n
INT16TOSTRING	INT len, []BYTE string, VAL INT16 n
INT32TOSTRING	INT len, []BYTE string, VAL INT32 n
INT64TOSTRING	INT len, []BYTE string, VAL INT64 n
REAL32TOSTRING	INT len, []BYTE string, VAL REAL32 X, VAL INT Ip, Dp
REAL64TOSTRING	INT len, []BYTE string, VAL REAL64 X, VAL INT Ip, Dp
STRINGTOBOOL	BOOL Error, b, VAL []BYTE string
STRINGTOHEX	BOOL Error, INT n, VAL []BYTE string
STRINGTOHEX16	BOOL Error, INT16 n, VAL []BYTE string
STRINGTOHEX32	BOOL Error, INT32 n, VAL []BYTE string
STRINGTOHEX64	BOOL Error, INT64 n, VAL []BYTE string
STRINGTOINT	BOOL Error, INT n, VAL []BYTE string
STRINGTOINT16	BOOL Error, INT16 n, VAL []BYTE string
STRINGTOINT32	BOOL Error, INT32 n, VAL []BYTE string
STRINGTOINT64	BOOL Error, INT64 n, VAL []BYTE string
STRINGTOREAL32	BOOL Error, REAL32 X, VAL []BYTE string
STRINGTOREAL64	BOOL Error, REAL64 X, VAL []BYTE string

Block CRC library

#USE "crc.lib"

Result	Function	Parameter Specifiers
INT	CRCFROMLSB	VAL []BYTE InputString VAL INT PolynomialGenerator, VAL INT OldCRC
INT	CRCFROMMSB	VAL []BYTE InputString, VAL INT PolynomialGenerator, VAL INT OldCRC

Link handling library

#USE "xlink.lib"

Procedure	Parameter Specifiers
InputOrFail.c	CHAN OF ANY c, []BYTE mess CHAN OF INT kill, BOOL aborted
InputOrFail.t	CHAN OF ANY c, []BYTE mess, TIMER t, VAL INT time, BOOL aborted
OutputOrFail.c	CHAN OF ANY c, VAL []BYTE mess, CHAN OF INT kill, BOOL aborted
OutputOrFail.t	CHAN OF ANY c, VAL []BYTE mess, TIMER t, VAL INT time, BOOL aborted
Reinitialise	CHAN OF ANY c

Debugging support library #USE "debug.lib"

Procedure	Parameter Specifiers
DEBUG.ASSERT	VAL BOOL assertion
DEBUG.MESSAGE	VAL []BYTE message
DEBUG.STOP	-
DEBUG.TIMER	CHAN OF INT stop

Mixed languages support library #USE "callc.lib"

Procedure	Parameter Specifiers
init.static	[] INT static.area, INT required.size, gsb
init.heap	VAL INT gsb, []INT heap.area
terminate.heap.use	VAL INT gsb
terminate.static.use	VAL INT gsb

DOS specific hostio library #USE "msdos.lib"

Procedure	Parameter Specifiers
dos.receive.block	CHAN OF SP fs, ts, VAL INT32 location, INT bytes.read, []BYTE block, BYTE result
dos.send.block	CHAN OF SP fs, ts, VAL INT32 location, VAL []BYTE block, INT len, BYTE result
dos.call.interrupt	CHAN OF SP fs, ts, VAL INT16 interrupt, VAL [dos.interrupt.regs.size] BYTE register.block.in, BYTE carry.flag, [dos.interrupt.regs.size] BYTE register.block.out, BYTE result
dos.read.regs	CHAN OF SP fs, ts, [dos.read.regs.size] BYTE registers, BYTE result
dos.port.read	CHAN OF SP fs, ts, VAL INT16 port.location, BYTE value, result
dos.port.write	CHAN OF SP fs, ts, VAL INT16 port.location, VAL BYTE value, BYTE result

Compiler library user functions

Function	Result(s)	Parameter specifiers
ABS	REAL32	VAL REAL32 x
ARGUMENT.REDUCE	BOOL, INT32, REAL32	VAL REAL32 x, y, y.err
ASHIFTRIGHT	INT	VAL INT argument, places
ASHIFTLEFT	INT	VAL INT argument, places
COPYSIGN	REAL32	VAL REAL32 x, y
DABS	REAL64	VAL REAL64 x
DARGUMENT.REDUCE	BOOL, INT32, REAL64	VAL REAL64 x, y, y.err
DCOPYSIGN	REAL64	VAL REAL64 x, y
DDIVBY2	REAL64	VAL REAL64 x
DFLOATING.UNPACK	INT, REAL64	VAL REAL64 x
DFPINT	REAL64	VAL REAL64 x
DIEEECOMPARE	INT	VAL REAL64 x, y
DISNAN	BOOL	VAL REAL64 x
DIVBY2	REAL32	VAL REAL32 x
DLOGB	REAL64	VAL REAL64 x
DMINUSX	REAL64	VAL REAL64 x
DMULBY2	REAL64	VAL REAL64 x
DNEXTAFTER	REAL64	VAL REAL64 x, y
DNOTFINITE	BOOL	VAL REAL64 x
DORDERED	BOOL	VAL REAL64 x, y
DSCALEB	REAL64	VAL REAL64 x, VAL INT n
DSQRT	REAL64	VAL REAL64 x
FLOATING.UNPACK	INT, REAL32	VAL REAL32 x
FPINT	REAL32	VAL REAL32 x

Function	Result(s)	Parameter specifiers
IEEE32OP	BOOL, REAL32	VAL REAL32 x, VAL INT rm, op, VAL REAL32 y
IEEE32REM	BOOL, REAL32	VAL REAL32 X, Y
IEEE64OP	BOOL, REAL64	VAL REAL64 x, VAL INT rm, op, VAL REAL64 y
IEEE64REM	BOOL, REAL64	VAL REAL64 X, Y
IEEECOMPARE	INT	VAL REAL32 x, y
ISNAN	BOOL	VAL REAL32 x
LOGB	REAL32	VAL REAL32 x
LONGADD	INT	VAL INT left, right, carry.in
LONGDIFF	INT, INT	VAL INT left, right, borrow.in
LONGDIV	INT, INT	VAL INT dividend.hi, dividend.lo, divisor
LONGPROD	INT, INT	VAL INT left, right, carry.in
LONGSUB	INT	VAL INT left, right, borrow.in
LONGSUM	INT, INT	VAL INT left, right, carry.in
MINUSX	REAL32	VAL REAL32 x
MULBY2	REAL32	VAL REAL32 x
NEXTAFTER	REAL32	VAL REAL32 x, y
NORMALISE	INT, INT, INT	VAL INT hi.in, lo.in
NOTFINITE	BOOL	VAL REAL32 x
ORDERED	BOOL	VAL REAL32 x, y
REAL32EQ	BOOL	VAL REAL32 x, y
REAL32GT	BOOL	VAL REAL32 x, y
REAL32OP	REAL32	VAL REAL32 x, VAL INT op, VAL REAL32 y

Function	Result(s)	Parameter specifiers
REAL32REM	REAL32	VAL REAL32 x, y
REAL64EQ	BOOL	VAL REAL64 x, y
REAL64GT	BOOL	VAL REAL64 x, y
REAL64OP	REAL64	VAL REAL64 x, VAL INT op, VAL REAL64 y
REAL64REM	REAL64	VAL REAL64 x, y
ROTATELEFT	INT	VAL INT argument, places
ROTATERIGHT	INT	VAL INT argument, places
SCALEB	REAL32	VAL REAL32 x, VAL INT n
SHIFTLEFT	INT, INT	VAL INT hi.in, lo.in, places
SHIFTRIGHT	INT, INT	VAL INT hi.in, lo.in, places
SQRT	REAL32	VAL REAL32 x

Dynamic code loading support procedures

Procedure	Parameter Specifiers
KERNEL.RUN	VAL []BYTE code, VAL INT entry.offset, []INT workspace, VAL INT no.of.parameters
LOAD.BYTE.VECTOR	INT here, []BYTE bytes
LOAD.INPUT.CHANNEL	INT here, CHAN OF ANY in
LOAD.INPUT.CHANNEL.VECTOR	INT here, []CHAN OF ANY in
LOAD.OUTPUT.CHANNEL	INT here, CHAN OF ANY out
LOAD.OUTPUT.CHANNEL.VECTOR	INT here, []CHAN OF ANY out

Transputer error flag manipulation

Procedure	Parameter Specifiers
CAUSEERROR	-
ASSERT	VAL BOOL test

Rescheduling priority process queue

Procedure	Parameter Specifiers
RESCHEDULE	-