UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

TESIS

PRINCIPIOS DE LA PROGRAMACIÓN PARALELA: UN ENFOQUE TEÓRICO Y PRÁCTICO AL LENGUAJE X10

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

PRESENTA:

ARMANDO RODRIGUEZ ARGUIJO

DIRECTORA DE TESIS
M.I. ELBA KAREN SAÉNZ GARCÍA

CIUDAD UNIVERSITARIA 21/05/2015.

Índice

1) Introducción.....7
1.1 ¿Por qué programar en paralelo?.....7
1.2 Características de un sistema distribuido.....9
1.3 Arquitecturas de supercómputo comunes.....10
2) Estado del arte de la programación paralela.....16
2.1) Programación paralela.....16
2.1.1) Clasificación de los algoritmos de acuerdo a la dependencia de tareas.....17
2.1.2) Dependencias estructurales.....20
2.1.2.1) Dependencia de datos.....20
2.1.2.2) Dependencia de control.....26
2.1.3) Concurrencia.....27
2.1.4) Sincronización.....28
2.1.5) Regiones condicionales críticas (Barreras).....28
2.1.6) Técnicas de paralelización.....28
2.1.7) Niveles de paralelismo.....30
2.1.8) Paralelización de datos.....30
2.1.9) Paralelización de ciclos.....31
2.1.10) Paralelización funcional.....31
2.1.11) Gráficas dirigidas de dependencia.....31
2.1.12) Matriz de dependencia.....32
2.2) Herramientas Actuales.....33
2.2.1)Balanceadores de carga.....33
2.2.2) Paso de mensajes.....34
2.2.3) Bibliotecas y directivas.....35
2.2.4) Lenguajes de programación.....36
3) Lenguaje de programación X10.....37
3.1) Funcionamiento.....38
3.2) Paradigma orientado a objetos.....40
3.3) Tipos de datos.....42
3.4) Estructuras de Control.....43
3.5) Expresiones.....47
3.6) Funciones.....53
3.7) Clases.....53
3.8) Estructuras.....56
3.9) Lugares.....58
3.10) Actividades.....59
3.11) Relojes.....60
3.12) Arreglos locales y distribuidos.....61
4) Ejemplos prácticos.....65
5)Conclusiones.....133
6) Apéndices.....135
6.1) Instalación y configuración de un entorno de desarrollo.....140
6.2) Instalación y configuración de X10 en arquitectura x86 y Sistema Operativo GNU\Linux.....144
7) Bibliografía.....148

Índice de figuras

Figura 1.1 Procesamiento Serial....8
Figura 1.2 Procesamiento Paralelo....10
Figura 1.3 Categorías de computadoras paralelas y distribuidas....11
Figura 1.4 Multiprocesador en Bus....12
Figura 1.5 Multiprocesador con conmutadores en malla....12
Figura 1.6 Multiprocesador con conmutadores....12
Figura 1.7 Multicomputadora en Bus....12
Figura 1.8 Multicomputadora con conmutadores....13
Figura 1.9 Multicomputadora con interconexión hipercubo....13
Figura 1.10 Memoria compartida (UMA)....14
Figura 1.11 Memoria distribuida (NUMA)....14
Figura 1.12 Dependencia entre arquitectura y algoritmo....15
Figura 2.1 Representación de la ejecución de un Algoritmo Serial....18
Figura 2.2 Representación de la ejecución de un Algoritmo Paralelo....19
Figura 2.3 Representación de la ejecución de un Algoritmo Serial-Paralelo....19
Figura 2.4 Lectura después de escritura (RAW)....21
Figura 2.5 Problema al Leer y escribir simultáneamente (RAW)....22
Figura 2.6 Problema al Escribir después de Leer (RAW)....22
Figura 2.7 Escritura después de lectura (WAR)....23
Figura 2.8 Problema al escribir y leer simultáneamente (WAR)....24
Figura 2.9 Problema al Leer después de escribir (WAR)....24
Figura 2.10 Escribir después de Escribir (WAW)....25
Figura 2.11 Problema al escribir simultáneamente (WAW)....26
Figura 2.12 Problema al escribir después de escribir (WAW)....26
Figura 2.13 Barreras en un algoritmo Serial-Paralelo....29
Figura 2.14 Modelo de paralelización....30
Figura 2.15 Gráfica dirigida de dependencia....33
Figura 2.16 Matriz de dependencia....34
Figura 3.1 Compilación en X10....39
Figura 3.2 Arquitectura de X10....40
Figura 3.3 Lugares en X10....59
Figura 3.4 Actividades en X10....60
Figura 3.5 Ejecución en 4 lugares....65
Figura 4.1 Gráfica del tiempo de ejecución del cálculo de Fibonacci en múltiples hilos....70
Figura 4.2 Gráfica del Speedup del cálculo de Fibonacci en múltiples hilos....71
Figura 4.3 Gráfica del tiempo de ejecución del cálculo de Fibonacci en múltiples lugares....72
Figura 4.4 Gráfica del Speedup del cálculo de Fibonacci en múltiples lugares....73
Figura 4.5 Gráfica del tiempo de ejecución del cálculo de Fibonacci en múltiples hilos y lugares....74
Figura 4.6 Gráfica del Speedup del cálculo de Fibonacci en múltiples hilos y lugares....76
Figura 4.7 Resultado de la ejecución del cálculo de Fibonacci en 4 lugares y 2 hilos, en Athlon x2 270 77
Figura 4.8 Salida del comando top durante la ejecución del cálculo de Fibonacci en 3 lugares y 1 hilo, en Athlon x2 270....77

Figura 4.9 Gráfica del tiempo de ejecución del cálculo de Fibonacci con recursividad en múltiples hilos 79
Figura 4.10 Gráfica del Speedup del cálculo de Fibonacci con recursividad en múltiples hilos............80
Figura 4.11 Gráfica del tiempo de ejecución del cálculo de Fibonacci con recursividad en múltiples lugares..........81
Figura 4.12 Gráfica del Speedup del cálculo de Fibonacci con recursividad en múltiples lugares.........82
Figura 4.13 Gráfica del tiempo de ejecución del cálculo de Fibonacci con recursividad en múltiples hilos y lugares..........84
Figura 4.14 Gráfica del Speedup del cálculo de Fibonacci con recursividad en múltiples hilos y lugares 86
Figura 4.15 Resultado de la ejecución del cálculo de Fibonacci con recursividad con 8 hilos, en Xeon 3.73Ghz..........87
Figura 4.16 Salida del comando top durante la ejecución del cálculo de Fibonacci con recursividad con 8 hilos, en Xeon 3.73Ghz..........87
Figura 4.17 Cálculo del número PI por la aproximación a la integral definida de 0 a 1........................88
Figura 4.18 Gráfica del tiempo de ejecución del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en múltiples hilos..........90
Figura 4.19 Gráfica del Speedup del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en múltiples hilos..........91
Figura 4.20 Gráfica del tiempo de ejecución del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en múltiples lugares..........92
Figura 4.21 Gráfica del Speedup de ejecución del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en múltiples lugares..........93
Figura 4.22 Gráfica del tiempo de ejecución del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en múltiples hilos y lugares..........95
Figura 4.23 Gráfica del Speedup del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en múltiples hilos y lugares..........97
Figura 4.24 Resultado de la ejecución del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en 2 lugares y 1 hilo, en Atom N270..........98
Figura 4.25 Salida del comando top durante la ejecución del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en 2 lugares y 1 hilo, en Atom N270..........98
Figura 4.26 Cálculo del número PI por el algoritmo de Montecarlo..........99
Figura 4.27 Gráfica del tiempo de ejecución del cálculo del número PI por el algoritmo de Montecarlo en múltiples hilos..........101
Figura 4.28 Gráfica del Speedup del cálculo del número PI por el algoritmo de Montecarlo en múltiples hilos..........102
Figura 4.29 Gráfica del tiempo de ejecución del cálculo del número PI por el algoritmo de Montecarlo en múltiples lugares..........103
Figura 4.30 Gráfica del Speedup del cálculo del número PI por el algoritmo de Montecarlo en múltiples lugares..........104
Figura 4.31 Gráfica del tiempo de ejecución del cálculo del número PI por el algoritmo de Montecarlo en múltiples hilos y lugares..........106
Figura 4.32 Gráfica del Speedup del cálculo del número PI por el algoritmo de Montecarlo en múltiples hilos y lugares..........108
Figura 4.33 Resultado de la ejecución del cálculo del número PI por el algoritmo de Montecarlo en 2 lugares y 1 hilo, en Atom N270..........109
Figura 4.34 Salida del comando top durante la ejecución del cálculo del número PI por el algoritmo de

Montecarlo en 2 lugares y 1 hilo, en Atom N270.....109
Figura 4.35 Suma de 2 matrices cuadradas.....110
Figura 4.36 Gráfica del tiempo de ejecución del cálculo de la suma de 2 matrices cuadradas en múltiples hilos.....112
Figura 4.37 Gráfica del Speedup del cálculo de la suma de 2 matrices cuadradas en múltiples hilos. .113
Figura 4.38 Gráfica del tiempo de ejecución del cálculo de la suma de 2 matrices cuadradas en múltiples lugares.....114
Figura 4.39 Gráfica del Speedup del cálculo de la suma de 2 matrices cuadradas en múltiples lugares 115
Figura 4.40 Gráfica del tiempo de ejecución del cálculo de la suma de 2 matrices cuadradas en múltiples hilos y lugares.....117
Figura 4.41 Gráfica del Speedup del cálculo de la suma de 2 matrices cuadradas en múltiples hilos y lugares.....119
Figura 4.42 Resultado de la ejecución del cálculo de la suma de 2 matrices cuadradas en 2 lugares, en Atom N270.....120
Figura 4.43 Salida del comando top durante la ejecución del cálculo de la suma de 2 matrices cuadradas en 8 lugares, en Atom N270.....120
Figura 4.44 Multiplicación de 2 matrices cuadradas.....121
Figura 4.45 Gráfica del tiempo de ejecución del cálculo de la multiplicación de 2 matrices cuadradas en múltiples hilos.....124
Figura 4.46 Gráfica del Speedup del cálculo de la multiplicación de 2 matrices cuadradas en múltiples hilos.....125
Figura 4.47 Gráfica del tiempo de ejecución del cálculo de la multiplicación de 2 matrices cuadradas en múltiples lugares.....126
Figura 4.48 Gráfica del Speedup del cálculo de la multiplicación de 2 matrices cuadradas en múltiples lugares.....127
Figura 4.49 Gráfica del tiempo de ejecución del cálculo de la multiplicación de 2 matrices cuadradas en múltiples hilos y lugares.....129
Figura 4.50 Gráfica del Speedup del cálculo de la multiplicación de 2 matrices cuadradas en múltiples hilos y lugares.....131
Figura 4.51 Resultado de la ejecución del cálculo de la multiplicación de 2 matrices cuadradas en 5 lugares, en Atom N270.....132
Figura 4.52 Salida del comando top durante la ejecución del cálculo de la multiplicación de 2 matrices cuadradas en 5 lugares, en Atom N270.....132
Figura 6.1 Página web de descarga de la máquina virtual de Java.....136
Figura 6.2 Descomprimir la máquina virtual de Java.....137
Figura 6.3 Listado del directorio de la máquina virtual de Java.....137
Figura 6.4 Conceder permisos de lectura y escritura al directorio de la máquina virtual de Java.....138
Figura 6.5 Instalación de la máquina virtual de Java.....138
Figura 6.6 Instalación del plug-in de la máquina virtual de Java para el navegador Firefox.....139
Figura 6.7 Página del lenguaje X10.....140
Figura 6.8 Página de descarga de X10dt.....140
Figura 6.9 Descompresión del archivo de X10dt.....141
Figura 6.10 Detalles de la descompresión del archivo X10dt.....141
Figura 6.11 Ejecución del entorno de X10dt.....142
Figura 6.12 Bienvenida al entorno de X10dt.....142
Figura 6.13 Hola Mundo en el entorno de desarrollo de X10.....143

Figura 6.14 Página de X10..144
Figura 6.15 Página de descarga de X10..144
Figura 6.16 Descompresión de X10...145
Figura 6.17 Detalles de la descompresión de X10...145
Figura 6.18 Listado del directorio de X10..146
Figura 6.19 Exportar la variable de entorno X10..146
Figura 6.20 Ejecución de X10c...147

1) Introducción

1.1 ¿Por qué programar en paralelo?

Desde el inicio del desarrollo de sistemas de cómputo se ha tratado de aumentar la velocidad de procesamiento utilizando diversas técnicas; una de ellas es el paralelismo.

El cómputo en paralelo o computación paralela ha existido desde hace décadas, sin embargo debido al auge de los microprocesadores y microcomputadoras el desarrollo en paralelo no ha tenido un gran avance respecto a otras tecnologías.
Actualmente la sociedad está mirando de nuevo el cómputo en paralelo para resolver problemas que necesitan grandes cálculos o velocidades superiores a utilizar solamente un sistema de cómputo, entiéndase sistema de cómputo como un sólo procesador que tiene una entrada y produce una salida.(véase figura 1.1)

Figura 1.1 Procesamiento Serial

Sin embargo al voltear a ver a la computación paralela nos encontramos con muchos retos que se han tratado de resolver desde el inicio de esta rama de la computación. Uno de ellos es la gran diversidad que se cuenta respecto a sistemas operativos, medios de interconexión, arquitecturas, bibliotecas, y lenguajes de programación.

La computación paralela siempre se ha asociado al cómputo científico, sin embargo las necesidades actuales requieren de nuevos programas, algoritmos y técnicas que permitan resolver problemas con el uso del cómputo paralelo; problemas como el manejo del tráfico automovilístico sincronizando semáforos, simulación de procesos naturales (como el flujo de corrientes de aire, ríos, mares, petróleo, etc..) , simulación del clima y desastres naturales. Así como también simulación y predicciones económicas que han utilizado cómputo paralelo para resolver las necesidades de diferentes áreas. Es por ello que actualmente se requiere de conocimiento para resolver problemas como el manejo de información distribuida y de gran tamaño (Petabytes), paralelizar servicios para obtener mayor confiabilidad y disponibilidad, mejores modelos de fenómenos naturales, económicos y sociales.

Por mucho tiempo se ha utilizado el cómputo serial para resolver problemas, sin embargo hay que recordar que todos los procesos naturales son continuos y paralelos; el simple hecho de platicar y caminar involucra procesos biológicos paralelos cómo el movimiento de las extremidades, el proceso de la respiración y el proceso del habla; y a su vez cada proceso involucra más actividades. Por esta razón el cómputo paralelo en el mundo se encuentra en desarrollo.

Existe mucha información del cómputo paralelo; inclusive cada año se realizan congresos

locales y mundiales, y se cuenta con grupos de trabajo de ACM (Association for Computing Machinery, Asociación para la maquinaria Informática) dedicados a la investigación. Sin embargo esta información la mayoría de las veces esta enfocada aplicaciones específicas o científicas, por ello la DARPA (Defense Advance Research Projects Agency, Agencia de Proyectos de Investigación Avanzados de Defensa) licitó un lenguaje que pudiera ser enseñado y que fuera lo suficientemente práctico para que más gente se adentre al mundo del cómputo paralelo. Al final de dicho concurso solamente quedaron 2 lenguajes, uno de nombre CHAPEL por CRAY INC. y otro X10 por IBM.

La principal diferencia de X10 radica en que su estructura, semántica y sintaxis es muy similar al lenguaje de programación JAVA, proporcionando una programación orientada a objetos y código objeto portable a diferentes plataformas mediante el uso de la máquina virtual.

El IEEE (Institute of Electrical and Electronics Engineers, Instituto de Ingeniería Eléctrica y Electrónica) define software paralelo como una “transferencia simultánea, ocurrencia o procesamiento de partes individuales de un todo,como los bits de un carácter y los caracteres de una palabra utilizando diferentes recursos para las diferentes partes” (Gebali 2011).Así que se puede decir que un algoritmo es paralelo cuando 2 o más partes independientes del algoritmo se pueden ejecutar simultáneamente en hardware, presuponiendo que existe el hardware correspondiente.

Así pues, se puede establecer que el paralelismo en software está fuertemente ligado al hardware en el cual será ejecutado. Por lo cual es tarea del programador, compilador o sistema operativo proveer tareas a las diferentes unidades de procesamiento para mantenerlas ocupadas. Actualmente existen ejemplos de algoritmos paralelos en las áreas como:

- Cómputo científico, simulaciones físicas, resolución de ecuaciones diferenciales, simulaciones de túneles de viento, y simulaciones del clima.

- Cómputo gráfico, procesamiento de imágenes, compresión de video,etc..

- Imágenes médicas, como resonancias magnéticas (MRI) y tomografías computarizadas (CT).

Sin embargo, existen muchas áreas en las cuales no es una tarea trivial el diseñar/reconocer algoritmos paralelos como en la minería de datos, almacenamiento de datos, banca en línea, aplicaciones administrativas,etc...
El reto actual es desarrollar arquitecturas y software paralelo que permita mejorar todo tipo de procesos.

1.2 Características de un sistema distribuido

Los términos de paralelismo y concurrencia algunas veces se puede pensar que son sinónimos, sin embargo esto no es correcto.

Según Breshears (2009) un sistema se dice que es concurrente si puede soportar 2 o más procesos en progreso al mismo tiempo,y un sistema es paralelo si puede soportar 2 o más procesos ejecutándose simultáneamente.
Una aplicación concurrente puede tener 2 o más hilos, los cuales pueden pasar al estado de ejecución en un solo procesador, de acuerdo a las políticas del sistema operativo, de acuerdo a Breshears(2009) a esto se le llama en progreso.

En una ejecución en paralelo, deben de existir múltiples núcleos o procesadores, en tal caso los hilos pueden ser asignados a diferentes núcleos.

Se puede deducir que el paralelismo es una derivación de la concurrencia. Al escribir aplicaciones concurrentes que utilizan múltiples hilos o procesos, y que se ejecutan en múltiples procesadores o núcleos, se hablaría de paralelismo.(véase figura 1.2)

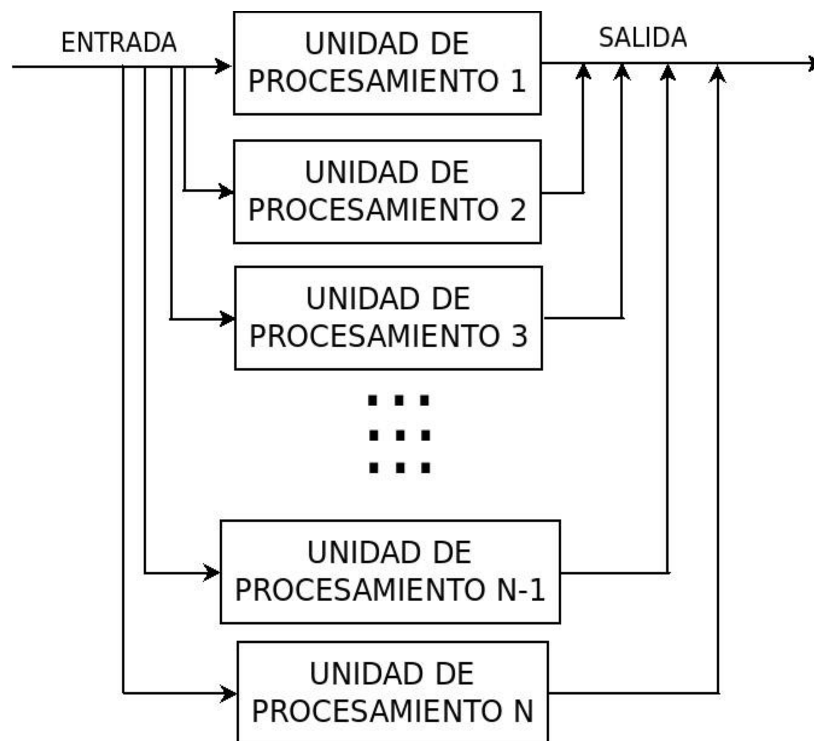


Figura 1.2 Procesamiento Paralelo

1.3 Arquitecturas de supercómputo comunes

De acuerdo a Alan Clements (2006), una arquitectura de cómputo es una vista abstracta de la computadora, que describe que puede hacer. Se puede decir que la arquitectura es similar la especificación funcional. La organización de una computadora describe cómo es implementada una arquitectura. Por ejemplo se pueden construir 2 computadoras con

arquitectura de 32bits, una puede hacer la suma de 2 números en una operación, mientras la otra computadora realiza la suma utilizando registros de 16bits; por lo cual tardará más. Al final será el mismo resultado. Las computadoras tienen la misma arquitectura pero diferente organización.

En la computación paralela-distribuida podemos hablar de arquitecturas de supercómputo de acuerdo a su interconexión y al tipo de acceso a la memoria RAM.

Por tipo de interconexión
Andrew Tanenbaum, propone una clasificación de acuerdo al tipo de conexión y al tipo de acceso a la memoria RAM.
Podemos observar en la Figura 1.3 que existen una división entre multiprocesadores y multicomputadoras.
Los multiprocesadores comparten la memoria RAM a través del uso de interconexiones muchas veces propietarias. Un ejemplo clásico de este tipo de computadoras, son los MainFrames y los sistemas SMP.
Las multicomputadoras no comparten la memoria RAM, para este tipo de computadoras muchas veces se utiliza RPC (Remote Procedure Call), paso de mensajes o alguna otra técnica como el modelo PRAM. Los ejemplos clásicos son los clusters beowulf y grids.
En cuestión de costo, las multicomputadoras son mucho más baratas de construir, debido a que por lo regular los componentes utilizados son más comunes. Podemos hablar de multicomputadoras o supercomputadoras construidas con procesadores X86 y redes Ethernet.

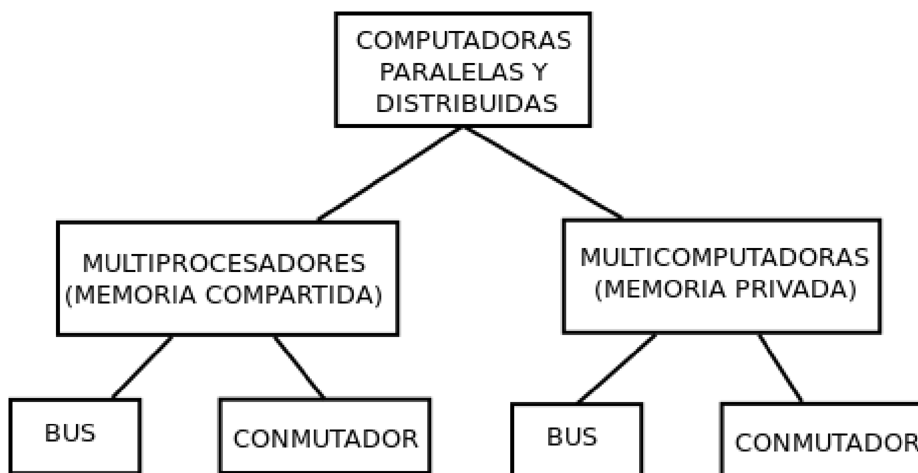


Figura 1.3 Categorías de computadoras paralelas y distribuidas

A
continuación se describen a más detalle

Multiprocesadores en bus
En esta clasificación, cada procesador está conectado a la memoria RAM a través de un sólo canal o bus. En este tipo de arquitectura, puede resultar que el bus se convierta en el cuello de botella si el ancho de banda no fue calculado correctamente y/o el algoritmo implementado de comunicación no es lo suficiente robusto.(véase figura 1.4)

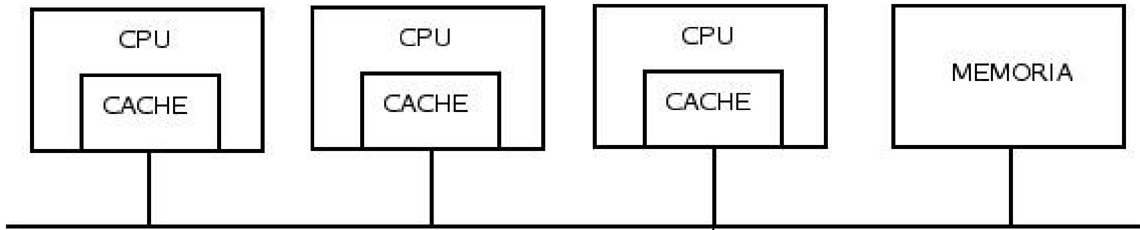


Figura 1.4 Multiprocesador en Bus

Multiprocesadores con conmutador

En esta clasificación se pueden reducir los cuellos de botella, utilizando conmutadores o actualmente chips de ruteo. Dependiendo del tipo de conmutador,el costo aumenta y es mucho mayor a la interconexión por bus.(véase figura 1.5 y 1.6)

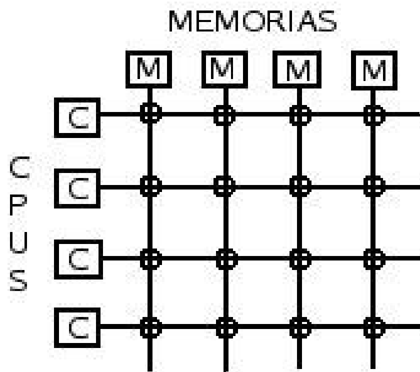


Figura 1.5 Multiprocesador con conmutadores en malla

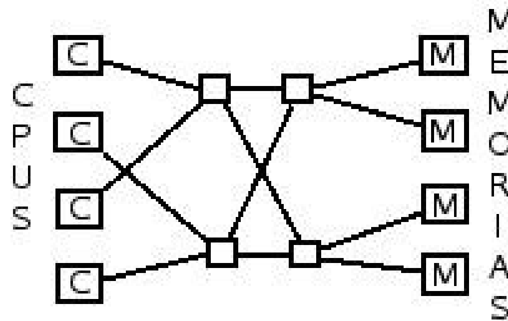


Figura 1.6 Multiprocesador con conmutadores

Multicomputadoras en bus

Las multicomputadoras en bus, es la arquitectura más barata que se puede utilizar. Cada nodo de cómputo consta de uno o varios CPU's y una memoria local (se podría hablar de una PC).Todos los nodos de cómputo están interconectados por un bus, al igual que los multiprocesadores en bus, existe el riesgo de que el bus sea un cuello de botella.(véase figura 1.7)

Esta arquitectura, es muy utilizada en la actualidad,sobre todo en cluster's beowulf.

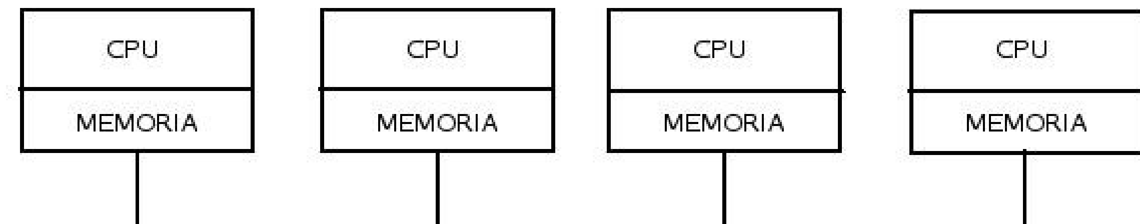


Figura 1.7 Multicomputadora en Bus

Multicomputadoras con conmutador

En esta arquitectura, los nodos de cómputo se componen de uno o varios procesadores y memoria RAM local. Los nodos están interconectados entre sí mediante conmutadores o

ruteadores, estos dispositivos puedes ser chips de red y/o ruteadores de red. Este tipo de arquitectura es un poco cara por el tipo y cantidad de conmutadores que se utilicen. Aunque es muy poco probable que exista un cuello de botella,su implementación es relativamente compleja dependiendo del algoritmo de ruteo que se utilice.(véase figura 1.8)
En la figura 1.9 se puede observar que existen diferentes rutas para llegar de un nodo a otro,esta interconexión se llama hipercubo.

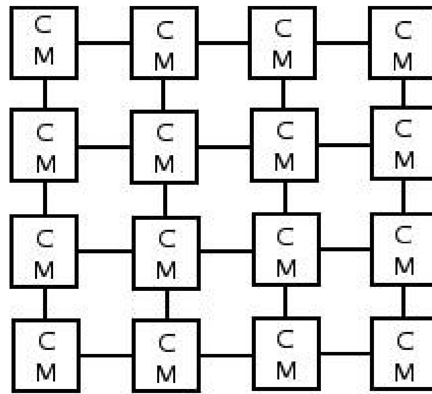


Figura 1.8 Multicomputadora con conmutadores

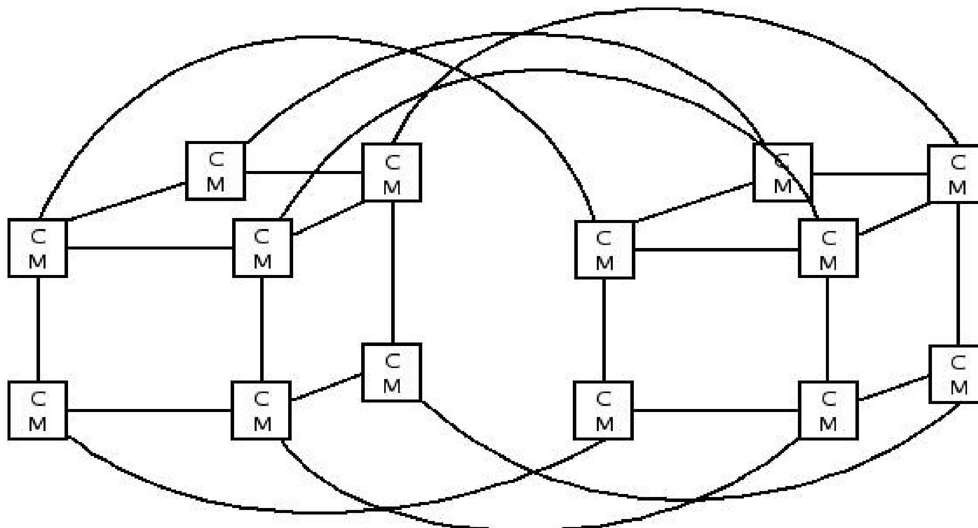


Figura 1.9 Multicomputadora con interconexión hipercubo

Por tipo de Memoria

Los procesadores de memoria compartida también conocidos como máquinas de acceso paralelo aleatorio (PRAMs) ,son muy populares debido a su simple modelo de programación, el cual permite el desarrollo simple de software paralelo.

UMA (Uniform Memory Access)

Este tipo de máquinas utilizan una memoria compartida o un espacio de direcciones compartidas para la comunicación entre los procesadores, este tipo de sistemas también se les llama sistema de acceso a memoria uniforme (UMA).
Todos los procesadores pueden acceder a la memoria compartida a través de una red de interconexión. Típicamente la red es un bus, pero en sistemas más grandes se utiliza otro tipo de red .(véase figura 1.10)
Se puede identificar que un cuello de botella de este tipo de sistemas es el ancho de banda del bus o de la red, por lo cual se utilizan redes de alta velocidad; otro problema es la coherencia del cache, para lo cual existen protocolos que permiten tener coherencia de datos en todos los procesadores.

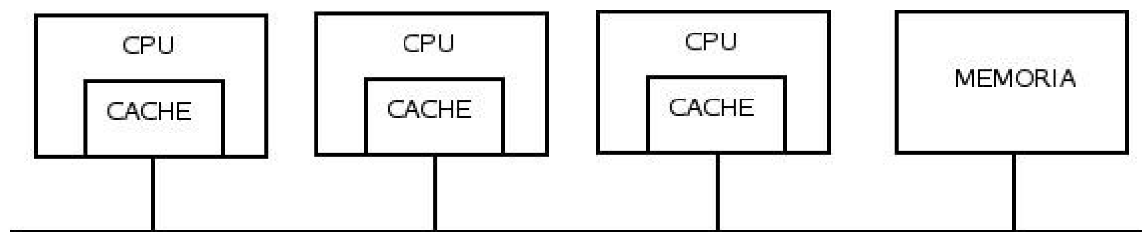


Figura 1.10 Memoria compartida (UMA)

NUMA (NonUniform Memory Access)

En un sistema de memoria distribuida con acceso no uniforme, cada módulo de memoria está asociado a un procesador, cada procesador tiene acceso a su propia memoria física. Por lo cual se puede decir que el acceso a memoria no es uniforme,ya que depende de que módulo de memoria se quiera acceder.(véase figura 1.11)
En este tipo de sistemas se utiliza un mecanismo de paso de mensajes (MP) para que los procesadores accedan a las otras memorias del sistema. Actualmente existe un protocolo de comunicación independiente del lenguaje de programación , llamado Interfaz de Paso de Mensaje (MPI).
Si el sistema se compone de procesadores idénticos (mismas características) , se dice que es un sistema de multiprocesadores simétricos (SMP),en caso contrario contaríamos con un sistema de multiprocesadores asimétricos (ASMP).
En el caso de que la red de interconexión del sistema distribuido sea global, como el internet, el sistema usualmente contendrá miles de computadoras, y será llamado computadora masivamente paralela, computadora distribuida o computadora en malla (GRID).

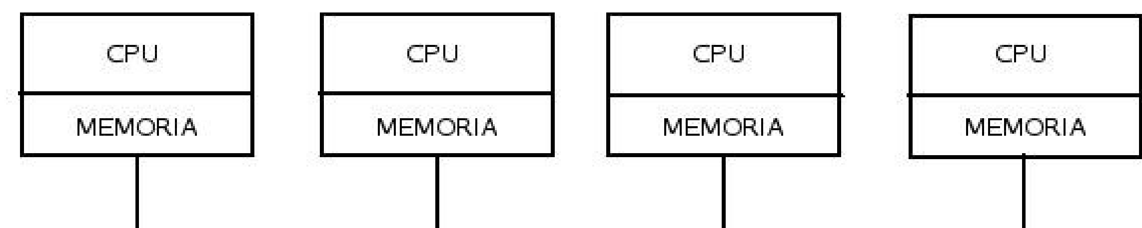


Figura 1.11 Memoria distribuida (NUMA)

De acuerdo con Gebali, se puede decir que tanto la arquitectura de la computadora paralela o distribuida y el algoritmo a implementar, están estrechamente ligados.(véase figura 1.12)

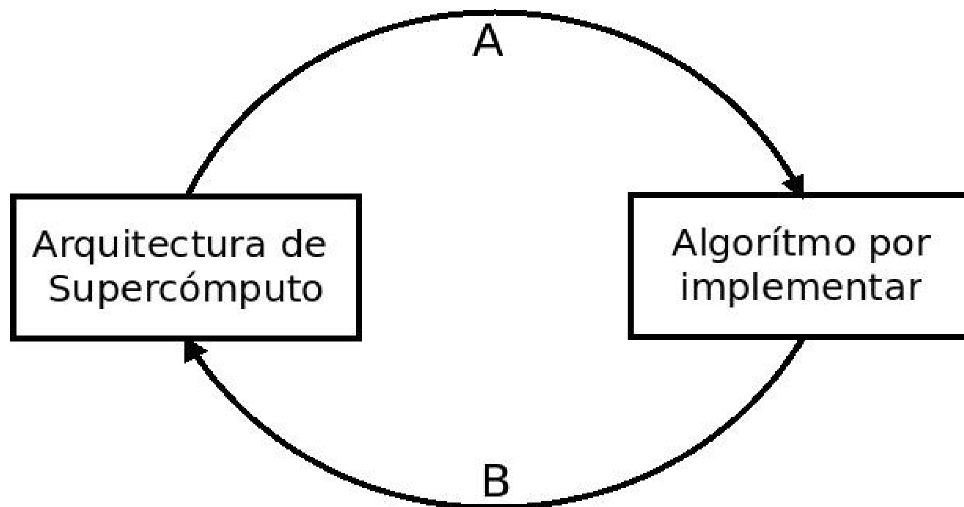


Figura 1.12 Dependencia entre arquitectura y algoritmo

En caso de que se tome el camino B, se parte de un algoritmo a implementar, será necesario conocer en qué tipo de arquitectura de computadora paralela o distribuida se ejecutará y de ser posible adaptarla.
En el caso A, se parte de que se cuenta con una arquitectura de computadora paralela o distribuida y se desea implementar un algoritmo; en este caso el algoritmo se tendrá que adaptar.

Actualmente con el uso de nuevos lenguajes de programación, compiladores y máquinas virtuales; esta relación se está tratando de eliminar, lo cual permitiría crear aplicaciones totalmente portables a diferentes arquitecturas de computadoras paralelas o distribuidas. Uno de estos lenguajes es X10.

2) Estado del arte de la programación paralela

2.1) Programación paralela

Se dice que es imposible aumentar el rendimiento de un sólo procesador,ya que dicho procesador consumiría demasiada energía y a su vez requeriría un sistema de refrigeración más costoso. Fayez Gebali (2011) opina que es más práctico utilizar un conjunto de procesadores para obtener el rendimiento esperado.

De acuerdo a observaciones realizadas por diferentes personas, se puede decir que si una aplicación serial no se ejecuta de manera rápida en un sólo procesador, se ejecutará aún más lento en un conjunto de procesadores (Gebali, 2011),a menos que la aplicación sea modificada utilizando técnicas de paralelismo.

Un reto que a los programadores se les presenta en el diseño y programación de aplicaciones paralelas,es que la mayoría se encuentran acostumbrados a solamente utilizar un sólo CPU y a un procesamiento secuencial (Gebali,2011).

En la actualidad se cuenta con compiladores que buscan dentro del código fuente , simples ciclos (loops) y los dividen de acuerdo al número de procesadores. Dichos compiladores pueden utilizarse para algoritmos llamados embarazosamente paralelos(Gebali,2011).Para el resto de los algoritmos, el programador es el encargado de utilizar las diferentes técnicas de paralelización para obtener los mejores resultados.

Dentro de los algoritmos paralelos, las tareas o procesos deben de ser independientes. Algunas tareas pueden ejecutarse en paralelo y otras en forma serial. Es decir, los algoritmos paralelos cuentan con una parte serial y otra paralela (Gebali,2011).Se pretende que la mayor parte del algoritmo sea paralelo.

Los componentes básicos que definen a un algoritmo son:

- Las diferentes tareas
- La dependencia entre las tareas
- El flujo de datos de entrada
- El flujo de datos de salida

De acuerdo a la ley de Amdhal, se presenta un límite en la aceleración de la ejecución de un algoritmo paralelo. (Gentle ,2004)

La idea básica de la ley de Amdhal se encuentra en asegurar que siempre existirá una parte serial la cual de acuerdo a la siguiente fórmula, afectará la aceleración esperada.

$$Speedup = \frac{1}{F + \frac{(1-F)}{N}}$$

F= Fracción del algoritmo que es serial
1-F= Fracción del algoritmo que es paralelo
N= Número de procesadores

Si el número de procesadores tiende a infinito, la aceleración que se puede alcanzar es 1/F

2.1.1) Clasificación de los algoritmos de acuerdo a la dependencia de tareas

De acuerdo con Gebali (2011), se puede considerar la siguiente clasificación de algoritmos.

- **Seriales**
- **Paralelos**
- **Seriales-Paralelos (SPA)**

Seriales

Un algoritmo serial es aquel en el que las tareas se ejecutan una tras otra, debido a su dependencia de datos.(Gebali 2011) (véase figura 2.1)

Estos algoritmos no se pueden paralelizar ya que deben de ejecutarse de manera secuencial. La única paralelización posible es cuando cada tarea es dividida en pequeñas sub-tareas.(Gebali 2011)

Figura 2.1 Representación de la ejecución de un Algoritmo Serial

Paralelos

Los algoritmos paralelos son aquellos en los cuales las tareas se pueden ejecutar en paralelo al mismo tiempo, debido a su independencia de datos.(Gebali 2011) (véase figura 2.2)

Un ejemplo es un servidor web que puede encargarse de diferentes peticiones de manera simultánea, debido a que cada petición es independiente de las demás.(Gebali 2011)

Otro ejemplo es el manejo de tareas independientes que realiza un sistema operativo multitarea,dicho sistema operativo puede ejecutar diferentes aplicaciones como navegadores web, procesadores de texto, reproductores multimedia,etc... en diferentes unidades de procesamiento.(Gebali 2011)

Estos algoritmos son fáciles de paralelizar debido a que todas las tareas son independientes y por lo tanto se pueden ejecutar en paralelo.(Gebali 2011)

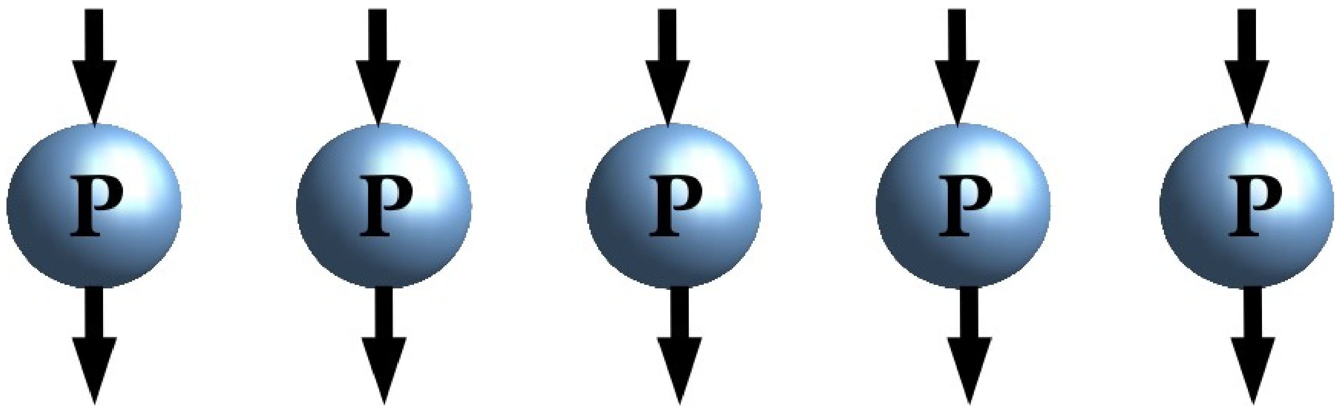


Figura 2.2 Representación de la ejecución de un Algoritmo Paralelo

Seriales-Paralelos (SPAs)

Un algoritmo SPA es aquel en el que las tareas están agrupadas en fases y se ejecutan en paralelo,además las fases se ejecutan de manera secuencial.(Gebali 2011) (véase figura 2.3)

Un algoritmo SPA se convierte en paralelo cuando el número de fases es mayor de uno, y se convierte en serial cuando el número de tareas dentro de cada fase es uno.(Gebali 2011).

Estos algoritmos se pueden paralelizar asignando cada tarea de una fase a una unidad de procesamiento. Las fases no se pueden paralelizar ya que su naturaleza serial, no se los permite.(Gebali 2011).

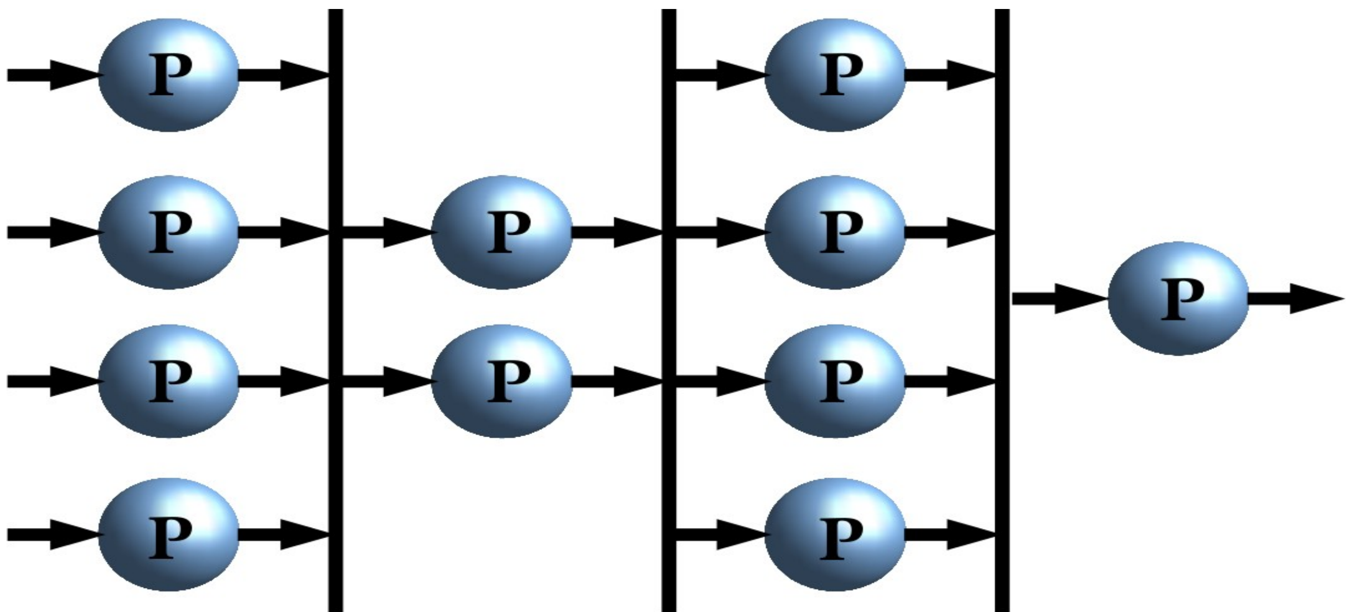


Figura 2.3 Representación de la ejecución de un Algoritmo Serial-Paralelo

El diseño de un sistema de cómputo paralelo requiere considerar muchas opciones. El diseñador debe escoger una arquitectura de procesamiento que sea capaz de realizar las tareas deseadas. El conjunto de procesadores puede estar compuesto de procesadores

simples y/o procesadores superescalares. Dichos procesadores deben poder comunicarse entre ellos, para lo cual se utiliza una red de interconexión.

La red de interconexión es un elemento que si no puede soportar la “carga”, se producirá un “cuello de botella” y el rendimiento del sistema de cómputo paralelo se verá mermado. Por esta razón existen múltiples formas de interconexión, siendo la conexión en Bus la más sencilla;actualmente se utilizan las redes en chip (NoC), que intercambian datos entre chips en formas de paquetes dirigidos a través de routers.

Otro punto importante de considerar en el diseño de sistemas de cómputo paralelo es el sistema de memoria. Los datos y programas deben estar almacenados en algún tipo de sistema de memoria, el diseñador tiene la opción de contar con módulos de memoria compartidos entre los procesadores y contar con módulos dedicados a cada procesador. Cuando los procesadores comparten la memoria, se necesitan implementar mecanismos para leer,escribir y cuidar la integridad de los datos.
(Gebali 2011).

2.1.2) Dependencias estructurales

Las dependencias estructurales se producen cuando una instrucción necesita un recurso de hardware, y ese recurso no está disponible, porque está siendo utilizado por otra instrucción. La instrucción no podrá continuar con su ejecución hasta que se resuelva esta dependencia, lo que retrasa al proceso.(Rodríguez Clemente, 2000).

Existen 2 categorías de dependencias estructurales, de datos y de control.

2.1.2.1) Dependencia de datos

Se dice que existe dependencia de datos cuando 2 o más instrucciones utilizan algún operando común (un registro o una localidad de memoria).(Rodríguez Clemente, 2000).

Existen 3 tipos de dependencia de datos, lectura después de escritura (Read After Write RAW) , escritura después de lectura (Write After Read WAR) y escritura después de escritura (Write After Write WAW)

Lectura después de escritura (RAW)

De acuerdo a Rodríguez Clemente, esta dependencia también llamada dependencia verdadera o dependencia de Rango-Dominio, se produce cuando una instrucción j necesita un dato como lectura que le debe proporcionar otra instrucción i previa. Al segmentar la ejecución (ejecutar en paralelo) de las instrucciones, podría producirse un conflicto si se intenta leer el dato antes de que sea actualizado por la instrucción previa.

En las técnicas de software para eliminar este tipo de dependencias, una muy utilizada es la

re-ordenación de código, que supone separar lo máximo posible las instrucciones con dependencias. Para ello se intercalan entre las instrucciones dependientes aquellas instrucciones que no cambien la semántica del programa. En el caso de que el compilador no encontrase ninguna instrucción, la solución es incluir operaciones no operación (NOP).

En la figura 2.4 se puede observar que en el tiempo 1, el proceso 1 escribe el dato A y en el tiempo 2 el proceso 2 lee el dato A.

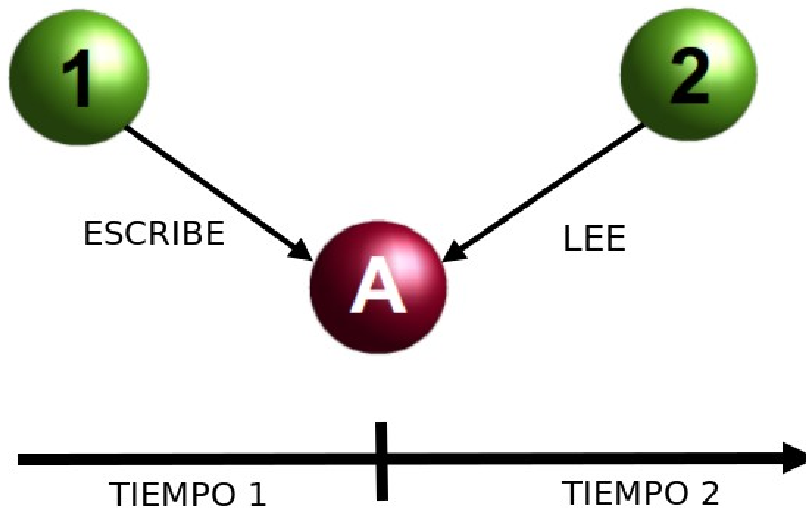


Figura 2.4 Lectura después de escritura (RAW)

En caso de que el proceso 1 y 2 se ejecutaran de forma paralela en el tiempo 1 como se muestra en la figura 2.5. No se podría definir que es lo qué el proceso 2 leería.

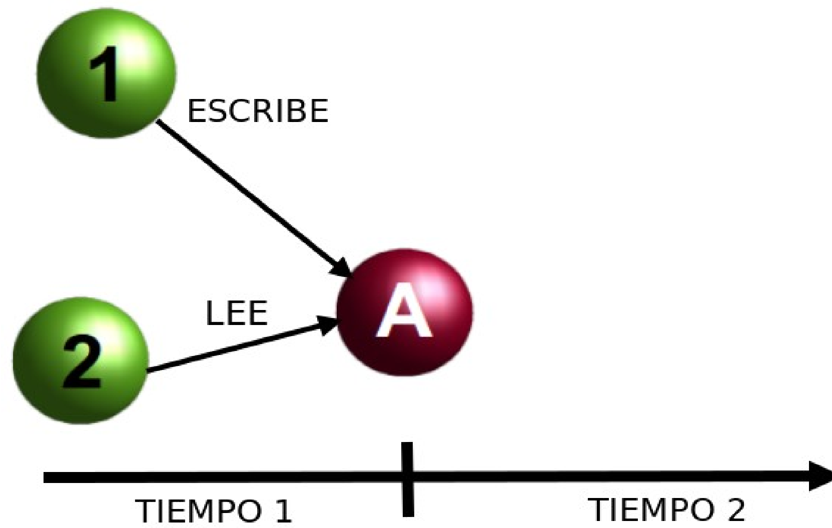


Figura 2.5 Problema al Leer y escribir simultáneamente (RAW)

En caso de que el proceso 2 lea el dato A antes de que el proceso 1 lo escriba, figura 2.6. No se podría saber que es lo que lee el proceso 2.

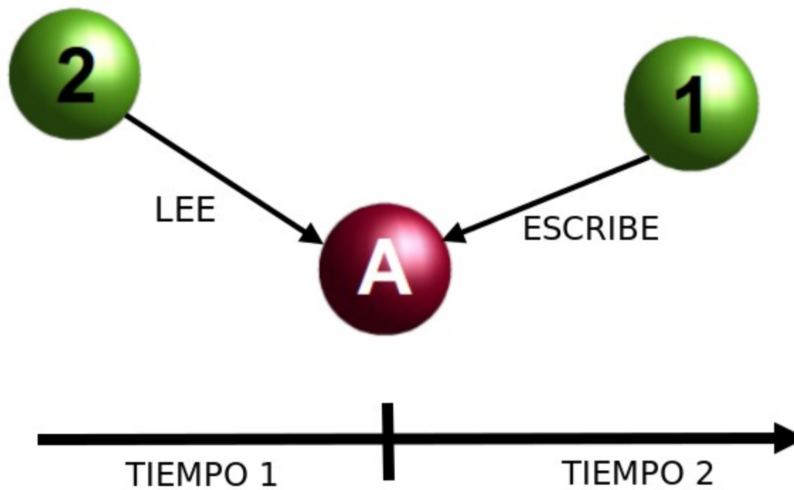


Figura 2.6 Problema al Escribir después de Leer (RAW)

Tomando en consideración la figura 2.5 y 2.6, se puede decir que no se puede paralelizar el proceso 1 o 2, ya que los resultados no serían los esperados.

Escritura después de lectura (WAR)

También llamada anti-dependencia o dependencia Dominio-Rango. Se produce cuando una instrucción k intenta escribir un dato antes de que lo lea otra instrucción j previa. Si esto ocurre se producirá un conflicto a que la instrucción j leería el dato modificado.

El compilador puede eliminar gran parte de este tipo de dependencias cambiando el nombre del registro destino,o cambiando el nombre de variable en caso de la programación paralela.

En la figura 2.7 se puede observar que en el tiempo 1, el proceso 1 lee el dato A y en el tiempo 2 el proceso 2 escribe el dato A.

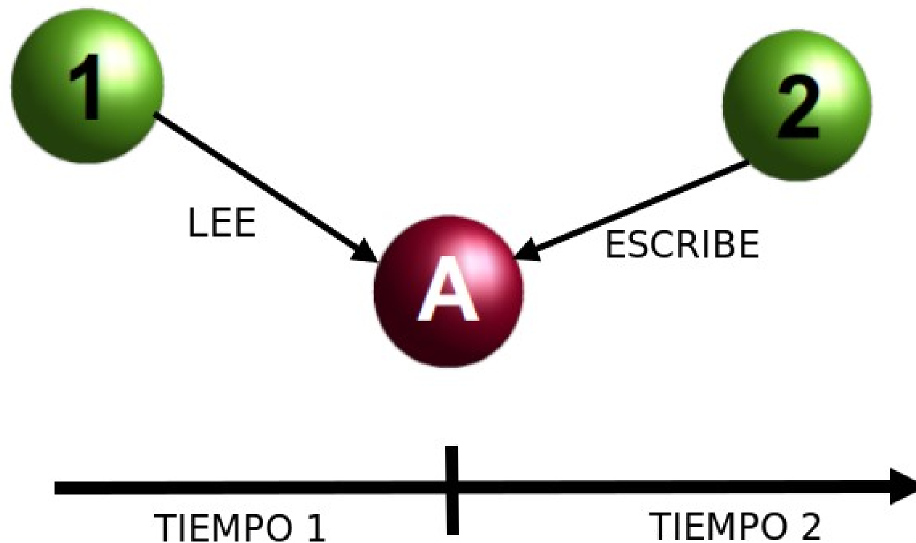


Figura 2.7 Escritura después de lectura (WAR)

En caso de que el proceso 1 y 2 se ejecutaran de forma paralela en el tiempo 1, figura 2.8.En el tiempo 2 no se puede asegurar que el dato A corresponda a la escritura del proceso 2, ni que el proceso 1 haya leído el dato A antes de la escritura del proceso 2.

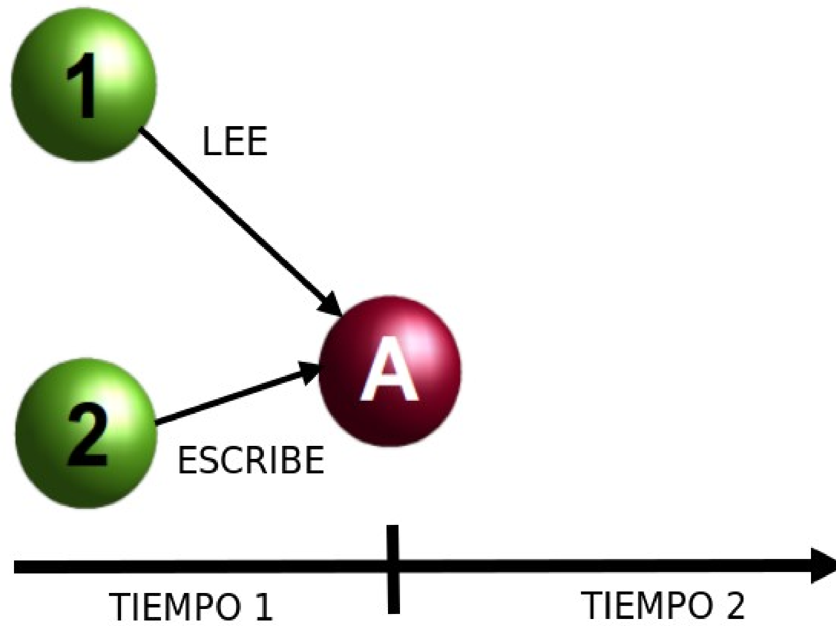


Figura 2.8 Problema al escribir y leer simultáneamente (WAR)

En caso de que el proceso 2 se ejecute en el tiempo 1,figura 2.9.El proceso 1 leería el dato A que acaba de escribir el proceso 2,lo cual no es lo deseado.

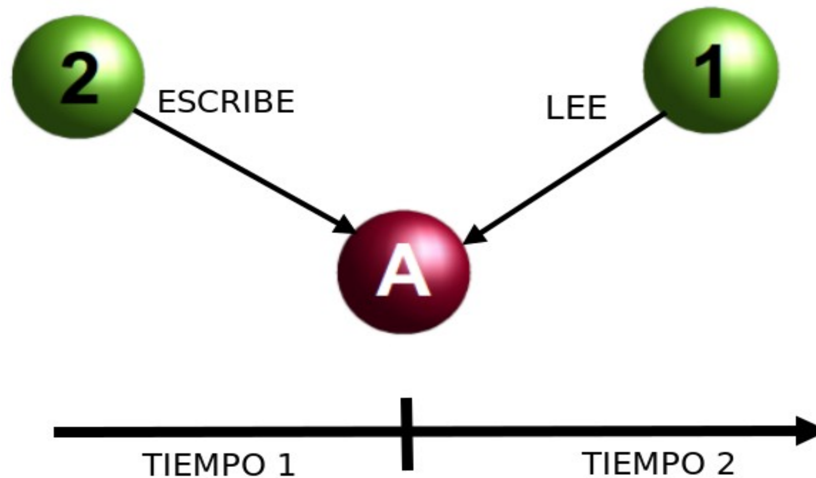


Figura 2.9 Problema al Leer después de escribir (WAR)

Tomando en consideración la figura 2.8 y 2.9, se puede decir que no se puede paralelizar el proceso 1 o 2, ya que los resultados no serían los esperados.

Escritura después de escritura (WAW)

También conocida como dependencia de salida o Rango-Rango. Ocurre cuando una instrucción k intenta escribir sobre un registro o memoria antes de que escriba otra instrucción previa. Si se permite, quedaría en el registro o memoria el resultado de la instrucción i y no el resultado de la k tal y como la semántica del programa lo estipula.

La forma de resolver este conflicto es la misma que la que se aplica para conflictos WAR,el compilador puede cambiar el nombre del registro destino,o cambiar el nombre de variable en caso de la programación paralela.

En la figura 2.10 podemos observar una dependencia WAW. En el tiempo 1, el proceso 1 escribe en A y en el tiempo 2, el proceso 2 escribe en A.

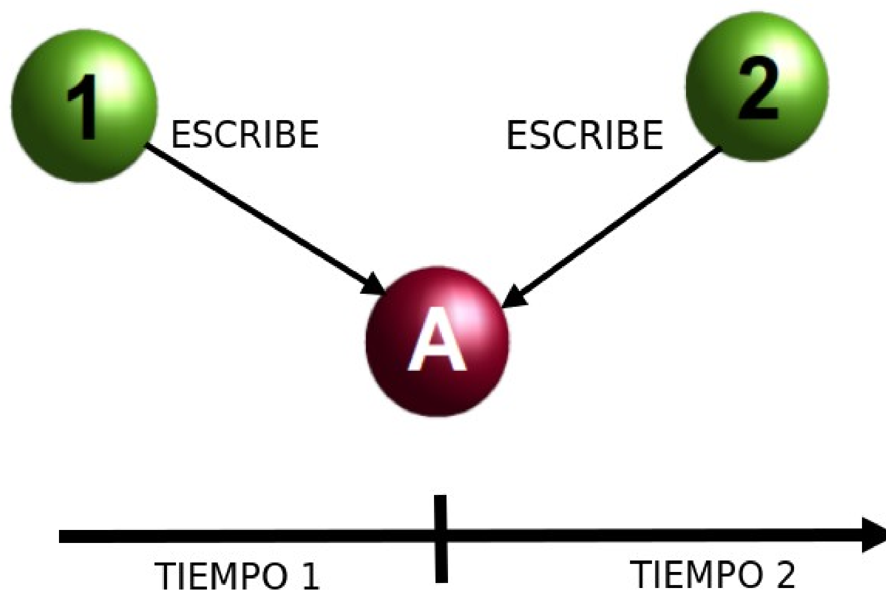


Figura 2.10 Escribir después de Escribir (WAW)

En caso de que el proceso 1 y 2 se ejecutaran de forma simultanea,figura 2.11.En el tiempo 2 no se podría decir que valor contiene A.

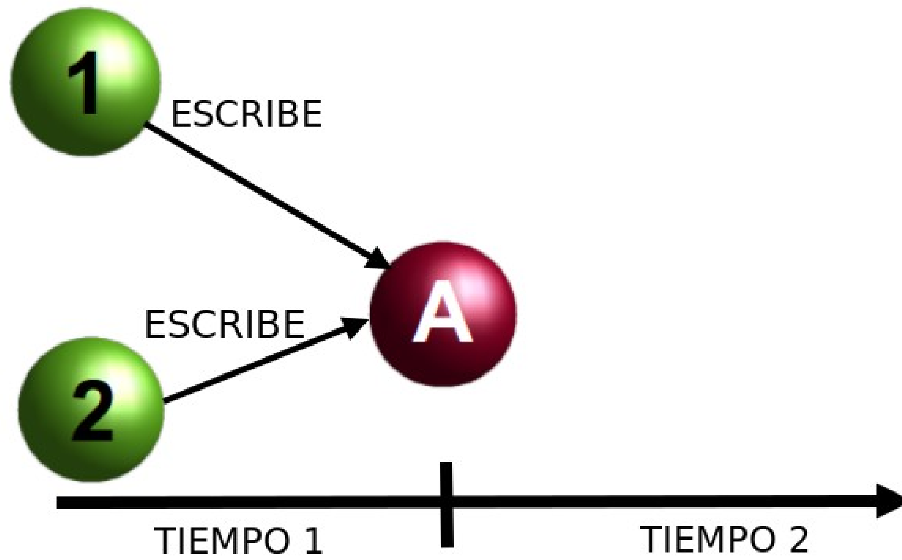


Figura 2.11 Problema al escribir simultáneamente (WAW)

En el caso de que primero se ejecutara el proceso 2,figura 2.12.En tiempos de ejecución posteriores al 2, se tendría el valor que escribió el proceso 1.

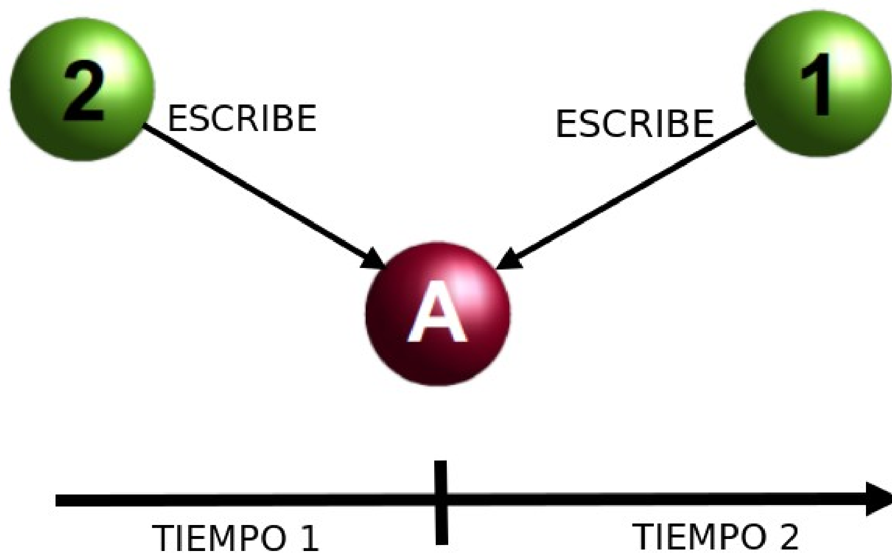


Figura 2.12 Problema al escribir después de escribir (WAW)

Las dependencias de datos se pueden encontrar en diferentes proyectos y procesos, por lo cual es necesario poder identificarlas , para tratar de resolver la dependencia o en su caso ejecutarlas en forma serial.

2.1.2.2) Dependencia de control

Los conflictos o dependencias de control, se derivan de la ejecución de instrucciones de control de flujo: saltos condicionales (if, then, switch) o incondicionales (goto) y las llamadas y retornos de subrutinas o funciones . A estas instrucciones por lo regular se le denomina genéricamente instrucciones de salto.

La instrucción de salto debe determinar si se realiza o no el salto y calcular la dirección de la nueva instrucción.

Las dependencias de control se manifiestan cuando el programa no puede segmentarse por la presencia de una instrucción de salto que no permite predecir de manera fácil el camino que seguirá la ejecución del programa.

La técnica más sencilla que se utiliza es la parada o barrera, que consiste en parar o esperar hasta que la instrucción o dependencia de control se resuelva.

Las dependencias estructurales han sido un punto importante a considerar en el diseño de los procesadores super-escalares, los fabricantes han encontrado técnicas en software y hardware para mitigarlas.

Hablando de software, de acuerdo con Rodríguez Clemente (2000), el compilador es el encargado de reordenar el código de los programas intentando distanciar las instrucciones que tienen dependencias, minimizando o eliminándolas . Sin embargo, muchas veces no es posible realizarlo , por lo cual se introducen instrucciones de NOP.

Respecto a soluciones hardware, Rodríguez Clemente (2000) afirma que lo más usual es ampliar el hardware, replicando unidades funciones, segmentando, ampliando el número de buses y cuando se produce un conflicto estructural, parar la ejecución de la instrucción.

Estas técnicas han funcionado en procesadores como Sparc, PowerPC , MIPS, ALPHA, etc.. ,podemos asumir que algunas técnicas se pueden utilizar para solucionar dependencias estructurales en programas paralelos,sin embargo no todas son útiles o fáciles de implementar.

2.1.3) Concurrencia

Gregory R. Andrews menciona que un programa concurrente es aquel que tiene 2 ó más procesos que cooperan para resolver una tarea. Cada proceso es un programa secuencial que ejecuta una serie de pasos. Los procesos se comunican utilizando variables compartidas o paso de mensajes.

De acuerdo a Gregory R. Andrews cuando se utilizan variables compartidas, un proceso

escribe en una variable y el valor es leído por otro, cuando se utiliza paso de mensajes, un proceso envía un mensaje y es recibido por otro. Algo importante a considerar son las dependencias de datos WAR, WAW y/o RAW.

Otra definición de acuerdo a Ramos Palacios y Rosales Madrigal (2006) los procesos son concurrentes si existen simultáneamente y pueden ser independientes y/o asíncronos. Y mencionan que podemos encontrar dos formas de concurrencia:

- **Implícita**. Por lo regular se encarga de ejecutar concurrencia implícita el sistema operativo y/o el hardware del equipo. Un ejemplo son las operaciones de E/S y la multitarea que se ejecuta no necesariamente en más de un procesador.

- **Explícita.** Definida por el programador o el usuario que ejecutará la aplicación.

El paralelismo es un caso particular de la concurrencia, Ramos Palacios y Rosales Madrigal (2006) describen los siguientes conceptos:

- Programa concurrente. Es aquél que define acciones que pueden realizarse simultáneamente. Puede ser una máquina con un procesador.
- Programa paralelo. Es un programa concurrente diseñado para su ejecución en un hardware paralelo. Puede ser una máquina con 2 o más procesadores que comparten memoria.
- Programa distribuido. Es un programa paralelo diseñado para su ejecución en una red de procesadores autónomos que no comparten memoria. Puede ser un conjunto de máquinas con uno ó más procesadores.

2.1.4) Sincronización

De acuerdo a la definición de programa concurrente por Gregory R. Andrews ,existen una necesidad de sincronización entre los procesos. Para lo cual existen 2 formas de sincronización por exclusión mutua o condicional. La exclusión mutua se utiliza para asegurar que un objeto,variable o recurso sea utilizado por un proceso a la vez. La sincronización condicional se utiliza para asegurar que un proceso se detenga, si es necesario ,hasta que se cumpla una condición.

2.1.5) Regiones condicionales críticas (Barreras)

De acuerdo a Gebali (2011) una barrera es utilizada o definida cuando se necesitan sincronizar diferentes tareas independientes que deben de completarse antes de ejecutar el resto. Las barreras son muy utilizadas en la implementación de algoritmos seriales-paralelos (SPA).
En la figura 2.13 se muestran 3 barreras que sincronizan los diferentes procesos, en caso de que no se colocaran las barreras ,se presentaría un error lógico obteniendo resultados incorrectos y/o impredecibles, lo cual puede resultar en una depuración complicada ya que el compilador por lo general sólo muestra errores de sintaxis y semántica.

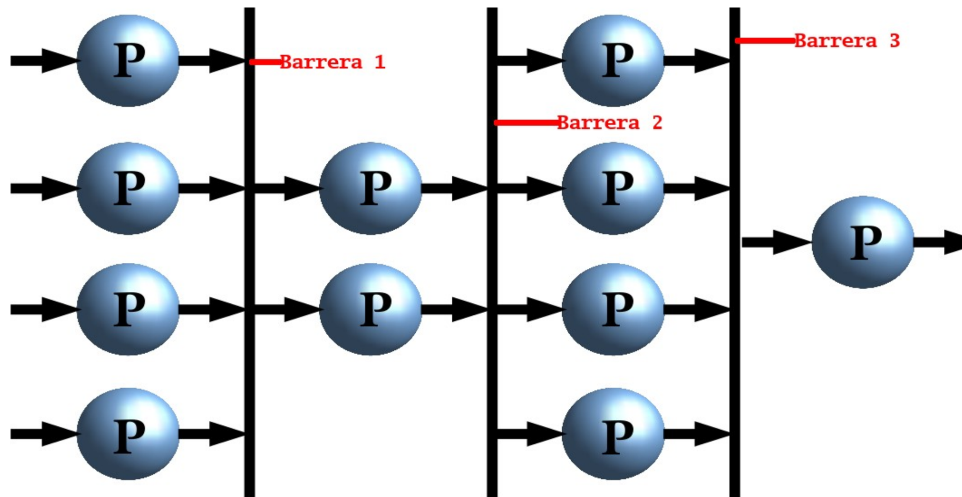


Figura 2.13 Barreras en un algoritmo Serial-Paralelo

2.1.6) Técnicas de paralelización

De acuerdo con Gebali, una forma de realizar una paralelización es utilizar el modelo que se ilustra en la figura 2.14.

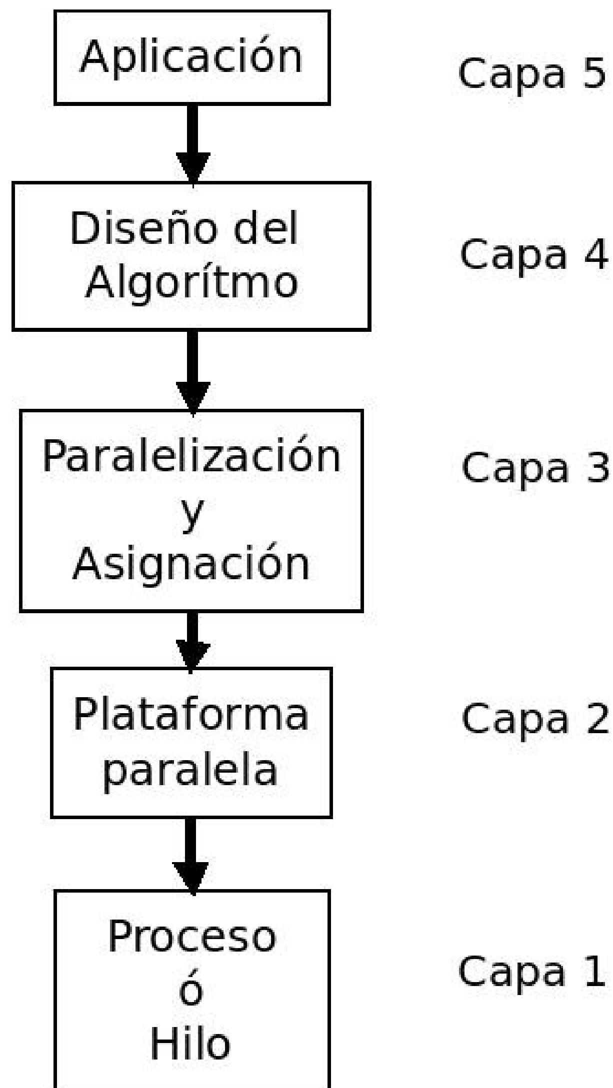


Figura 2.14 Modelo de paralelización

Podemos observar que el proceso cuenta con 5 capas o fases, comenzando por la de aplicación hasta llegar a la capa 1 donde el código se ejecutará mediante procesos o hilos.
En la capa 5 se da lugar a la especificación del problema a resolver o aplicación a desarrollar.

En la capa 4 se desarrolla o aplica algún algoritmo para resolver el problema.

En la capa 3 se paraleliza el algoritmo y en caso de que no se utilice algún balanceador de carga, se procede a asignar cada tarea a un procesador y/o nodo de cómputo.

En la capa 2 si es necesario, se adapta el algoritmo a la plataforma donde se ejecutará el código.

En la capa 1 dependiendo de la plataforma donde se ejecutará el código, se decidirá si será una ejecución por proceso o por hilos.

Una vez que el programador termina de realizar las fases, puede escribir el código teniendo la seguridad de que se podrá ejecutar en la plataforma de destino, de acuerdo a las especificaciones.

En la capa 3 existen técnicas de paralelización ,como ,por nivel de paralelismo, paralelización de datos, paralelización de ciclos, paralelización funcional, gráficas dirigidas de dependencia y matriz de dependencia.

2.1.7) Niveles de paralelismo

Paralelismo a nivel de datos (DLP)
Se dice que existe un nivel de paralelismo de datos cuando se opera simultáneamente sobre múltiples bits de datos o múltiples datos. Por ejemplo al realizar múltiples sumas,multiplicaciones y divisiones de números binarios a nivel de bit.

Paralelismo a nivel de instrucción (ILP)
Se dice que existe un nivel de paralelismo de instrucción cuando el procesador ejecuta más de una instrucción en cada ciclo, un ejemplo es el “pipeline”.

Paralelismo a nivel de hilo (TLP)
Un hilo es una porción de programa que comparte recursos de procesador y memoria con otros hilos. También se les llama procesos de peso ligero.
Se dice que existe un nivel de paralelismo a nivel de hilo cuando varios hilos se ejecutan de manera simultánea en uno o más procesadores.

Paralelismo a nivel de proceso (PLP)
Existe un paralelismo a nivel de proceso,cuando existen diferentes procesos independientes ejecutándose en un sistema de cómputo multitarea con tiempo compartido. Un ejemplo es la ejecución de diferentes procesos en una computadora personal que tiene un sistema operativo multitarea y/o cuenta con múltiples unidades de procesamiento.

2.1.8) Paralelización de datos

La paralelización de datos se utiliza cuando los datos no tienen alguna dependencia entre sí, un ejemplo es el cambio de formato de archivos multimedia. Cada proceso puedo convertir algún pedazo del archivo y al final se unifican utilizando una barrera explícita.

2.1.9) Paralelización de ciclos

Por lo regular la paralelización de ciclos se utiliza en el manejo de arreglos o matrices

distribuidas. En caso de que no exista una dependencia de datos,cada iteración es ejecutada por diferentes procesadores; un ejemplo es la suma de matrices.

2.1.10) Paralelización funcional

Es utilizada cuando un problema se divide en problemas o funciones más sencillas , que no tienen alguna dependencia entre ellas y son ejecutadas en paralelo.

2.1.11) Gráficas dirigidas de dependencia

Un algoritmo se puede representar mediante una gráfica dirigida (GD),figura 2.15.En la cual se puede observar la dependencia entre las diferentes tareas. A dicha gráfica se le conoce como gráfica de dependencia.

Fayez Gebali(2011) propone algunas definiciones respecto a las gráficas de dependencia.

Una gráfica de dependencia consta de una serie de nodos y flechas dirigidas. Los nodos representan las tareas que realiza el algoritmo y las flechas el flujo de datos (Entrada, Salida e Intermedio).

Una gráfica dirigida acíclica (DAG) es una gráfica dirigida que no tiene ciclos.

Una flecha de salida en una gráfica dirigida es aquella en la cual inicia en la salida de un nodo,pero no termina en otro nodo. Representa una salida del algoritmo.

Una flecha intermedia, es aquella que inicia en un nodo y termina en uno o más nodos. Representa alguna variable intermedia.

Una flecha de entrada, es aquella que termina en uno o más nodos, pero no empieza en ningún nodo. Representa las entradas del algoritmo.

Un nodo de entrada es aquel en el cual todas sus entradas,son flechas de entrada.

Un nodo de salida es aquel en el cual todas las salidas son flechas de salida.

Un nodo intermedio, es aquel que tiene al menos una flecha de entrada y al menos una de salida.

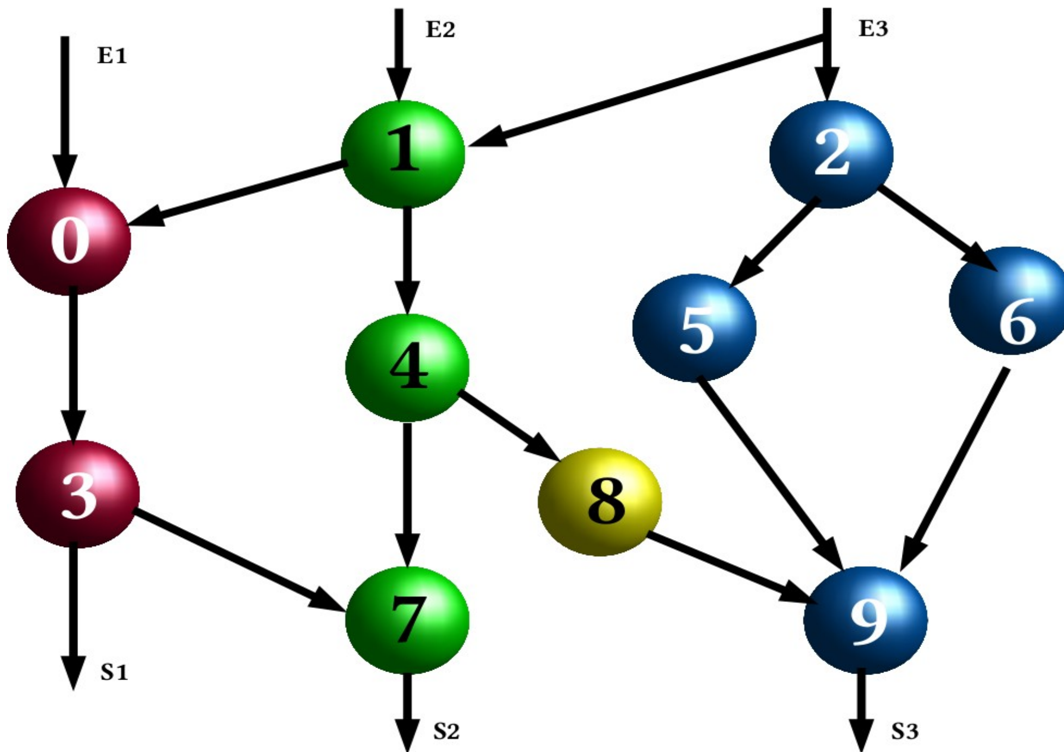


Figura 2.15 Gráfica dirigida de dependencia

2.1.12) Matriz de dependencia

Un algoritmo paralelo también se puede representar como una matriz cuadrada, donde se tienen nodos o tareas.
La matriz tiene las siguientes características:

- a(i,j)=1 indica que el nodo i depende de la salida del nodo j
- a(i,i)=0 indica que el nodo i no depende de su misma salida
- a(i,j) != a(j,i) indica que la matriz es asimétrica y en caso contrario a(i,j)=a(j,i) se tendría un interbloqueo

Tomando como ejemplo el algoritmo representado con la gráfica dirigida de dependencia del tema anterior, se tiene la siguiente matriz de dependencia.

	n0	n1	n2	n3	n4	n5	n6	n7	n8	n9
n0	0	**1**	0	0	0	0	0	0	0	0
n1	0	0	0	0	0	0	0	0	0	0
n2	0	0	0	0	0	0	0	0	0	0
n3	**1**	0	0	0	0	0	0	0	0	0
n4	0	**1**	0	0	0	0	0	0	0	0
n5	0	0	**1**	0	0	0	0	0	0	0
n6	0	0	**1**	0	0	0	0	0	0	0
n7	0	0	0	**1**	**1**	0	0	0	0	0
n8	0	0	0	0	**1**	0	0	0	0	0
n9	0	0	0	0	0	**1**	**1**	0	**1**	0

Figura 2.16 Matriz de dependencia

2.2) Herramientas Actuales

2.2.1)Balanceadores de carga

Un balanceador de carga se encarga de repartir los diferentes trabajos que se están ejecutando en un nodo, hacia los demás, de tal manera que la carga de trabajo sea casi la misma para todos los nodos. Esto con el fin de que no se encuentren nodos sin realizar tareas y se pueda aprovechar todo el hardware con el que se cuente.
Existen diferentes balanceadores de carga en el mercado (por lo regular software propietario), por mencionar algunos PBS, Moab y Mosix.

MOSIX
De acuerdo a la documentación oficial, MOSIX es un sistema de administración para sistemas de cómputo distribuidos, basados en arquitectura x86 y GNU\Linux. MOSIX incorpora un sistema dinámico de descubrimiento de recursos y distribuye las cargas de trabajo de manera proactiva. En un sistema típico con MOSIX, un usuario puede ejecutar múltiples procesos, y automáticamente MOSIX se encargará de distribuir cada carga de una manera eficiente para obtener rendimientos óptimos, todo esto sin cambiar nada a los procesos.

El uso de MOSIX está enfocado en el cómputo distribuido y concurrente, y en el uso intensivo de cómputo. MOSIX no funciona bien con procesos con una carga de datos grande de Entrada/Salida, ni con servicios web.

MOSIX tiene las siguientes características de acuerdo a la documentación oficial:

- Provee una imagen única del sistema

- Descubrimiento de recursos y asignación de trabajos automático
- Comunicación entre los procesos migrados
- Provee puntos de recuperación
- Soporta arquitecturas x86 de 64bits
- Incluye herramientas para su instalación, configuración y monitoreo
- En la última versión no se necesita ningún parche al Kernel Linux

Para instalar y configurar MOSIX , se necesitan los siguientes requerimientos:

- Todos los nodos deben de ser de arquitectura x86
- Todos los núcleos de cada nodo deben de ser de la misma velocidad
- Todos los nodos deben de estar interconectados a través de una red que soporte TCP/IP y UDP/IP
- Los puertos TCP 252 , TCP 253, UDP 249 y UDP 253 deben de estar libres y sin ningún bloqueo
- Para la última versión, el Kernel Linux debe de ser mínimo 3.12
- En caso de utilizar procesadores Intel de última generación con extensiones SSE 4.1 y 4.2, todos los nodos del cluster, deben de tenerlas activadas.

Solamente se menciona la información de MOSIX por ser el programa más usado en los últimos años para el balanceo de carga, tanto de forma comercial como educativa. MOSIX es Software propietario y se necesita pagar para tener una licencia.

Cuando el desarrollo de MOSIX comenzó ,se podía utilizar sin ningún costo. Sin embargo los desarrolladores Amnon Barak y Amnon Shiloh decidieron capitalizar el Software. Lo cual derivó en el extinto proyecto de Software Libre openMOSIX que intentaba recrear el funcionamiento de MOSIX.

2.2.2) Paso de mensajes

El paso de mensajes consiste en que un proceso envía un mensaje a otro, conteniendo información (Poblete Muñoz, 2001).Los mensaje pueden ser enviados de manera síncrona o asíncrona. Durante la historia de la programación paralela, han existido diferentes implementaciones del paso de mensajes, entre ellas Piranha, Sistema P4, PVM , MPI, etc...
Actualmente se podría decir que MPI es el estándar de facto para el paso de mensajes.

De acuerdo a Poblete Muñoz(2001), MPI es un estándar que nació entre noviembre de 1992 y enero de 1994 durante varios encuentros de aproximadamente 40 instituciones, incluyendo fabricantes de máquinas paralelas. Le construcción comenzó en el taller sobre estándares para el intercambio de mensajes en ambientes de memoria distribuida, patrocinado por el centro para la investigación en cómputo paralelo en Williamsbur, Virginia.

Como lo menciona Poblete Muñoz (2001), MPI incluye aspectos importantes como las comunicaciones punto a punto, operaciones colectivas, grupos de procesos, contextos de comunicación, formación de topología y enlaces con Fortran y C.

En el código 2.1 se muestra el programa de “Hola Mundo” en C con MPI , que imprime desde cada procesador una leyenda.

Código 2.1

```
#include <mpi.h>;

int main(int argc, char** argv) {

   MPI_Init(NULL, NULL);

   int world_size;
   MPI_Comm_size(MPI_COMM_WORLD, &world_size);

   int world_rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

   char processor_name[MPI_MAX_PROCESSOR_NAME];
   int name_len;
   MPI_Get_processor_name(processor_name, &name_len);


   printf("Hola Mundo desde el procesador %s, rank %d"
        " de %d procesadores\n",
        processor_name, world_rank, world_size);


   MPI_Finalize();
}
```

2.2.3) Bibliotecas y directivas

De acuerdo a Chapman (2008) OpenMP es una interfaz de programación de memoria compartida, que su prioridad es facilitar la programación paralela en ambientes de memoria compartida. OpenMP no es un lenguaje de programación, es un conjunto de bibliotecas y directivas que pueden ser agregadas a lenguajes secuenciales como Fortran , C , C++, etc..

En el código 2.2 se muestra el programa “Hola Mundo” escrito en C con openMP.

Código 2.2

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]) {
  int iam = 0, np = 1;

  #pragma omp parallel default(shared) private(iam, np)
```

```
  {
    #if defined (_OPENMP)
      np = omp_get_num_threads();
      iam = omp_get_thread_num();
    #endif
    printf("Hola Mundo desde el procesador %d de  %d\n", iam, np);
  }
}
```

2.2.4) Lenguajes de programación

De acuerdo a Chandra (2001) un lenguaje de programación paralela debe soportar 3 aspectos básicos: poder especificar la ejecución paralela, comunicarse entre los diferentes procesos, y tener sincronización.

Algunos lenguajes soportan la programación paralela a través de extensiones, esto tiene una gran ventaja, que es la de conocer la sintaxis y semántica del lenguaje.

En el código 2.3 se muestra el programa "Hola Mundo" escrito en X10, se puede observar la simplicidad del código.

Código 2.3

```
import x10.io.Console;

class HelloWholeWorld {
  public static def main(args:Rail[String]):void {


     finish for (p in Place.places()) {
        at (p) async Console.OUT.println(here+" Hola Mundo “);
     }

  }
}
```

3) Lenguaje de programación X10

3.1) Funcionamiento

Todas las aplicaciones desarrolladas utilizando X10 se necesitan compilar, para esto, existen 2 métodos de compilación uno utilizando JAVA (X10C) y otro utilizando C++ (X10C++).
Se podría decir que la compilación, realiza lo siguiente:

1) Se genera el árbol de sintaxis (AST) optimizado
2) Se transforma el código a Java o C++
3) Se compila utilizando el compilador disponible en el equipo

Se representa a detalle en la siguiente ilustración de Olivier Tardieu. (véase figura 3.1)

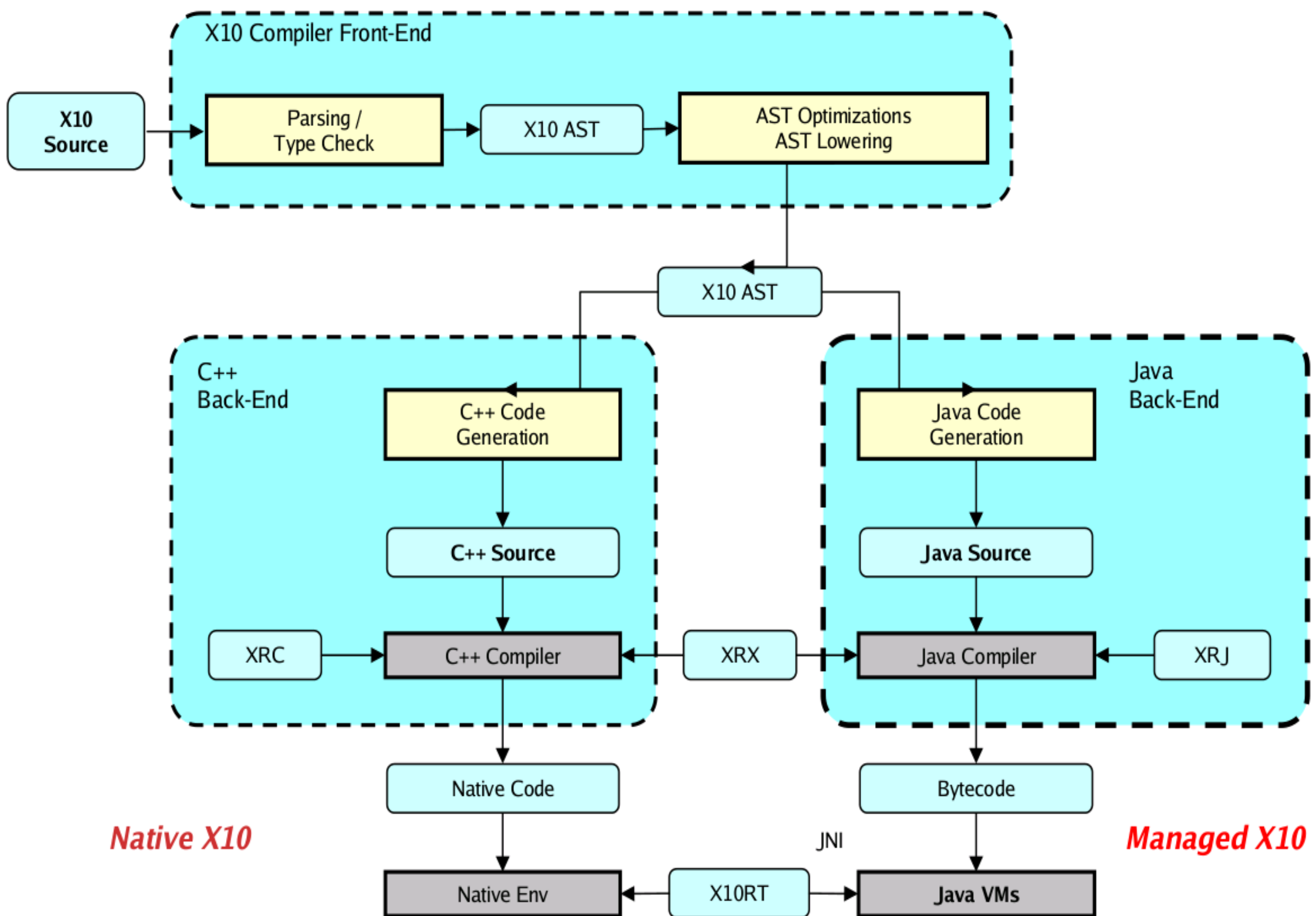


Figura 3.1 Compilación en X10

Debido al proceso de compilación, X10 se puede decir que es independiente de la arquitectura y sistema operativo.

La ejecución de un programa se puede llevar acabo utilizando la máquina virtual de Java o de forma nativa, dependiendo de la compilación.

Si decidimos compilar en forma nativa utilizando el comando X10C++, basta con ejecutar el archivo binario desde la línea de comandos.

```
./nombreprograma
```

Si compilamos utilizando X10C, necesitaremos ejecutar el programa utilizando la máquina virtual.

```
X10 nombreprograma        //Sin extensión del archivo
```

De acuerdo a la Figura 3.2, la ejecución puede estar vinculada con las bibliotecas de MPI, TCP/IP, PAMI, DCMF o CUDA.

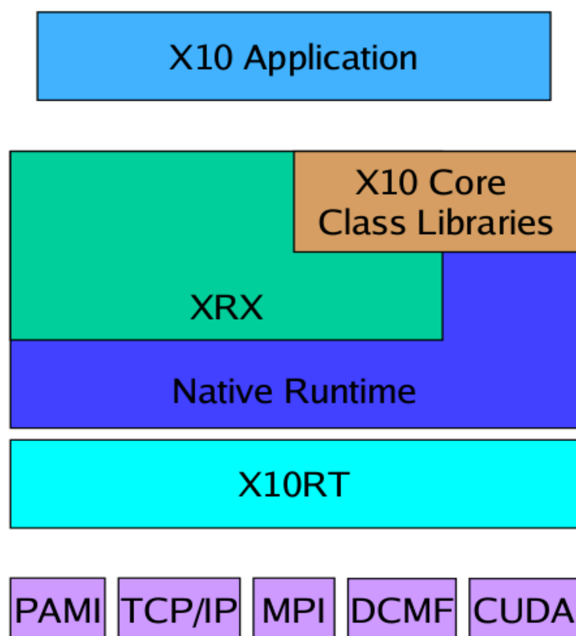


Figura 3.2 Arquitectura de X10

Dependiendo del conocimiento que tengamos de nuestro entorno de ejecución, podemos utilizar las bibliotecas antes mencionadas para aumentar el rendimiento o utilizar todo el poder de cómputo.

X10 nos permite definir 2 variables de entorno que afectan directamente la ejecución:

X10_NPLACES define el número de lugares donde se ejecutará la aplicación, se podría decir que es el número de procesadores en el cual se ejecutará la aplicación.

X10_NTHREADS define el número de hilos de ejecución por cada lugar, por defecto X10 crea sólo un hilo por lugar.

Por ejemplo, si nuestra aplicación va a correr en una máquina con 2 procesadores físicos y

con 8 núcleos (cada procesador cuenta con 4 núcleos), lo ideal sería definir la variables de la siguiente manera:

```
X10_NPLACES=2
X10_NTHREADS=4
```

Sin embargo X10 nos permite definir lugares e hilos sin tener una relación estrecha con el hardware, es decir, podemos tener X10_NPLACES=8 X10_NTHREADS=8 y disponer de una máquina con un procesador y un núcleo ,o tener cientos de núcleos.

Para X10 la interconexión de los diferentes procesadores y núcleos puede ser a través de TCP/IP, MPI, CUDA, DCMF (BlueGene) , PAMI (BlueGene) o de forma local. Lo que permite ejecutar nuestra aplicación en multicomputadoras o multiprocesadores, sin tener que modificar el código fuente o los archivos binarios.

Para instrucciones y detalles de la instalación, en los apéndices se muestra cómo instalar X10 y el entorno de desarrollo en un arquitectura X86 con sistema operativo GNU\Linux distribución JarroNegro 3.0.0.

3.2) Paradigma orientado a objetos

De acuerdo a Javier Ceballos (2003) la programación orientada a objetos (POO) es un modelo que utiliza objetos, ligados mediante mensajes, para la solución de problemas. Los conceptos de la POO tienen origen en los años 60s con el lenguaje Simula 67.

Javier Ceballos (2003) propone que un programa orientado a objetos realiza fundamentalmente tres cosas:

1. Crea los objetos.
2. Se procesa la información al enviar y recibir mensajes a los objetos.
3. Finalmente, cuando los objetos no son necesarios, son borrados, liberándose la memoria utilizando un colector de basura.

Objetos
Javier Ceballos (2003) menciona que un programa orientado a objetos se compone solamente de objetos, entendiendo por objeto una encapsulación genérica de datos y de los métodos para manipularlos.

Mensajes
Normalmente en la ejecución de un programa orientado a objetos, los objetos se encuentran recibiendo, interpretando y respondiendo a mensajes de otros. Los mensajes se encuentran asociados a un método, por lo cual cuando un objeto recibe un mensaje se ejecuta el método asociado.

Métodos
Un método es una porción de código que determina como tiene que actuar un objeto al recibir algún mensaje asociado a dicho método.

Por lo regular los métodos contienen variables locales y pueden llamar a otros métodos del mismo u otro objeto.
Cuando se diseña un objeto la estructura interna se oculta y sólo se dejan visibles los métodos para manipular al objeto.

Clases
Una clase es un tipo de objeto definido por el usuario. Una clase equivale a la generalización o descripción de un tipo específico de objeto.
De acuerdo a Ceballos (2003), cuando se escribe un programa orientado a objetos, no se definen objetos verdaderos, se definen clases de objetos, donde una clase se ve como una plantilla para uno o más objetos.
Muchos autores utilizan el término instancia, siendo una instancia la representación concreta de una clase, desde el punto de vista de Ceballos (2003), instancia y objeto son lo mismo.

Abstracción
De acuerdo a Javier Ceballos (2003) por medio de la abstracción conseguimos no detenernos en los detalles concretos de las cosas que no interesen en cada momento, sino generalizar y centrarse en los aspectos que permitan tener una visión global del problema.

Encapsulamiento
El encapsulamiento permite ver un objeto como una caja negra en la que se ha introducido de alguna manera toda la información relacionada a dicho objeto. Esto permite manipular los objetos como unidades básicas, permaneciendo su estructura interna oculta.

Javier Ceballos (2003) menciona que la abstracción y encapsulamiento están representadas por las clases. Una clase es una abstracción porque en ella se definen las propiedades o atributos de un determinado conjunto de objetos con características comunes, y es una encapsulación porque constituye una caja negra que encierra tanto los datos que almacena cada objeto como los métodos que permiten manipularlos.

Herencia
Javier Ceballos (2003) menciona que la herencia permite el acceso automático a la información contenida en otras clases. De esta forma se reutiliza código. Con la herencia todas las clases están clasificadas en una jerarquía. Cada clase tiene una superclase (La clase superior en la jerarquía),y cada clase puede tener una o más subclases.
Las clases que están en la parte inferior se dice que heredan de las clases superiores.
El término heredar significa que las subclases disponen de todos los métodos y propiedades de su superclase.

Polimorfismo
Esta característica permite implementar múltiples formas de un mismo método dependiendo del mensaje o parámetros utilizados.

Constructores y Destructores
Un constructor es un procedimiento especial de una clase que es llamado automáticamente siempre que se crea un objeto de esa clase.

Por su parte un destructor es un procedimiento que es llamado automáticamente al destruir un objeto de esa clase. Javier Ceballos (2003).

3.3) Tipos de datos

X10 es un lenguaje orientado a objetos fuertemente tipado, es decir cada variable y expresión tienen un tipo que es conocido en el tiempo de compilación y escritura del código fuente. Como en varios lenguajes de programación,cada tipo de datos tiene un rango de valores soportados.
El lenguaje de programación soporta cuatro tipos de valores: objetos, valores de estructura, funciones y “null”.

Los objetos son los tipos de datos más utilizados en X10, al igual que cualquier lenguaje orientado a objetos;estos son instancias de las clases que contienen campos de datos (propiedades) y pueden responder a métodos propios o de una superclase (herencia).

Las estructuras son similares en los objetos,sin embargo utilizan menos espacio y tiempo. Aunque en X10 pueden tener métodos,no tienen un comportamiento inherente. (véase código 3.1)

Código 3.1

```
Struct Posicion{
        public val x:Double;
        public val y:Double;
        public val z:Dobule;

        def this(x:Double, y:Double,z:Double){
        this.x=x; this.y=y; this.z=z;
        }

}
```

Las funciones pueden tener parámetros (argumentos) y un cuerpo en el cual pueden existir expresiones. (véase código 3.2)

Código 3.2

```
def sumar(x1:Double;x2:Double)
{
        return x1+x2;
}
```

“null” es una constante, utilizada como el valor de defecto de las variables.

Palabras reservadas

Aunque son palabras utilizadas por el lenguaje de programación,pueden ser utilizadas como identificadores encerrándolas entre comillas simples ´ ´ .

abstract	as	assert	async	at	athome
ateach	atomic	break	case	catch	class
clocked	continue	def	default	do	else
extends	false	final	finally	finish	for
goto	haszero	here	if	implements	import
in	instanceof	interface	native	new	null
offer	offers	operator	package	private	property
protected	public	return	self	static	struct
super	switch	this	throw	transient	true
try	type	val	var	void	when
while					

3.4) Estructuras de Control

Al igual que la mayoría de lenguajes de programación ,se cuenta con las estructuras *if, if-else*,*switch*, *while*,*do-while*, *for*, *break*,*continue*,*return* ,*throw*,*try-catch* ,*assert*.

En X10 la sentencia if tiene 2 formas, una con la cláusula else o sin ella.
La sentencia if evalúa una expresión que debe ser del tipo Booleana (Verdadera o Falsa).
Si la condición es verdadera se ejecuta el código que contiene el bloque if, contrariamente salta.
En caso de que la sentencia sea del tipo if-else, si la condición es verdadera se ejecuta el código que contiene la estructura if, en caso contrario, se ejecuta el código que contiene el bloque else. (véase código 3.3)

Código 3.3

```
if(i==3)
{
	Console.OUT.println(“Es igual a tres”);
}
else
{
	Console.OUT.println(“No es igual a tres”);
}
```

La sentencia switch evalúa una expresión índice y dependiendo del resultado salta a un caso en el cual el valor es igual al obtenido. Si no existe tal caso, la sentencia salta al caso default (en caso de existir).
Cada caso es evaluado en una secuencia serial descendente. Para prevenir la ejecución de otro caso se utiliza la sentencia break para salir del bloque switch. (véase código 3.4)

Código 3.4

```
switch(i){
	case 1: Console.OUT.println(“uno“);
	case 2: Console.OUT.println (“dos”);
			break;
	case 3: Console.OUT.println(“tres”);
			break;
	default: Console.OUT.println(“Otro valor”);
			break;
}
```

La estructura de control while evalúa una condición Booleana y ejecuta una repetición mientras la condición sea verdadera. En cada ciclo la condición es reevaluada,en caso de ser falsa el ciclo termina. (véase código 3.5)

Código 3.5

```
n=3;
while(n>1)
{
	n=n-1;
	Console.OUT.println(n);
}
```

La estructura do-while es similar a while,sin embargo, ejecuta primero el bloque de repetición y al final evalúa una condición Booleana. (véase código 3.6)

Código 3.6

```
n=3;
do {
	n=n-1;
	Console.OUT.println(n);
}
while (n>1);
```

La estructura de control for , tiene dos formas básicas de funcionamiento.
La primera para iteraciones de instrucciones contenidas en bloque. (véase código 3.7)

Código 3.7

```
for(n=1;n<3;n++)
{
```

```
	Console.OUT.println(n);
}
```

y la segunda para iterar sobre una colección de datos .En el siguiente ejemplo se muestra una función que recibe como argumento una lista tipo Long y regresa el resultado de la sumatoria de todos los elementos de la colección. (véase código 3.8)

Código 3.8

```
static def sum(a:x10.utils.List[Long]):Long{
	var s: Long =0;
	for(x in a) s+=x;
	return s;
}
```

Métodos y funciones pueden regresar valores dependiendo del algoritmo que se esté aplicando; para ello X10 como muchos lenguajes tiene la sentencia return. (véase código 3.9)

Código 3.9

```
static def resta(x:Long,y:Long): Long{
return x-y;
}
```

Los programas en X10 pueden mostrar excepciones para indicar situaciones problemáticas o inusuales, que se manifiestan como terminaciones abruptas. Las excepciones se pueden ejecutar de manera intencional a través de la estructura *throw*.
Cuando una excepción es ejecutada puede ser manejada por la estructura *try-catch*. Si no se cuenta con una declaración *try-catch* el método terminará y lanzará una excepción desde el lugar donde fue llamado.
Dentro de la estructura *try-catch* existe el bloque *finally* que es ejecutado cada vez que es manejada una excepción, sin importar si existe un bloque *catch*. (véase código 3.10)

Código 3.10

```
class Ejemplo{
class excepcion1 extends Exception {}
class excepcion2 extends Exception {}
var finalizo: Boolean =false;

def ejemplo(b:Boolean){
	try {
	throw b ? new excepcion1() : new excepcion2();
	}

catch(excepcion1) {return true;}
catch(excepcion2) {return false;}

	finally {
		this.finalizo=true;
```

```
        }
}

static def ejecutar() {
        val e=new Ejemplo();
        e.ejemplo(true);
        e.ejemplo(false);
}

 public static def main(Rail[String]) {

      ejecutar();

      }


}
```

Existe una estructura llamada assert ,la cual verifica que una expresión sea verdadera y de lo contrario arrojará una excepción x10.lang.Error conteniendo el segundo argumento de la estructura .Es utilizada para verificar expresiones que de antemano se sabe que serán verdaderas y poder detectar errores en la ejecución del programa. (véase código 3.11)

Código 3.11

```
class Ejemploassert {

      public static def main(argv:Rail[String]){

            val a = 2;

            assert a != 2 : "El valor de a es 2";

      }

}
```

También se puede utilizar para detectar errores lógicos. (véase código 3.12)

Código 3.12

```
static def division(x:Double, y:Double){
        val div=x/y;
        assert y != 0.0 : [y]
        return div;
}
```

Algunas veces puede ser importante no ejecutar las estructuras *assert* por ejemplo, si la prueba ocupa mucho tiempo de procesamiento y el código ya fue verificado, se puede compilar el programa con el argumento *-noassert* para que el compilador ignore todas las estructuras *assert*.

3.5) Expresiones

X10 soporta diferentes tipos de expresiones, por lo cual en el documento de especificación se dice que X10 es un lenguaje rico en expresiones.
Al evaluar una expresión se puede obtener un valor, cambiar el valor de una variable o de una estructura de datos, asignar nuevos valores o lanzar una excepción.

Literales
Las literales denotan un valor fijo, X10 permite la siguiente sintaxis para denotarlas.

Tipo	Valores permitidos	Ejemplo
Boolean	True o False	true
Null	Null	self==null
Int	(+ -)(uno o más digitos decimales , octales ó hexadecimales)(n N)	123n Decimal 0123N Octal -0X321N Hexadecimal
Long	(+ -)(uno o más digitos decimales , octales ó hexadecimales)(l L)	1234567890 Decimal -0123456789l Octal 0XBABEl Hexadecimal
UInt	(uno o más digitos decimales , octales ó hexadecimales)(un UN)	123un Decimal 0123un Octal 0xBEAun Hexadecimal
ULong	(uno o más digitos decimales , octales ó hexadecimales)(ul UL)	1234567890ul Decimal 0123456789ul Octal 0XBABEul Hexadecimal
Short	(+ -)(uno o más digitos decimales , octales ó hexadecimales)(s S)	123s Decimal -0123S Octal 0xACs Hexadecimal
UShort	(uno o más digitos decimales , octales ó hexadecimales)(us US)	123us Decimal 0123US Octal 0xACus Hexadecimal
Byte	(+ -)(uno o más digitos decimales , octales ó hexadecimales)(y Y)	50Y Decimal 020y Octal -0xFF Hexadecimal
UByte	(+ -)(uno o más digitos decimales , octales ó hexadecimales)(uy UY)	50Y Decimal 020y Octal 0xFF Hexadecimal

Float	(+ -)(uno o más digitos decimales)(f F)	1f 6,02F 5,52E+23f
Double	(+ -)(uno o más digitos decimales)(d D)	1d 6,02D 12345e-8d
Char	'c' carácter o escape	'a' '\b' retroceso '\t' tabulador '\n' nueva línea '\f' siguiente página '\r' retorno de carro
String	“Cadena de caracteres”	“Hola Mundo!”

operador this
La expresión this is un valor val local que contiene una referencia a una instancia de la clase envolvente. (véase código 3.13)

Código 3.13

```
import x10.io.Console;

class auto{

val  color=0;


         public static def main(args:Rail[String]) {
        val auto=new auto();
        Console.OUT.println("El color del auto es:" + auto.color);
        }


}
```

Variables locales
Una expresión de variable local consiste simplemente en la definición de la variable dentro del objeto actual.

```
val  n=10;
```

Acceso a campos
Una expresión para tener acceso a un campo de un objeto se puede definir como:

- Primary.Campo

- super.Campo
- NombreClase.super.Campo
- this.Campo

Por ejemplo, véase el código 3.14.

Código 3.14

```
import x10.io.Console;

class auto{

val  color=0;
val kilometraje = this.color;

        public static def main(args:Rail[String]) {
       val auto=new auto();
       Console.OUT.println("El color del auto es:" + auto.color);
Console.OUT.println("El kilometraje del auto es:" + auto.kilometraje);
       }


}
```

Llamadas

La sintaxis para la invocaciones de métodos es ambigua, al ejecutar ob.m() puede ser una invocación al método de nombre ***m*** del objeto ***ob*** ó la aplicación de una función localizada en el campo ***ob.m***. Si ambos están definidos en la misma clase, X10 invocará al método. (véase código 3.15)

Código 3.15

```
import x10.io.Console;
class llamada{

static val ejem : () => Long = () => 1;      //función () en campo
static def ejem()=3;                         //Método

   public static def main(args:Rail[String]) {
       val llamada=new llamada();
       Console.OUT.println("El valor es:" + llamada.ejem());

       }

}
```

Asignaciones

La expresión x=e asigna el valor de la expresión e a la variable x.
X10 maneja 2 tipos de variables mutables definidas con la palabra reservada var e

inmutables definidas con la palabra val.

En el siguiente fragmento de código 3.16 se muestra la diferencia entre var y val. En general las variables val solamente pueden ser inicializadas una vez.

Código 3.16

```
var x: Int;
val y: Int;
x=1;
y=2;
y=4;   //Error
```

En X10 existen 3 diferentes asignaciones.

1. x=e;   //asignación a una variable local
2. x.f=e; //asignación a un campo de un objeto
3. x(i, ….. , in) = e;   //asignación de un arreglo u otra estructura

Incremento y Decremento
En X10 existen los operadores ++ y -- , los cuales incrementan o decrementan una variable. La expresión de incremento o decremento puede ser en forma pre o post . (véase código 3.17)

Código 3.17

```
var x:Long;
var y:Long;
x=3;
y=++x;          //x=4 y=4     Forma pre
y=x++;          //x=5 y=4;    Forma post
```

Operaciones numéricas
Los tipos numéricos (Byte, Short, Int , Long, Float, Double, Complex y sus variantes sin signo) son estructuras del lenguaje, y casi todos sus métodos son implementados de forma nativa. Sin embargo, algunas de sus operaciones también pueden ser redefinidas por el usuario.

Concatenación de cadenas
El operador + es utilizado para concatenar cadenas y como operador de adición. Si es utilizado con algún operando que no sea del tipo String, se convertirá utilizando el método toString() de la clase.

Por ejemplo.
“Hola” + 2 + true da como resultado la cadena “Hola2true”

Operaciones lógicas boleanas

X10 soporta los operadores lógicos booleanos de negación (!) , operadores lógicos (& |) y operadores condicionales (&& ||).
La diferencia entre los lógicos y los condicionales, radica en que los lógicos evalúan ambos operandos ; y los condicionales en caso de que el operando del lado izquierdo sea false , ya no se evalúa el siguiente operando.

Operaciones relacionales
El lenguaje soporta las operaciones de relación < , <= , >, >= , == y != .
Sin embargo en X10 se considera igual el 0 y -0 , y también que todos los valores finitos son mayores a menos infinito y menores a infinito.

Conversiones y cast
Las conversiones y cast son utilizadas para forzar el tipo de dato de una expresión. En caso de que no se pueda realizar, se mostrará un error en tiempo de compilación o una excepción (x10.lang.ClassCastException) en tiempo de ejecución.

La diferencia entre una conversión y un cast radica en el tipo de dato origen y destino.
Se dice que es un cast cuando en tiempo de ejecución no se realiza ninguna operación.
Por ejemplo al cambiar un tipo String a Any.

En X10 se dice que es una conversión cuando se toma un valor y se produce otro.
Por ejemplo en el código 3.18, al cambiar el número 1 a tipo Float, se produce una conversión al agregarle la parte decimal.

Código 3.18

```
import x10.io.Console;

public class cast{


     public static def main(args:Rail[String]) {
    val Cadena:String="Hola Mundo";
    val ob : Any = Cadena as Any;          //cast
    Console.OUT.println(ob);


    val flotante: float = 1 as float;      //conversión
    Console.OUT.println(flotante);

    }


}
```

Operador instanceof
Este operador nos permite determinar si un objeto es una instancia de algún tipo soportado por el lenguaje X10.
Tiene la estructura:

Expresión **instanceof** Tipo

Donde *Expresión* puede ser alguna variable o cualquier expresión soportada por X10 y *Tipo* cualquier tipo de datos.

En el siguiente código 3.19 se utiliza el operador *instanceof* para mostrar si la variable *j* es una instancia de la clase Long.

Código 3.19

```
import x10.io.Console;

public class instancia {

 public static def main(args:Rail[String]) {
var r:Long;
var resultado:Boolean;
r=10;
resultado= r instanceof Long;
		Console.OUT.println("La variable j es del tipo Long ? " + resultado);
}

}
```

Paréntesis
Los paréntesis como en otros lenguajes son utilizados para escribir expresiones complejas que no utilicen la precedencia de operadores por defecto, por ejemplo:

1+2*3=7
(1+2)*3=9

Otro uso de los paréntesis es para eliminar ambigüedades, en el siguiente código 3.20 . **ejem.f()** significa "Evaluá el método *f* en el objeto *ejem* " y **(ejem.f)()** significa " Selecciona el campo *f* de *ejem* y evalúalo"

Código 3.20

```
import x10.io.Console;

public class parentesis {

def f()=1;
```

```
val f = () =>2;

 public static def main(args:Rail[String]) {
val ejem=new parentesis();

Console.OUT.println( ejem.f() );        //1
Console.OUT.println( (ejem.f)() );      //2

}

}
```

Rail

X10 incluye una forma corta de construir Rails. Sólo es necesario encerrar las expresiones en corchetes, por ejemplo:

```
val ints <: Rail[Long] = [1,3,5,3,15,5];
```

Los *Rails* se podría decir que son similares a los arreglos de una dimensión, sin embargo utilizan menos recursos. Son utilizados como argumento de la funciones *main* de los ejemplos mostrados en este texto.

3.6) Funciones

Las funciones son bloques de código que se pueden aplicar a un vector de argumentos para producir un valor. Las funciones son tratadas como cualquier código en X10, es decir pueden terminar de manera abrupta, tener excepciones, modificar variables y ejecutarse en diferentes lugares. (véase código 3.2)

3.7) Clases

De acuerdo a la documentación oficial del lenguaje, los objetos son instancias de las clases, la estructura más común y poderosa que maneja X10.

Las clases son estructuradas en un “bosque” de jerarquía de código de herencia simple, cómo en los lenguajes de programación C++ o Java. Sin embargo no existe una clase raíz o padre, de la cual todas las clases sean herederas.

Las clases pueden implementar cualquier número de interfaces, definir e instanciar campos val, instanciar campos var, definir e instanciar métodos, tener constructores, tener propiedades, tener contenedores y definir tipos estáticos.

Los objetos en X10 no tienen bloqueos,por lo cual los programadores deben usar bloques atómicos para la exclusión mutua y relojes para secuenciar operaciones múltiples paralelas.

Los objetos existen en el lugar donde fueron creados. Un lugar no puede utilizar o referirse a

un objeto en un lugar diferente, para ello, existe un tipo especial GlobalRef[T] que permite referencias explicitas a través de los lugares de ejecución.

Las operaciones básicas sobre los objetos son:

- Construcción: Los objetos son creados y sus campos var y val son inicializados
- Acceso a campos: Los campos estáticos, instancias y propiedades pueden ser utilizados a través de llamadas y/o funciones
- Invocación a métodos: Los métodos pueden ser invocados
- Casting y prueba de instancia: Los objetos pueden ser convertidos o examinados para algún tipo
- == y != : Los objetos pueden ser comparados

En el siguiente ejemplo del cálculo del área de un círculo, se muestran algunos conceptos. (véase código 3.21)

Código 3.21

```
1.  import x10.io.Console;
2.
3.  public class clase(radio: Double) {
4.
5.  val     pi=3.1416;      //valor estático
6.  var     area:Double;    //valor dinámico (variable) tipo Long
7.
8.
9.          def this(radio:Double){
10.         property(radio);
11.         }
12.
13.          public static def main(args:Rail[String]) {
14.         val circulo1=new clase(10);
15.         circulo1.area=circulo1.pi*circulo1.radio*circulo1.radio; //el operador ^ es un xor
16.         Console.OUT.println(circulo1.area);
17.         }
18.
19.
20. }
```

Se pueden observar varios detalles en el ejemplo:

- Al definir un campo val (valor) no es necesario definir el tipo de dato.
- En la línea 9 a la 11, se define una propiedad de la clase,que se puede decir es un campo var tipo público. Las propiedades difieren a los campos públicos en que pueden ser referenciadas con self, son definidas en el constructor y son declaradas

en el cabezal de la clase (línea 3).

- En la línea 15, no se utilizó el operador ^ ya que en X10 es un xor bit a bit.
- En la línea 14, se instancia un objeto de la clase circulo con propiedad radio igual a 10.

En el siguiente código 3.22 se ejemplifica la comparación campo a campo de 2 círculos de radio 10, utilizando paralelismo y una barrera.

Código 3.22

```
1.  import x10.io.Console;
2.
3.  public class comparar {
4.
5.
6.           public static def main(args:Rail[String]) {
7.
8.          val circulo1=new circulo(10);
9.          val circulo2=new circulo(10);
10.
11.         finish{
12.                 async{  circulo1.calcula_area(); }
13.                 async{  circulo1.calcula_perimetro(); }
14.         }
15.
16.         if(circulo1.radio==circulo2.radio && circulo1.area==circulo2.area &&
    circulo1.perimetro==circulo2.perimetro)
17.         Console.OUT.println("Los círculos son iguales");
18.         else
19.         Console.OUT.println("Los círculos son diferentes");
20.
21.
22.         }
23.
24. }
25.
26.
27. class circulo(radio:Double)
28. {
29. var     area:Double;
30. var     perimetro:Double;
31. val     pi=3.1416;
32.
```

```
33.        def this(radio:Double){
34.        property(radio);
35.        }
36.
37.        public def calcula_area()
38.        {
39.        area=pi*radio*radio;
40.        }
41.
42.        public def calcula_perimetro()
43.        {
44.         perimetro=pi*radio*2;
45.        }
46.
47. }
```

De la línea 27 a la 47, se define una clase con el nombre circulo que contiene 2 campos variables (**var**) tipo **Double** y un valor estático llamado **pi**; también se definen 2 métodos, **calcula_area** y **calcula_perimetro** .
En la línea 8 y 9, se instancia 2 objetos con la propiedad radio igual a 10.
En la línea 16 a la 19 se comparan los 2 objetos campo a campo.
En la línea 12 y 13 se ejecutan los métodos en forma concurrente y asíncrona para el primer círculo.
Con el bloque **finish{}** se define una barrera que espera a que terminen las tareas contenidas en el.
Al ejecutar el programa el resultado es que los 2 círculos son diferentes, ya que en el segundo no se calculó el área ni el perímetro.

En este ejemplo podemos observar que de no poner la barrera con **finish**, no se calculará a tiempo el área ni perímetro del primer círculo, mostrando un resultado no esperado; esto es debido a que termina el proceso padre antes que los 2 procesos creados con **async**.

3.8) Estructuras

Aunque X10 es un lenguaje orientado a objetos, existen casos en los cuales no se puede pagar el costo de tiempo y espacio de memoria de la implementación de un objeto. Por lo cual X10 provee estructuras que son objetos reducidos, los cuales no tienen herencia o campos mutables por lo que los métodos se implementan directamente sin ninguna búsqueda.

Las estructuras y clases son interoperables, ambas pueden implementar interfaces. Las subrutinas que tienen como argumentos interfaces pueden tomar estructuras y clases.

En algunos casos las estructuras se pueden convertir en clases o viceversa. Si se empieza a declarar una estructura y se decide que se necesitaba una clase, bastará cambiar la palabra **struct** por **class**. Por lo cual si se tiene una clase que no utiliza herencia o campos mutables, se pueden cambiar la palabra **class** por **struct** .

El siguiente código 3.23 utiliza una estructura para definir números complejos, la estructura cuenta con un método para imprimir su valor y en la línea 34 a 37 se define un operador público **+** para la suma de números complejos.

Código 3.23

```
1.  import x10.io.Console;
2.
3.  class estructura{
4.
5.          public static def main(args:Rail[String]):void {
6.
7.                  Console.OUT.print("La suma de ");
8.                  val x:complejo=complejo(3,4);
9.                  x.imprimir();
10.
11.                 val y:complejo=complejo(1,2);
12.                 Console.OUT.print(" y ");
13.                 y.imprimir();
14.
15.                 Console.OUT.print(" es ");
16.                 val resultado:complejo=x+y;
17.                 resultado.imprimir();
18.                 Console.OUT.println(" ");
19.
20.         }
21.
22.
23. }
24.
25.
26. struct complejo(re:Double,im:Double)
27. {
28.
29.         public def imprimir()
30.         {
31.                 Console.OUT.print("(" + re + " + " + im +"i)");
32.         }
33.
34.         public operator this + (that:complejo):complejo
35.         {
36.                 return complejo(re + that.re, im + that.im);
37.         }
```

```
38.
39.
40.
41.}
```

3.9) Lugares

De acuerdo a la documentación de X10, un lugar es un repositorio para datos y actividades, que puede corresponder a un proceso , a un procesador o a un nodo de cómputo.
Las actividades ejecutándose en un lugar, pueden acceder a datos locales o remotos.

La asignación de los lugares depende del programador y/o del entorno de ejecución.
El número de lugares es determinado al comienzo de la ejecución del programa y permanece constante. Para ello se puede definir la variable de ambiente *X10_PLACES .*

Los lugares en X10 son instancias de x10.lang.Place . La clase Place provee diferentes métodos como Place.places el cual es una secuencia de los lugares disponibles para la ejecución del programa.

De acuerdo a Olivier Tardieu los lugares se pueden representar como en la figura 3.3.

En la figura

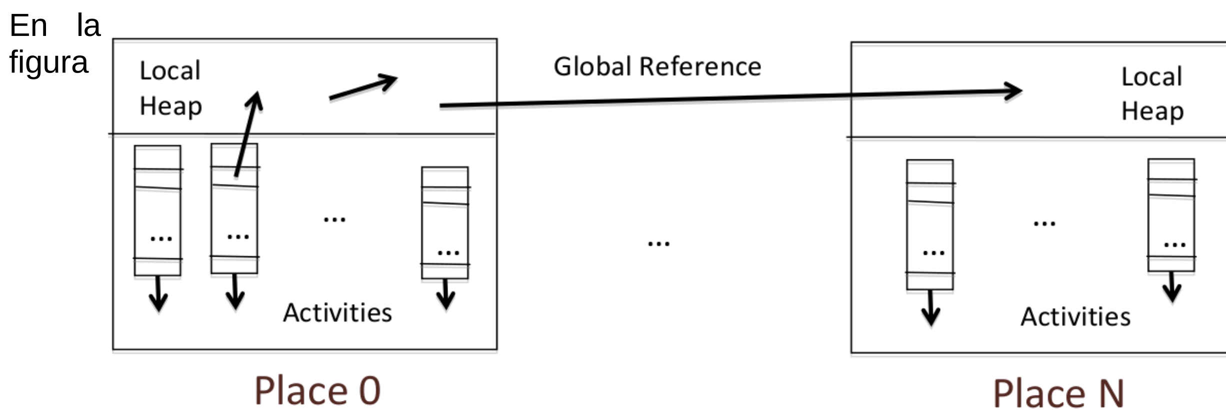


Figura 3.3 Lugares en X10

también podemos observar las actividades, las cuales se detallan a continuación.

3.10) Actividades

Una actividad es una serie de sentencias que se ejecutan independientemente con sus propias variables locales y pila, se podría decir que es un “hilo ligero”.

Un programa en X10 puede tener muchas actividades ejecutándose concurrentemente. Todos los códigos de X10 corren sobre una actividad padre, cuando el programa es iniciado, el método *main* es invocado en la actividad llamada raíz.

Una actividad puede estar en estado de ejecución, bloqueado o terminado.
Una actividad se encuentra en estado terminado cuando ya no existen sentencias a ejecutar, y puede terminar normalmente o abruptamente.
Las actividades pueden ejecutarse por mucho tiempo y tener acceso a muchos datos, sobre todo si llaman a métodos recursivos.
Una actividad puede lanzar nuevas actividades de forma asíncrona o en paralelo. Todas las actividades son “lanzadas” por otras, excepto por la actividad raíz . Por lo cual las actividades las podemos representar en forma de árbol.

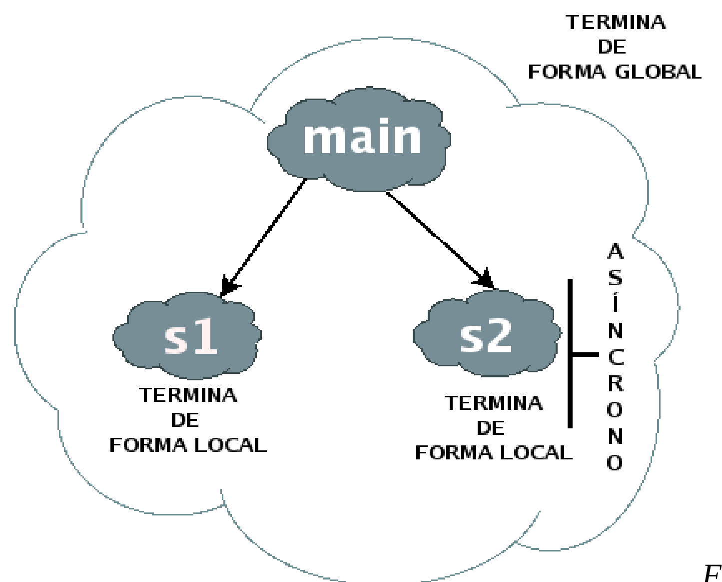
X10 maneja 2 tipos de terminación de actividades, de forma local y global.
Una actividad termina de forma local cuando ya no existen instrucciones por ejecutar y termina de forma global cuando todas las actividades creadas a partir de ella terminan.

Por ejemplo si se crean 2 actividades s1 y s2 .

```
async{ s1();}
async{ s2();}
```

La actividad raíz (main) termina de forma global cuando terminen s1 y s2 de forma local.

Si vemos el árbol de actividades creado, se vería como la figura 3.4.



F
igura 3.4 Actividades en X10

3.11) Relojes

Muchos algoritmos paralelos se ejecutan en fases , y se necesitan sincronizar las actividades. X10 proporciona un mecanismo llamado relojes que son un tipo de barreras. Los relojes están diseñados para ser dinámicos, es decir, nuevas actividades pueden registrarse a relojes ya definidos y las actividades terminadas puede des-asociarse de los relojes.

En el siguiente ejemplo (véase código 3.24), se tienen 2 actividades A y B, y 3 fases. Se necesita que se muestre en la primera fase el texto “A-1” y “B-1” , en la segunda “A-2” y “B-2” y por último “A-3” y “B-3”. El programa utiliza un reloj llamado c1 para la ejecución por fases.

Al momento de terminar cada fase, se ejecuta el método advanceAll(); para continuar a la segunda fase. El método advanceAll() causa que todos las actividades esperen a que terminen todas las involucradas.

Código 3.24

```
import x10.io.Console;

class relojes{

static def decir(s:String) {
Console.OUT.println(s);
}

public static def main(argv:Rail[String]){

    finish async{
    val c1=Clock.make();

    async clocked(c1){
        decir("A-1");
        Clock.advanceAll();
        decir("A-2");
        Clock.advanceAll();
        decir("A-3");
    }

   async clocked(c1){
       decir("B-1");
        Clock.advanceAll();
        decir("B-2");
        Clock.advanceAll();
        decir("B-3");
    }

  }

}
}
```

Al utilizar relojes o barreras, puede ser que el programa sufra un deadlock (cerradura o punto muerto) , sin embargo se recomienda avanzar todas las tareas con el método advanceAll() ,y no utilizar advance() el cual en caso de que una tarea este asociada a 2 o más relojes y se utilice dicho método solamente se avanzará un reloj y se producirá una cerradura o punto

muerto.

Otra recomendación de los desarrolladores de X10 es que al utilizar

```
finish async clocked(c){
…..
}
```

el reloj *c* se encuentre en el mismo alcance es decir

```
val c:Clock = Clock.make();
finish async clocked(c){
…..
}
```

Y no definir código anidado como el siguiente, el cual nunca terminará.

```
val c:Clock= Clock.make();

async clocked(c) {

        finish async clocked(c){
        …..
        }

        c.advanceAll();

}
```

3.12) Arreglos locales y distribuidos

X10 proporciona un paquete de clases x10.array para la definición y utilización de arreglos locales y distribuidos de múltiples dimensiones. Los arreglos generados resultan equivalentes en cuanto a rendimiento a los utilizados en lenguajes como C o Fortran. Los arreglos son una parte muy importante en la programación paralela, se podría decir que la mayoría de los programas utilizan al menos un arreglo.

La clase array provee subclases para arreglos multidimensionales por ejemplo para un arreglo de una dimensión se utiliza la subclase Array_1, para 2 dimensiones Array_2, para 3 dimensiones Array_3,y Array_4 para 4 dimensiones. En caso de requerir más dimensiones , el arreglo se puede construir con la clase Array.

Por ejemplo para crear un arreglo llamado a, de 2 dimensiones de 10x10 elementos del tipo Long se utiliza la siguiente expresión

```
val a=new Array_2[Long](10,10);
```

X10 nos proporciona varias maneras de manipular los elementos de un arreglo de acuerdo al gusto o costumbre del programador, sin embargo al momento de compilar el código generado será idéntico.

A continuación se muestra la suma de todos los elementos de un arreglo definido como "a" de 2 dimensiones, en 3 diferentes formas, al final la variable sum contendrá el mismo resultado. (véase código 3.25)

Código 3.25

```
for(var i:long=0; i<a.numElems_1;i++){
        for(var j:long=0; j<a.numElems_2;j++){
                sum+=a(i,j);
        }
}
```

```
for([i,j] in a.indices()){
sum += a(i,j);
}
```

```
for(v in a){
        sum += v;
}
```

De acuerdo a la experiencia y gusto del programador, podrá elegir cualquiera de las 3 formas. Siendo la primer forma la más conocida, ya que es utilizada en múltiples lenguajes estructurados como C.

La clase DistArray y sus subclases son una extensión de la clase Array para distribuir los elementos de los arreglos en diferentes lugares. Al momento de escribir este trabajo, los desarrolladores del lenguaje aún trabajan en implementación; sin embargo existen 3 subclases que se pueden utilizar.

DistArray_Unique que distribuye un elemento por cada lugar, DistArray_Block_1 que distribuye los elementos de un arreglo de una dimensión a un grupo de lugares, cada lugar tendrá un arreglo de una dimensión.Finalmente DistArray_Block_2 que distribuye los elementos de un arreglo de 2 dimensiones a un grupo de lugares, cada lugar tendrá un arreglo de 2 dimensiones.

Los datos de un arreglo local se pueden migrar a otro nodo automáticamente, al realizar alguna operación con async. Sin embargo al hacerlo manual utilizando un arreglo distribuido se acorta el tiempo de migración hacia el nodo.

En el siguiente ejemplo se crea un arreglo distribuido de una dimensión con 10 elementos, cada elemento es iniciado a la multiplicación de su índice por 2.Se utiliza DistArray_Block_1 para que cada lugar tenga un arreglo de una dimensión. (véase código 3.26)
Se itera por cada lugar disponible y se obtienen los índices de los elementos que contiene dicho lugar, e imprime los elementos. Se definen 3 variables locales al lugar: *contador_local*, *i* e *indice*. En caso de definir las variables globales, los procesos modifican la misma variable por lo cual ocurre una condición de ejecución anormal.

Código 3.26

```
import x10.io.Console;
import x10.array.*;

public class distribuidos {


 public static def main(args:Rail[String]) {

val distribuido =new DistArray_Block_1[Long](10, (i:Long)=>i*2 as long);


finish for (p in Place.places()) {
	at (p) async
	{

		val contador_local= distribuido.localIndices().iterator();
		var i:Long;
		var indice:Point;


		while(contador_local.hasNext())
		{
		indice=contador_local.next();
		i=distribuido(indice);
	     Console.OUT.println( "distribuido(" + indice + ")=" + i + " se encuentra en el lugar: " +here  );

		}

	}
 }


}

}
```

A continuación se muestra la ejecución del programa con 4 lugares. Se puede observar por

no mostrar en orden los índices, que no existe alguna dependencia de datos entre los arreglos y que se ejecuta de manera paralela. (véase figura 3.5)

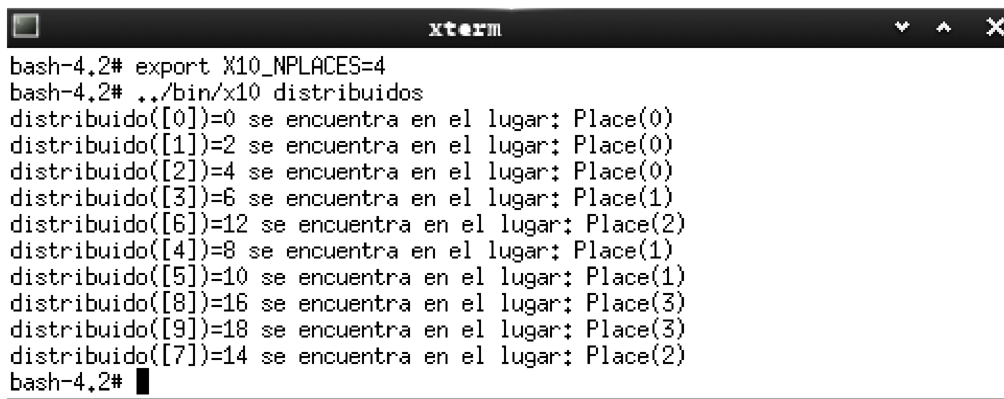


Figura 3.5 Ejecución en 4 lugares

4) Ejemplos prácticos

En las siguientes páginas se muestran algoritmos y códigos fuente de los ejemplos más utilizados en la enseñanza de la programación paralela. Debido a su gran uso existen muchas implementaciones y algoritmos, sin embargo los utilizados en la elaboración de este trabajo, son sencillos y fáciles de implementar para mostrar las bondades del lenguaje X10.

Los resultados fueron obtenidos utilizando cuatro computadoras con diferentes distribuciones GNU\Linux , cuyas características se describen en la siguiente tabla.

Procesador	Velocidad	Memoria	#Procesadores	#Núcleos físicos	#Hilos por núcleo	Núcleos detectados por el S.O.
AMD Athlon x2 270	3.4Ghz	4GB	1	2	1	2
Intel Atom N270	1.6Ghz	2GB	1	1	2	2
Intel Xeon E5620	2.40Ghz	12GB	1	4	1	4
Intel Xeon	3.73Ghz	20GB	2	2	2	8

Aunque se utilizaron computadoras SMP, los ejemplos se pudieron haber ejecutado en un conjunto de múltiples computadoras (Cluster) interconectadas con TCP/IP, MPI o PAMI. Solamente es necesario que compartan autenticación por llaves mediante SSH , que los archivos de Java y X10 se encuentren en los mismos directorios, definir las variables X10_HOSTLIST o X10_HOSTFILE y definir la variable X10_NPLACES igual al número de nodos del Cluster.

La aceleración (Speedup) se calculó de acuerdo a la siguiente fórmula.

$$Speedup = \frac{TiempoSerial}{TiempoParalelo}$$

Donde el tiempo paralelo cambia de acuerdo al número de hilos y/o número de lugares de ejecución.

En la mayoría de los ejemplos se puede observar que el procesador Intel Xeon E5620 a 2.40Ghz tiene el mejor desempeño, esto es debido a que cuenta con 4 núcleos físicos sin tecnología HyperThreading en un solo socket.
Al igual el procesador AMD Athlon x2 270 tiene 2 núcleos físicos con lo cual lo pone en segundo lugar de desempeño. El Intel Xeon a 3.73Ghz, aunque tiene más núcleos no se obtiene el rendimiento esperado por el tiempo de migración entre los 2 sockets , 2 núcleos y a su vez la migración usando la tecnología HyperThreading.
Por último, el Intel Atom N270 es el más lento.Aunque tiene 2 hilos de ejecución no se mostró una mejoría en general durante la ejecución de los ejemplos.

En todos los ejemplos se puede observar que el uso de múltiples lugares en X10 no tienen buen rendimiento utilizándolo en una sola máquina, por lo cual es recomendable sólo definirlos cuando se utilicen múltiples nodos de cómputo.

Cálculo de la secuencia Fibonacci

La sucesión fue descrita por Leonardo de Pisa (Fibonacci) como la solución a un problema de la cría de conejos: Cierto hombre tenía una pareja de conejos en un lugar cerrado y deseaba saber cuántos se podrían reproducir en un año a partir de la pareja inicial teniendo en cuenta que de forma natural tienen una pareja en un mes, y que a partir del segundo se empiezan a reproducir. Se puede ilustrar con la siguiente tabla.

Mes	Explicación de la genealogía	Parejas de conejos
Comienzo del mes 1	Nace una pareja de conejos (pareja A).	1 pareja en total.
Fin del mes 1	La pareja A tiene un mes de edad. Se cruza la pareja A.	1+0=1 pareja en total.
Fin del mes 2	La pareja A da a luz a la pareja B. Se vuelve a cruzar la pareja A.	1+1=2 parejas en total.
Fin del mes 3	La pareja A da a luz a la pareja C. La pareja B cumple 1 mes. Se cruzan las parejas A y B.	2+1=3 parejas en total.
Fin del mes 4	Las parejas A y B dan a luz a D y E. La pareja C cumple 1 mes. Se cruzan las parejas A, B y C.	3+2=5 parejas en total.
Fin del mes 5	A, B y C dan a luz a F, G y H. D y E cumplen un mes. Se cruzan A, B, C, D y E.	5+3=8 parejas en total.
Fin del mes 6	A, B, C, D y E dan a luz a I, J, K, L y M. F, G y H cumplen un mes. Se cruzan A, B, C, D, E, F, G y H.	8+5=13 parejas en total.
...	...	...
	...	...

Tiene numerosas aplicaciones en ciencias de la computación, matemáticas y teoría de juegos. También aparece en configuraciones biológicas, como por ejemplo en las ramas de los árboles, en la disposición de las hojas en el tallo, en la flora de la alcachofa, las inflorescencias del brécol romanesco y en el arreglo de un cono.

Para obtener la sucesión se puede realizar de forma iterativa o con recursividad de acuerdo a los siguientes algoritmos.

Iterativa

```
fib(n)
{
	i=1
	j=0
para k desde 0 hasta n-1 hacer
```

```
        t=i+j
        i=j
        j=t
imprime j
}
```

En el siguiente código 4.1 se muestra una implementación en X10 utilizando paralelismo.

Código 4.1

```
import x10.io.Console;

public class fib{

 public static def main(args:Rail[String]) {

val n = (args.size > 0) ? Long.parse(args(0)) : 10;
Console.OUT.println("Calculando fib("+n+")");

var x:Long=0;
var y:Long=1;
var z:Long=0;
var i:Long;


     finish{
     async{
     for(i=1;i<n;i++)
     {
          z=x+y;
          Console.OUT.println(z);
          x=y;
          y=z;
     }

     }

     }

}

}
```

El algoritmo presenta una dependencia de datos, al ser una sucesión. Debido a esto el rendimiento puede ser el no esperado.

Resultados del cálculo de la posición 38

	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
1 Hilo	0.379	2.092	0.318	0.552
2 Hilos	0.380	2.095	0.312	0.554
3 Hilos	0.378	2.102	0.310	0.555
4 Hilos	0.382	2.103	0.313	0.558
5 Hilos	0.370	2.111	0.312	0.556
6 Hilos	0.399	2.106	0.315	0.569
7 Hilos	0.357	2.125	0.316	0.562
8 Hilos	0.392	2.107	0.317	0.563

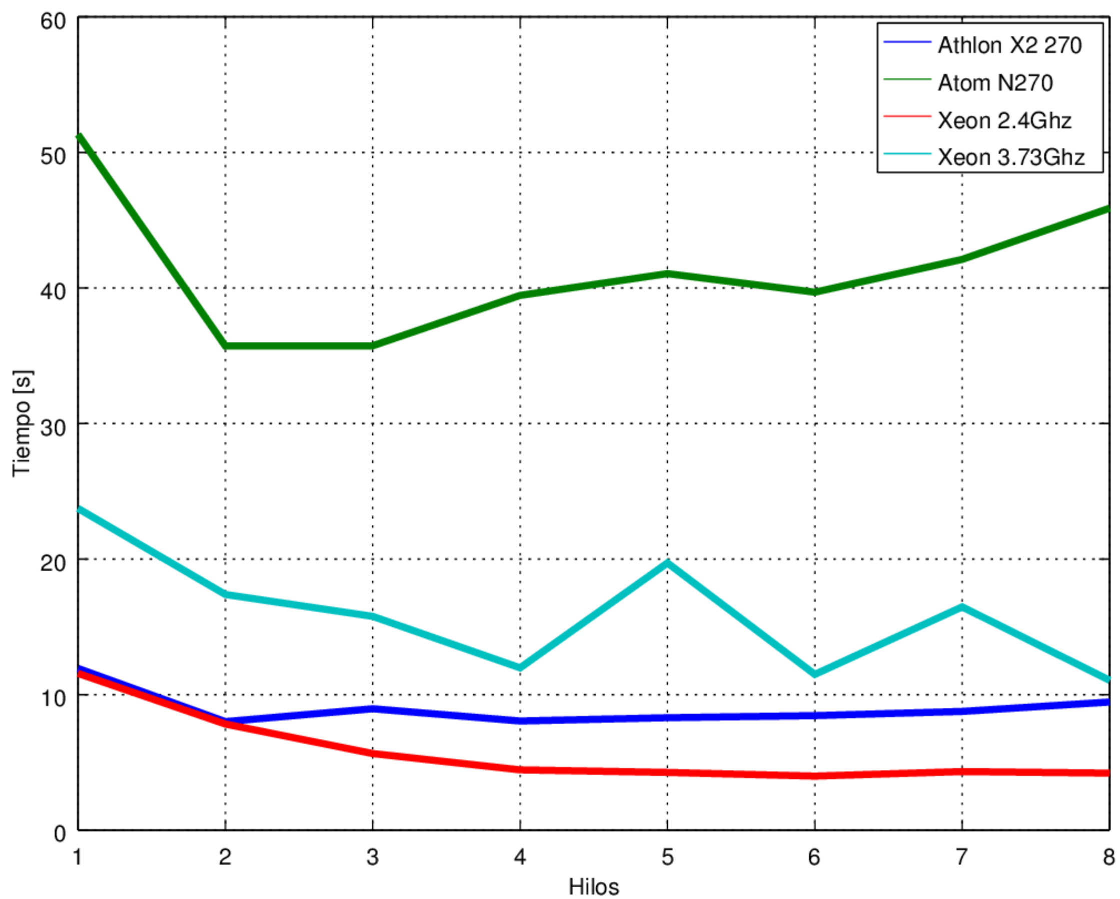


Figura 4.1 Gráfica del tiempo de ejecución del cálculo de Fibonacci en múltiples hilos

	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
1 Hilo	1.000	1.000	1.000	1.000
2 Hilos	0.99737	0.99857	1.0192	0.99639
3 Hilos	1.00265	0.99524	1.0258	0.99459
4 Hilos	0.99215	0.99477	1.0160	0.98925
5 Hilos	1.02432	0.99100	1.0192	0.99281
6 Hilos	0.9487	0.99335	1.0095	0.97012
7 Hilos	1.06162	0.98447	1.0063	0.98221
8 Hilos	0.96684	0.99288	1.0032	0.98046

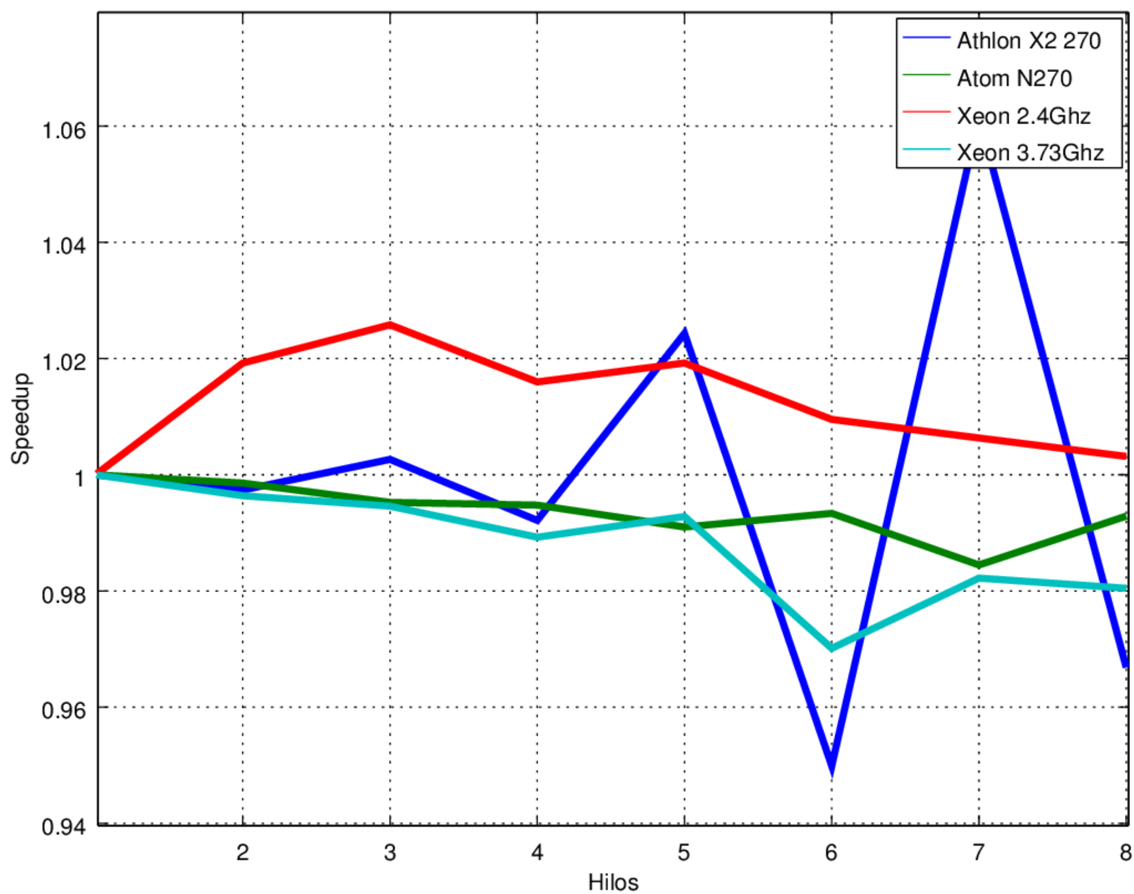


Figura 4.2 Gráfica del Speedup del cálculo de Fibonacci en múltiples hilos

	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
1 Lugar	0.379	2.092	0.318	0.552
2 Lugares	0.694	5.518	0.641	1.001
3 Lugares	1.004	6.370	0.577	0.972
4 Lugares	1.281	7.498	0.624	1.026
5 Lugares	1.380	8.145	0.692	1.226
6 Lugares	1.669	8.752	0.873	1.498
7 Lugares	1.849	9.724	1.035	1.587
8 Lugares	2.149	11.165	1.019	1.584

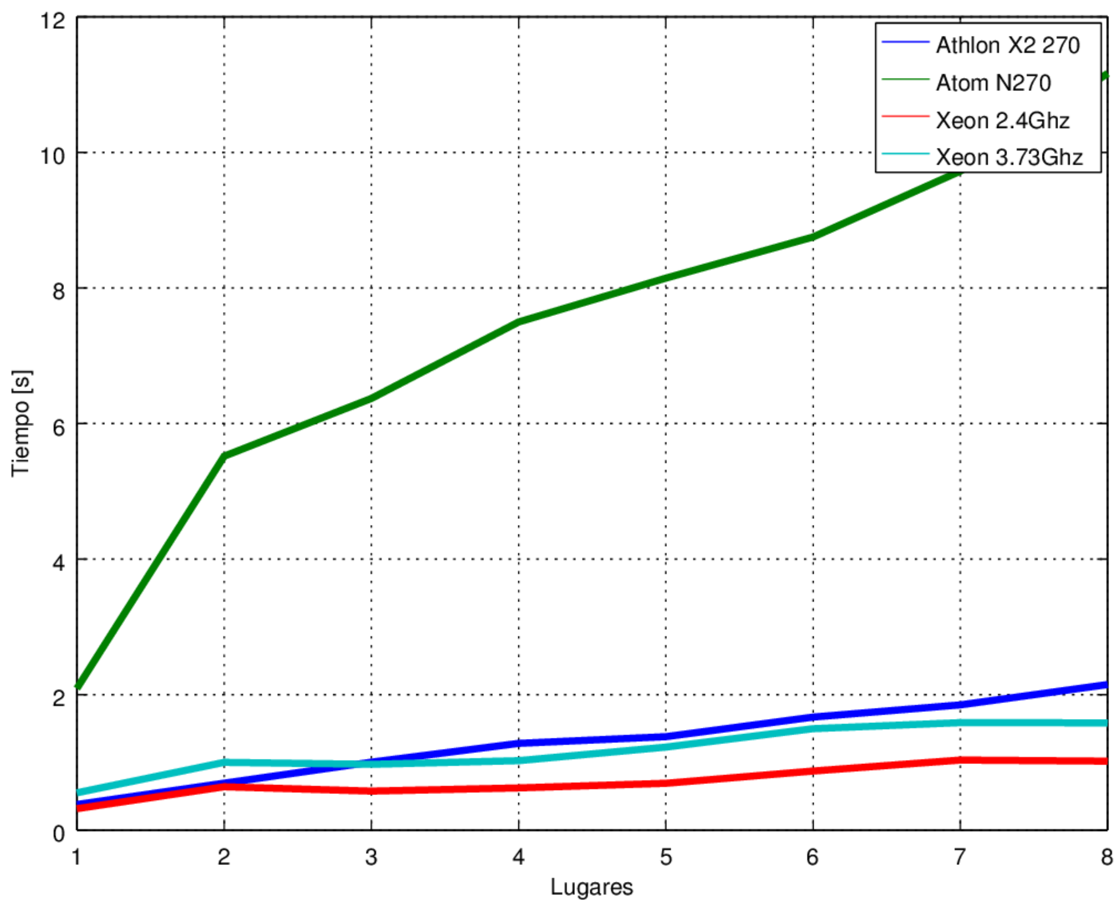


Figura 4.3 Gráfica del tiempo de ejecución del cálculo de Fibonacci en múltiples lugares

	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
1 Lugar	1.000	1.000	1.000	1.000
2 Lugares	0.54611	0.37912	0.49610	0.55145
3 Lugares	0.37749	0.32841	0.55113	0.56790
4 Lugares	0.29586	0.27901	0.50962	0.53801
5 Lugares	0.27464	0.25684	0.45954	0.45024
6 Lugares	0.22708	0.23903	0.36426	0.36849
7 Lugares	0.20498	0.21514	0.30725	0.34783
8 Lugares	0.17636	0.18737	0.31207	0.34848

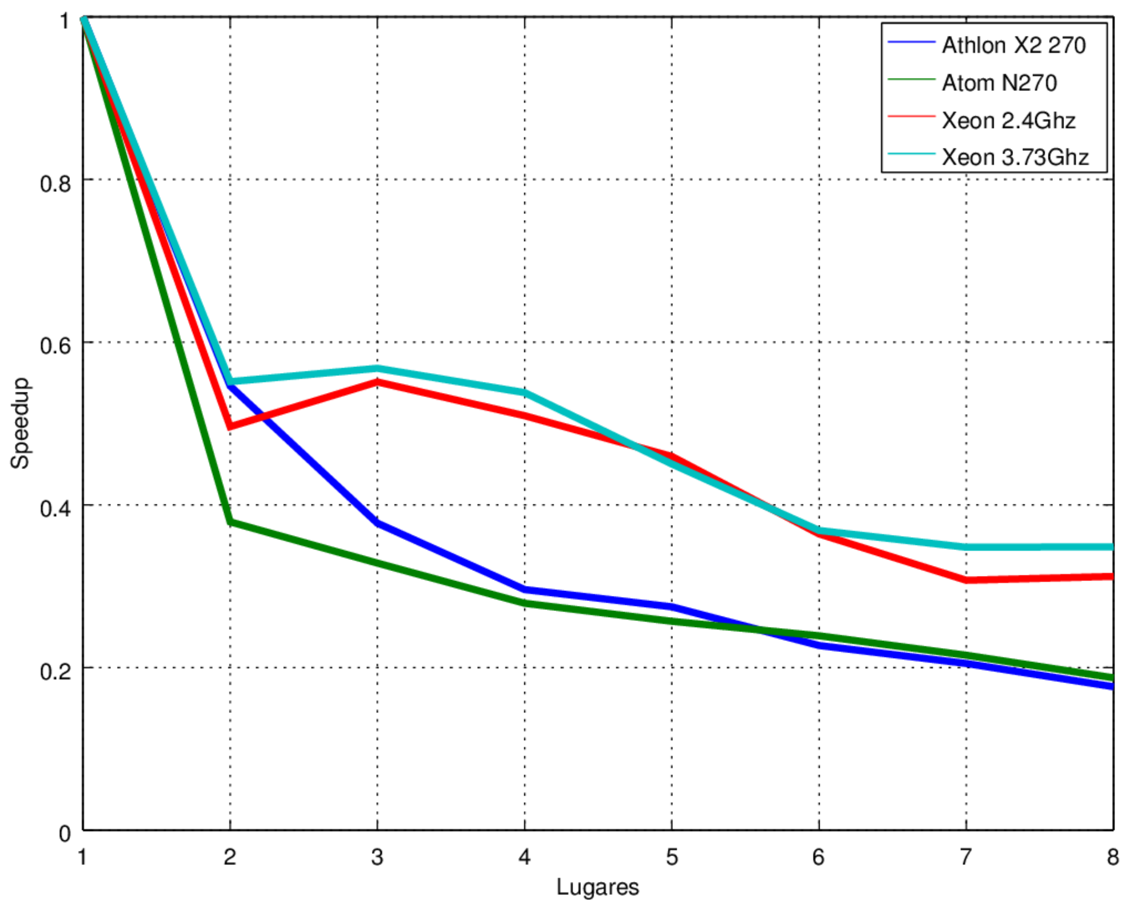


Figura 4.4 Gráfica del Speedup del cálculo de Fibonacci en múltiples lugares

Muestras	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
(1) 1 Lugar y 1 hilo	0.379	2.092	0.318	0.552
(2) 1 Lugar y 2 hilos	0.385	2.120	0.309	0.558
(3) 2 Lugares y 1 hilo	0.682	5.253	0.547	0.923
(4) 2 Lugares y 2 hilos	0.688	5.106	0.550	0.922
(5) 2 Lugares y 3 hilos	0.711	5.194	0.549	0.923
(6) 2 Lugares y 4 hilos	0.715	4.346	0.549	0.931
(7) 3 Lugares y 1 hilo	0.894	6.292	0.575	0.969
(8) 3 Lugares y 2 hilos	0.913	5.847	0.576	0.964
(9) 3 Lugares y 3 hilos	0.973	5.794	0.571	0.978
(10) 4 Lugares y 1 hilo	1.109	6.683	0.633	1.026
(11) 4 Lugares y 2 hilos	1.112	6.528	0.639	1.028

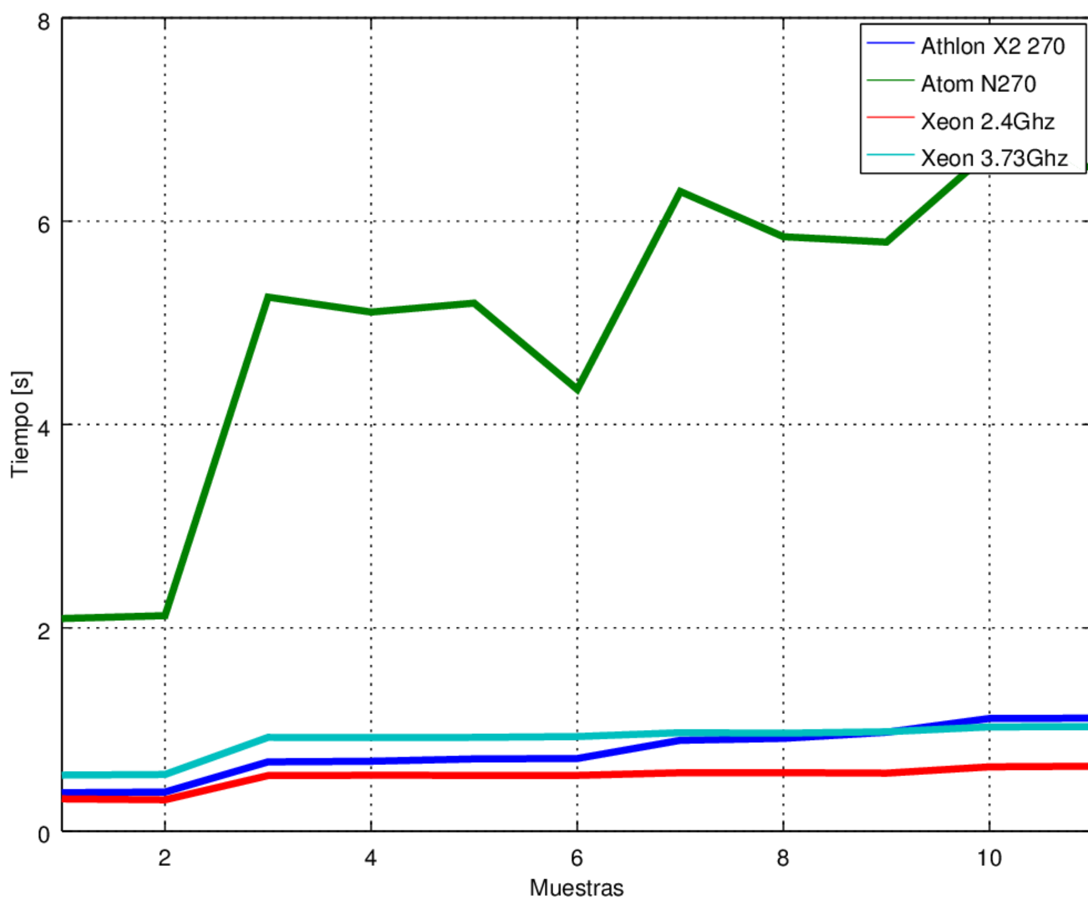


Figura 4.5 Gráfica del tiempo de ejecución del cálculo de Fibonacci en múltiples hilos y lugares

Muestras	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
(1) 1 Lugar y 1 hilo	1.000	1.000	1.000	1.000
(2) 1 Lugar y 2 hilos	0.98442	0.89679	1.02913	0.98925
(3) 2 Lugares y 1 hilo	0.55572	0.39825	0.58135	0.59805
(4) 2 Lugares y 2 hilos	0.55087	0.40971	0.57818	0.59870
(5) 2 Lugares y 3 hilos	0.53305	0.40277	0.57923	0.59805
(6) 2 Lugares y 4 hilos	0.53007	0.48136	0.57923	0.59291
(7) 3 Lugares y 1 hilo	0.42394	0.33249	0.55304	0.56966
(8) 3 Lugares y 2 hilos	0.41512	0.35779	0.55208	0.57261
(9) 3 Lugares y 3 hilos	0.38952	0.36106	0.55692	0.56442
(10) 4 Lugares y 1 hilo	0.34175	0.31303	0.50237	0.53801
(11) 4 Lugares y 2 hilos	0.34083	0.32047	0.49765	0.53696

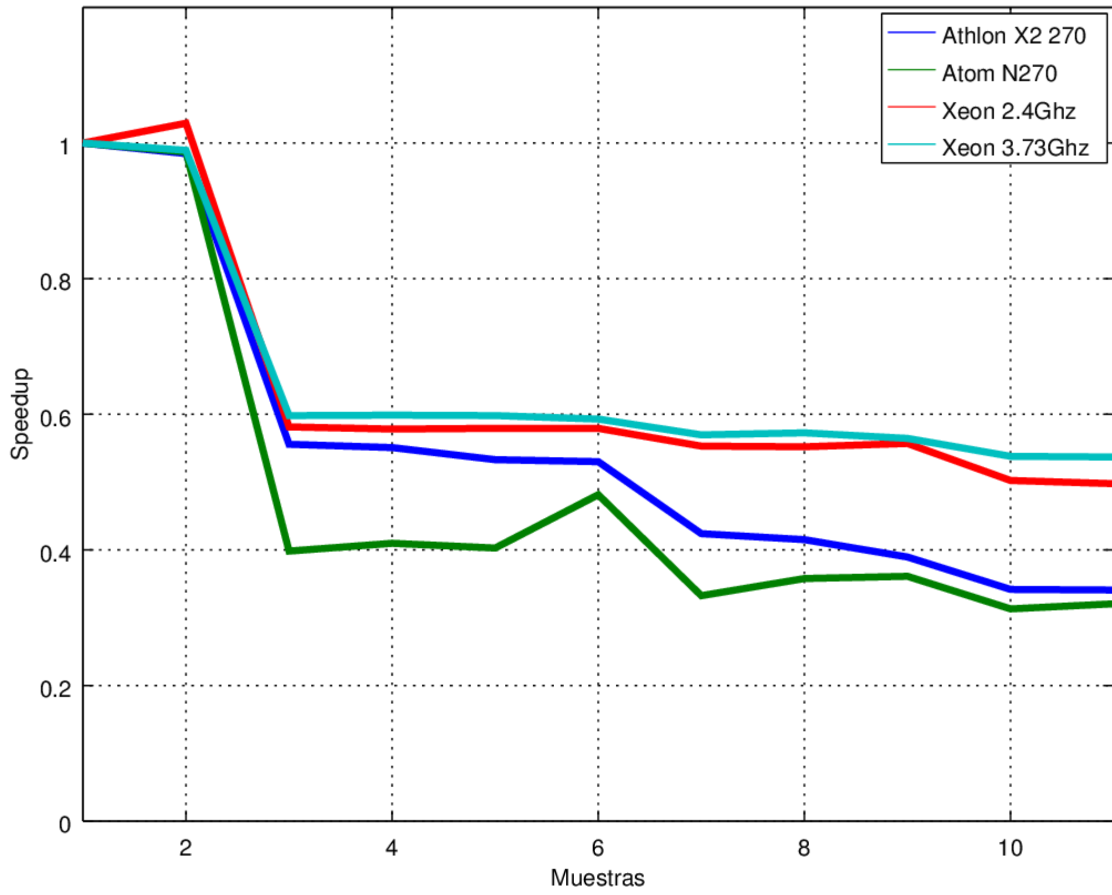


Figura 4.6 Gráfica del Speedup del cálculo de Fibonacci en múltiples hilos y lugares

```
armando@chocochoco: /media/armando/ADATA UFD/tesis/cap4/fib
46368
75025
121393
196418
317811
514229
832040
1346269
2178309
3524578
5702887
9227465
14930352
24157817
39088169

real    0m1.112s
user    0m1.780s
sys     0m0.174s
armando@chocochoco:/media/armando/ADATA UFD/tesis/cap4/fib$
```

Figura 4.7 Resultado de la ejecución del cálculo de Fibonacci en 4 lugares y 2 hilos, en Athlon x2 270

```
armando@chocochoco: /media/armando/ADATA UFD/tesis/cap4/fib
top - 08:48:52 up  1:47,  4 users,  load average: 2.38, 1.00, 0.77
Tareas: 213 total,   3 ejecutar,  210 hibernar,    0 detener,    0 zombie
%Cpu(s): 38.1 usuario,  6.3 sist,  0.0 adecuado, 55.7 inact,  0.0 en espera,  0.0 h
KiB Mem:   3788232 total,  2493396 used,  1294836 free,   148860 buffers
KiB Swap:  1952764 total,        0 used,  1952764 free.  1347124 cached Mem

  PID USUARIO   PR  NI    VIRT    RES    SHR S  %CPU %MEM     HORA+ ORDEN
11283 armando   20   0 2238164  46800  16512 S  20.6  1.2   0:00.62 java
11292 armando   20   0 2238164  43348  16652 S  11.9  1.1   0:00.36 java
11286 armando   20   0 2238164  41496  16372 S  10.6  1.1   0:00.32 java
 1953 armando   20   0 1577564 138460  70160 S   8.3  3.7   3:05.81 compiz
 1169 root      20   0  298748  56212  26928 S   8.0  1.5   5:12.72 Xorg
 2390 armando   20   0  665336  36712  24148 S   3.3  1.0   0:32.68 gnome-terminal
 4363 armando   20   0  417200  23776  18256 S   1.0  0.6   0:20.84 gkrellm
 1768 armando   20   0  363436   8908   5708 S   0.7  0.2   0:33.79 ibus-daemon
 6450 root      20   0       0      0      0 R   0.7  0.0   0:01.01 kworker/u16:1
 9105 root      20   0       0      0      0 S   0.7  0.0   0:00.73 kworker/u16:3
    7 root      20   0       0      0      0 S   0.3  0.0   0:08.21 rcu_sched
    8 root      20   0       0      0      0 S   0.3  0.0   0:04.40 rcuos/0
    9 root      20   0       0      0      0 S   0.3  0.0   0:04.30 rcuos/1
```

Figura 4.8 Salida del comando top durante la ejecución del cálculo de Fibonacci en 3 lugares y 1 hilo, en Athlon x2 270

Cálculo de la secuencia Fibonacci con recursividad

Para el cálculo con recursividad se utiliza el siguiente algoritmo.

```
fib(n)
si n<2 entonces
devuelve n
sino
devuelve fib(n-1)+fib(n-2)
```

El algoritmo aunque sigue presentando la dependencia de datos, se puede paralelizar, al realizar la operación fib(n-1) y fib(n-2) de manera concurrente.
En el siguiente código 4.2 se muestra una implementación en X10 utilizando paralelismo.

Código 4.2

```
import x10.io.Console;
public class Fibonacci {

  public static def fib(n:long) {
    if (n<=2) return 1;

    val f1:long;
    val f2:long;
    finish {                          //barrera explícita
      async { f1 = fib(n-1); }        //paralelizar
      f2 = fib(n-2);                  //llamada recursiva
    }
    return f1 + f2;
  }

  public static def main(args:Rail[String]) {
    val n = (args.size > 0) ? Long.parse(args(0)) : 10;
    Console.OUT.println("Calculando fib("+n+")");
    val f = fib(n);
    Console.OUT.println("fib("+n+") = "+f);
  }
}
```

En los 2 códigos se utiliza una barrera explícita para esperar a que todos los procesos terminen de ejecutarse y devolver el valor correcto. Se puede utilizar cualquiera de los 2 algoritmos, sin embargo al ejecutar de manera recursiva algún programa, la ejecución demanda más recursos al utilizar la pila en cada llamada a la función. Para una demostración del paralelismo o balanceo de carga se recomienda utilizar la recursividad.

Resultados del cálculo de la posición 38

	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
1 Hilo	11.933	51.320	11.569	23.740
2 Hilos	8.009	35.716	7.833	17.390
3 Hilos	8.959	35.736	5.655	15.775
4 Hilos	8.056	39.451	4.452	11.978
5 Hilos	8.299	41.057	4.263	19.716
6 Hilos	8.452	39.688	3.991	11.495
7 Hilos	8.762	42.111	4.326	16.474
8 Hilos	9.455	45.884	4.211	11.059

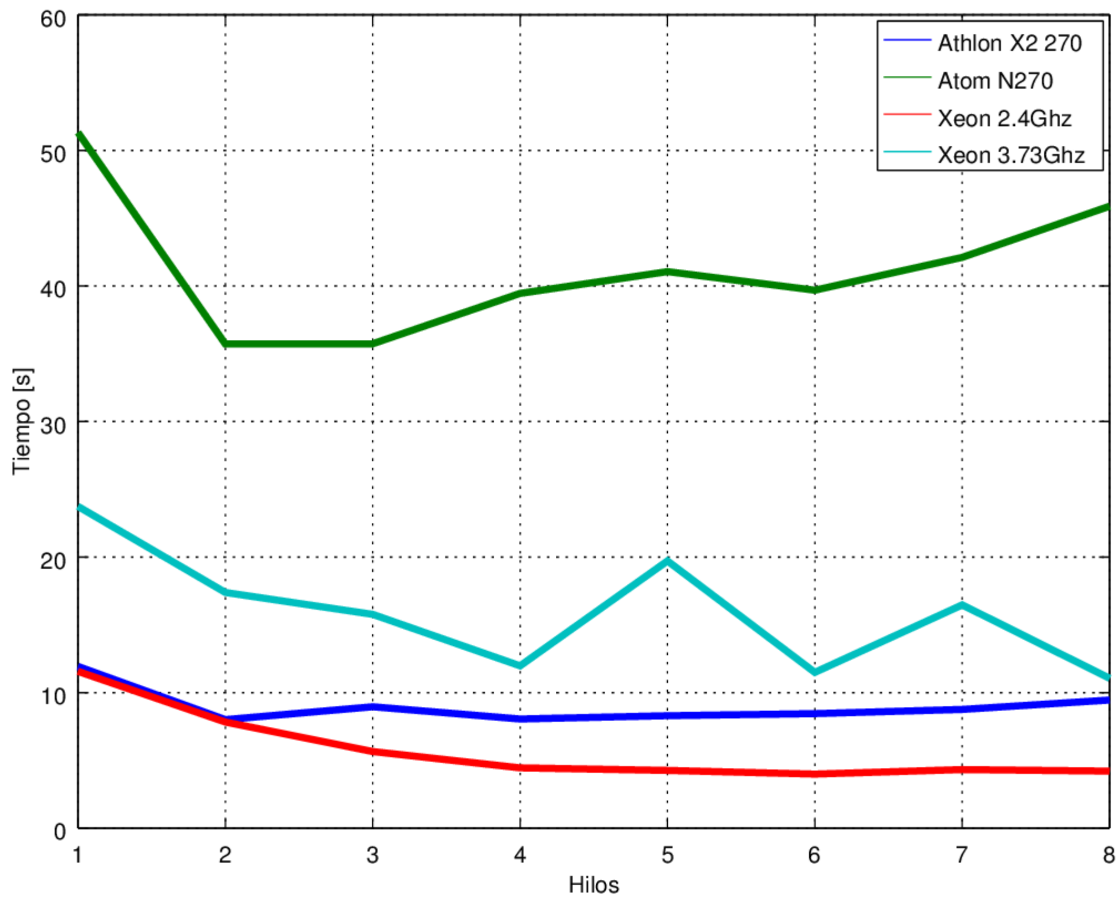


Figura 4.9 Gráfica del tiempo de ejecución del cálculo de Fibonacci con recursividad en múltiples hilos

	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
1 Hilo	1.000	1.000	1.000	1.000
2 Hilos	1.4899	1.4369	1.4770	1.3652
3 Hilos	1.3320	1.4361	2.0458	1.5049
4 Hilos	1.4813	1.3009	2.5986	1.9820
5 Hilos	1.4379	1.2500	2.7138	1.2041
6 Hilos	1.4119	1.2931	2.8988	2.0652
7 Hilos	1.3619	1.2187	2.6743	1.4411
8 Hilos	1.2621	1.1185	2.7473	2.1467

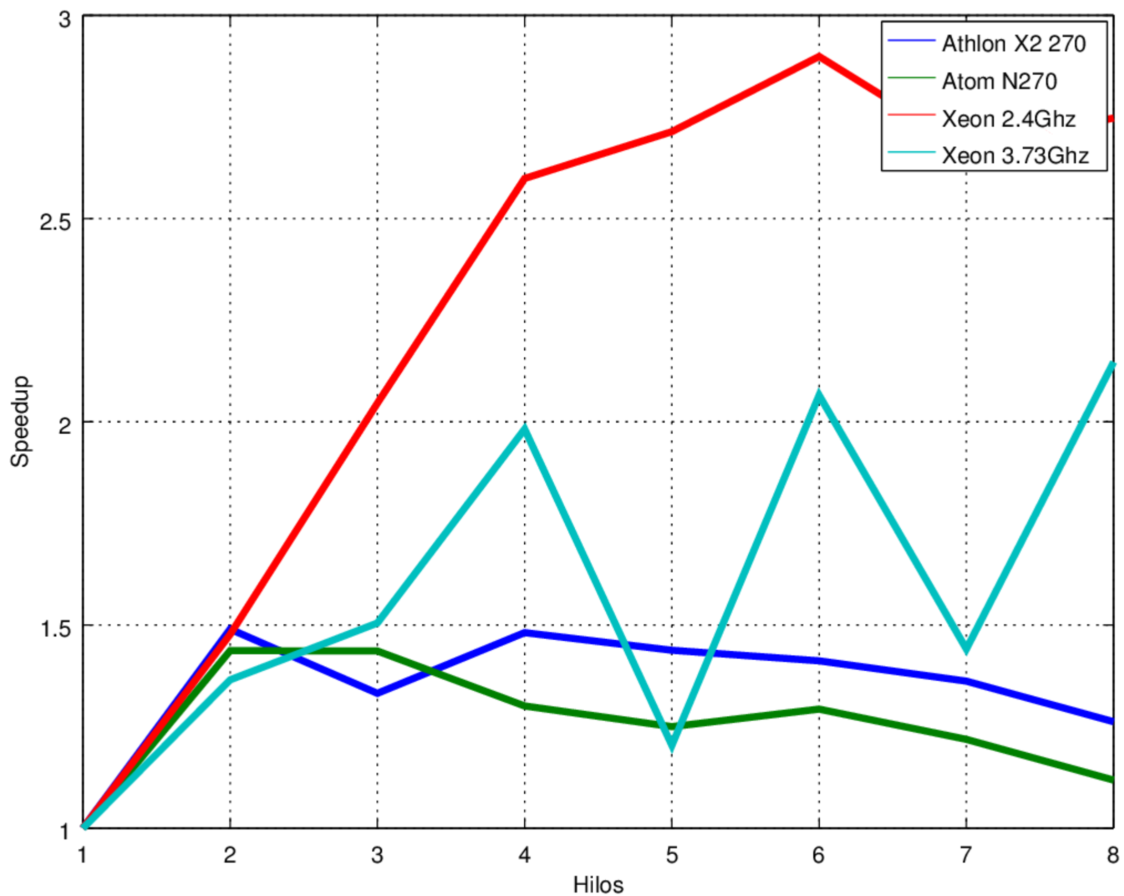


Figura 4.10 Gráfica del Speedup del cálculo de Fibonacci con recursividad en múltiples hilos

	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
1 Lugar	11.933	51.320	11.569	23.740
2 Lugares	12.155	75.872	11.947	24.143
3 Lugares	12.257	120.712	11.977	24.332
4 Lugares	12.723	164.210	11.870	24.371
5 Lugares	12.801	201.687	12.146	25.462
6 Lugares	13.058	246.069	12.179	24.735
7 Lugares	13.260	297.210	12.373	24.695
8 Lugares	13.560	330.126	12.540	24.657

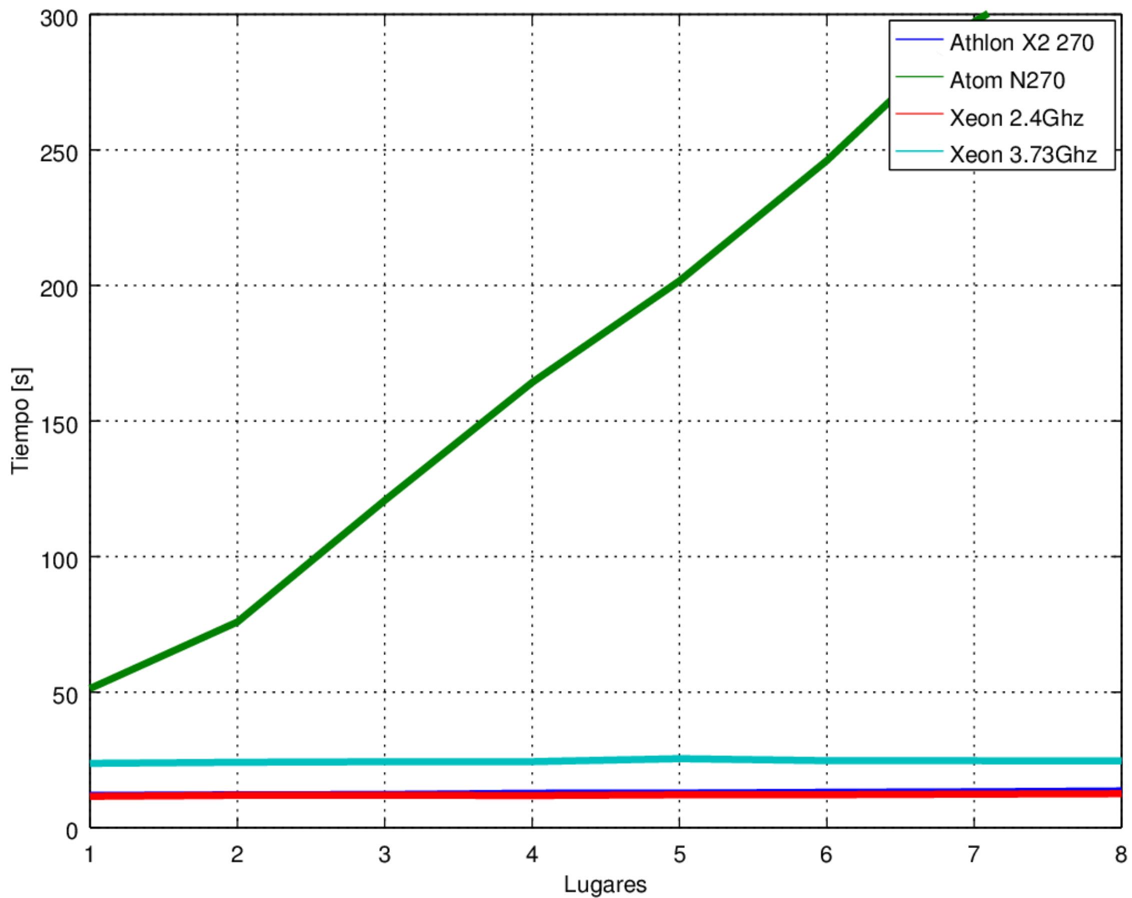


Figura 4.11 Gráfica del tiempo de ejecución del cálculo de Fibonacci con recursividad en múltiples lugares

	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
1 Lugar	1.000	1.000	1.000	1.000
2 Lugares	0.98174	0.67640	0.96836	0.98331
3 Lugares	0.97357	0.42514	0.96593	0.97567
4 Lugares	0.93791	0.31253	0.97464	0.97411
5 Lugares	0.93219	0.25445	0.95249	0.93237
6 Lugares	0.91385	0.20856	0.94991	0.95977
7 Lugares	0.89992	0.17267	0.93502	0.96133
8 Lugares	0.88001	0.15546	0.92257	0.96281

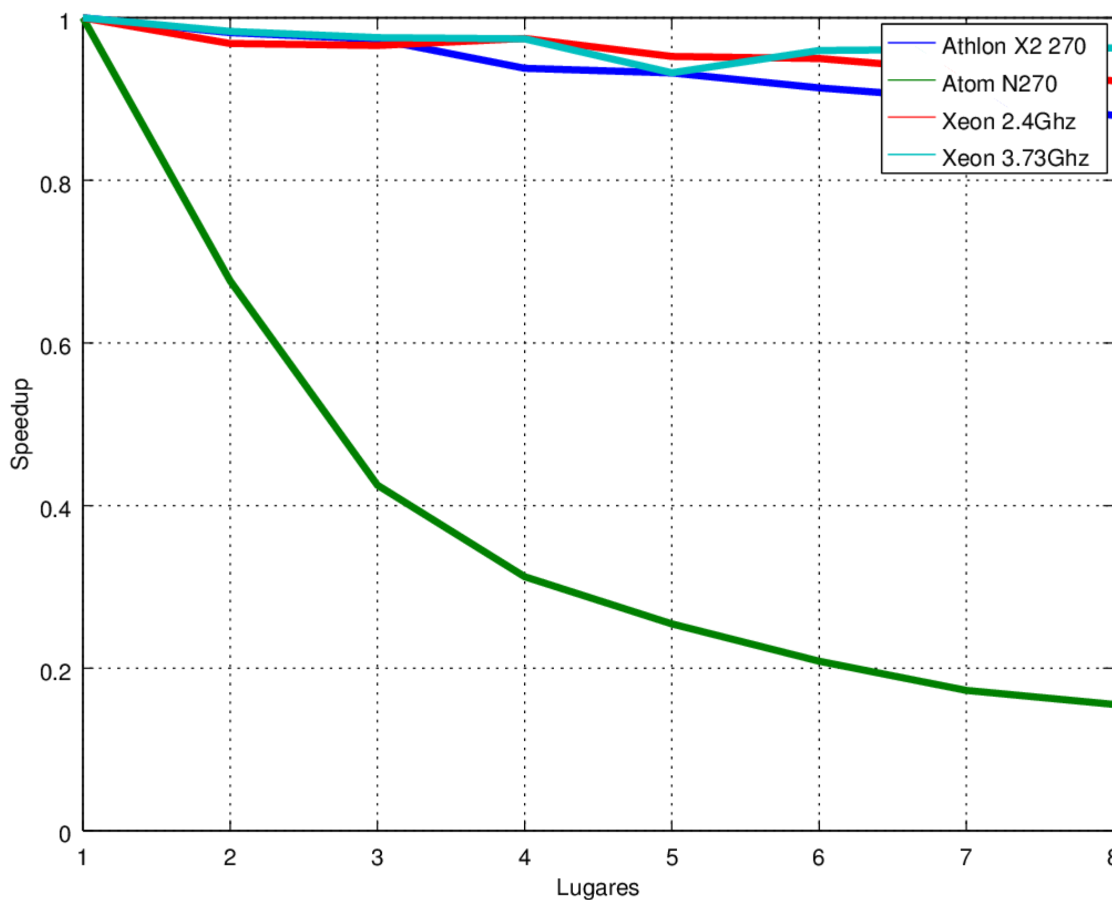


Figura 4.12 Gráfica del Speedup del cálculo de Fibonacci con recursividad en múltiples lugares

Muestras	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
(1) 1 Lugar y 1 hilo	11.933	51.320	11.569	23.740
(2) 1 Lugar y 2 hilos	7.663	35.704	9.244	20.675
(3) 2 Lugares y 1 hilo	11.988	81.678	11.960	24.157
(4) 2 Lugares y 2 hilos	8.364	59.356	7.895	15.426
(5) 2 Lugares y 3 hilos	8.838	55.216	5.717	16.031
(6) 2 Lugares y 4 hilos	8.316	49.421	4.281	17.777
(7) 3 Lugares y 1 hilo	12.452	115.534	11.906	24.241
(8) 3 Lugares y 2 hilos	8.129	79.965	7.389	15.499
(9) 3 Lugares y 3 hilos	8.287	67.116	6.217	14.543
(10) 4 Lugares y 1 hilo	12.672	166.010	12.076	24.242
(11) 4 Lugares y 2 hilos	10.707	98.349	7.930	15.315

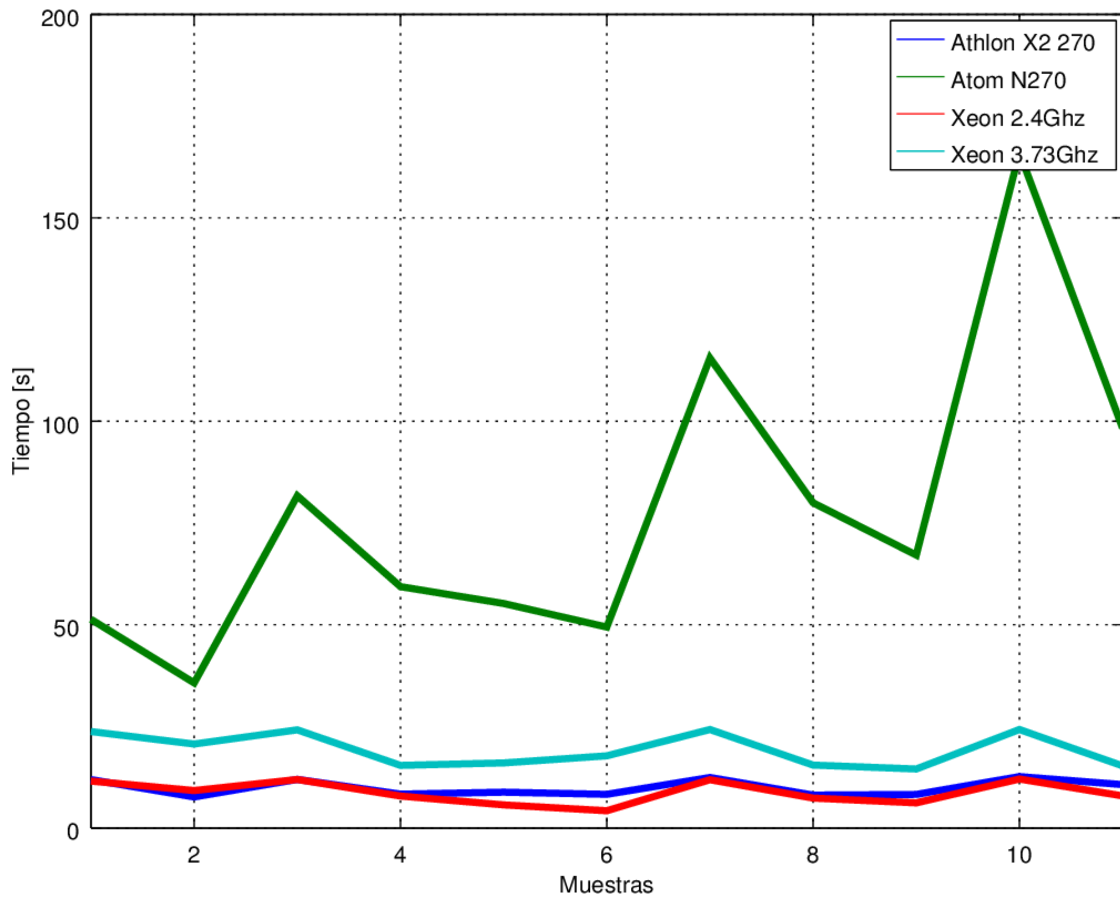


Figura 4.13 Gráfica del tiempo de ejecución del cálculo de Fibonacci con recursividad en múltiples hilos y lugares

Muestras	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
(1) 1 Lugar y 1 hilo	1.000	1.000	1.000	1.000
(2) 1 Lugar y 2 hilos	1.55722	1.43737	1.25151	1.14825
(3) 2 Lugares y 1 hilo	0.99541	0.62832	0.96731	0.98274
(4) 2 Lugares y 2 hilos	1.42671	0.86461	1.46536	1.53896
(5) 2 Lugares y 3 hilos	1.35019	0.92944	2.02361	1.48088
(6) 2 Lugares y 4 hilos	1.43494	1.03842	2.70241	1.33543
(7) 3 Lugares y 1 hilo	0.95832	0.44420	0.97169	0.97933
(8) 3 Lugares y 2 hilos	1.46795	0.64178	1.56571	1.53171
(9) 3 Lugares y 3 hilos	1.43997	0.76465	1.86087	1.63240
(10) 4 Lugares y 1 hilo	0.94168	0.30914	0.95802	0.97929
(11) 4 Lugares y 2 hilos	1.11450	0.52182	1.45889	1.55011

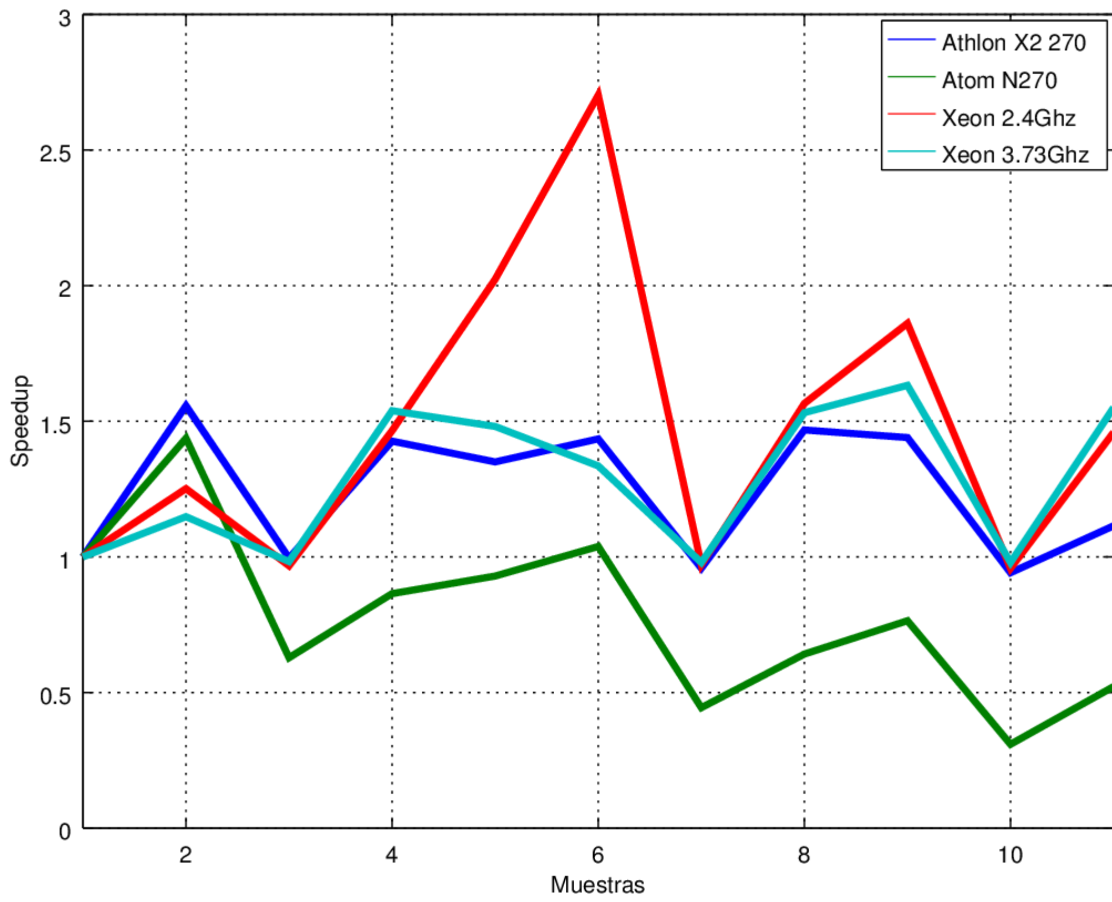


Figura 4.14 Gráfica del Speedup del cálculo de Fibonacci con recursividad en múltiples hilos y lugares

```
alumno01@Olive:~/cap4/fib-r
[alumno01@Olive fib-r]$ time x10 fib 38
Calculando fib(38)
fib(38) = 39088169

real    0m11.059s
user    1m15.817s
sys     0m2.092s
[alumno01@Olive fib-r]$
```

Figura 4.15 Resultado de la ejecución del cálculo de Fibonacci con recursividad con 8 hilos, en Xeon 3.73Ghz

```
alumno01@Olive:~
top - 00:09:45 up 16:00,  2 users,  load average: 2.12, 0.92, 0.76
Tasks: 269 total,   1 running, 268 sleeping,   0 stopped,   0 zombie
Cpu(s): 96.2%us,  0.2%sy,  0.0%ni,  3.6%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  20470396k total,  2160640k used, 18309756k free,    88200k buffers
Swap: 10305528k total,        0k used, 10305528k free,   452588k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
18446 alumno01  20   0 8141m 1.0g  10m S 769.4  5.3   1:35.76 java
12112 alumno01  20   0 15168 1344  904 R  0.3  0.0   0:08.51 top
    1 root      20   0 19356 1508 1192 S  0.0  0.0   0:00.80 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.02 ksoftirqd/0
    5 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    6 root      RT   0     0    0    0 S  0.0  0.0   0:00.04 watchdog/0
    7 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/1
    8 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/1
    9 root      20   0     0    0    0 S  0.0  0.0   0:00.03 ksoftirqd/1
   10 root      RT   0     0    0    0 S  0.0  0.0   0:00.03 watchdog/1
```

Figura 4.16 Salida del comando top durante la ejecución del cálculo de Fibonacci con recursividad con 8 hilos, en Xeon 3.73Ghz

Cálculo de PI

Existen diferentes algoritmos para el cálculo del Número PI, sin embargo se utilizan 2 algoritmos en este trabajo.
El primero de ellos es por la aproximación a la integral definida de 0 a 1:

$$\int \left(\frac{4}{1+x^2} \right)$$

La aproximación de la integral mediante la suma de Riemann permite dividir el trabajo en unidades independientes, lo cual es necesario para paralelizar el cálculo. En la siguiente figura 4.17 se muestra como a través de la regla del rectángulo, cada pedazo se puede calcular de manera independiente.

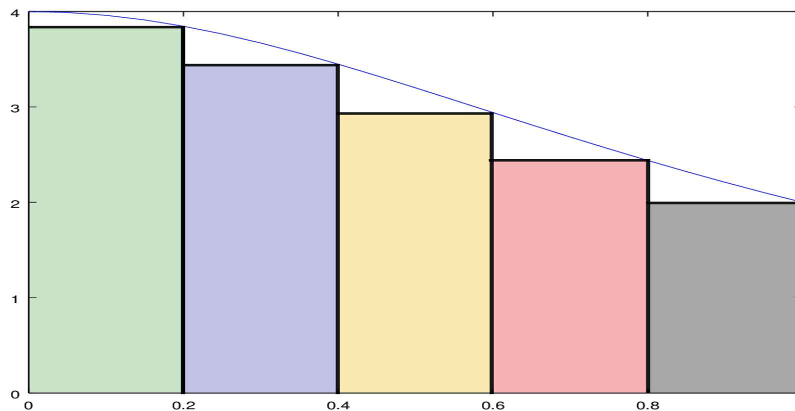


Figura 4.17 Cálculo del número PI por la aproximación a la integral definida de 0 a 1

La implementación en X10 del algoritmo se muestra a continuación. (véase código 4.3)

Código 4.3

```
import x10.io.Console;

public class pi{

 public static def main(args:Rail[String]) {
val total:Long=1000000000;
var paso:Double;
var x:Double;
var pi:Double;
var sum:Double=0.0;
var i:Long;

paso=1.0/total;
```

```
finish {
  async{
              for(i=0;i<total;i++)
             {
                  x=(i+.5)*paso;
              sum=sum+4.0/(1.0+x*x);
          }
       }
}


pi=sum*paso;
Console.OUT.println("PI es: " + pi);

}

}
```

En este algoritmo se utiliza un barrera explícita para esperar que terminen de ejecutarse todas los procesos y al final se realiza la suma de todos los pedazos o unidades independientes en el procesador donde se ejecutó el programa.

Resultados del cálculo

	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
1 Hilo	8.152	81.086	9.103	11.528
2 Hilos	8.159	81.707	9.003	11.546
3 Hilos	8.145	81.020	9.002	11.527
4 Hilos	8.153	80.915	9.007	11.527
5 Hilos	8.154	80.946	9.094	11.534
6 Hilos	8.165	80.941	9.115	11.527
7 Hilos	8.146	81.013	9.010	11.540
8 Hilos	8.179	81.002	9.126	11.527

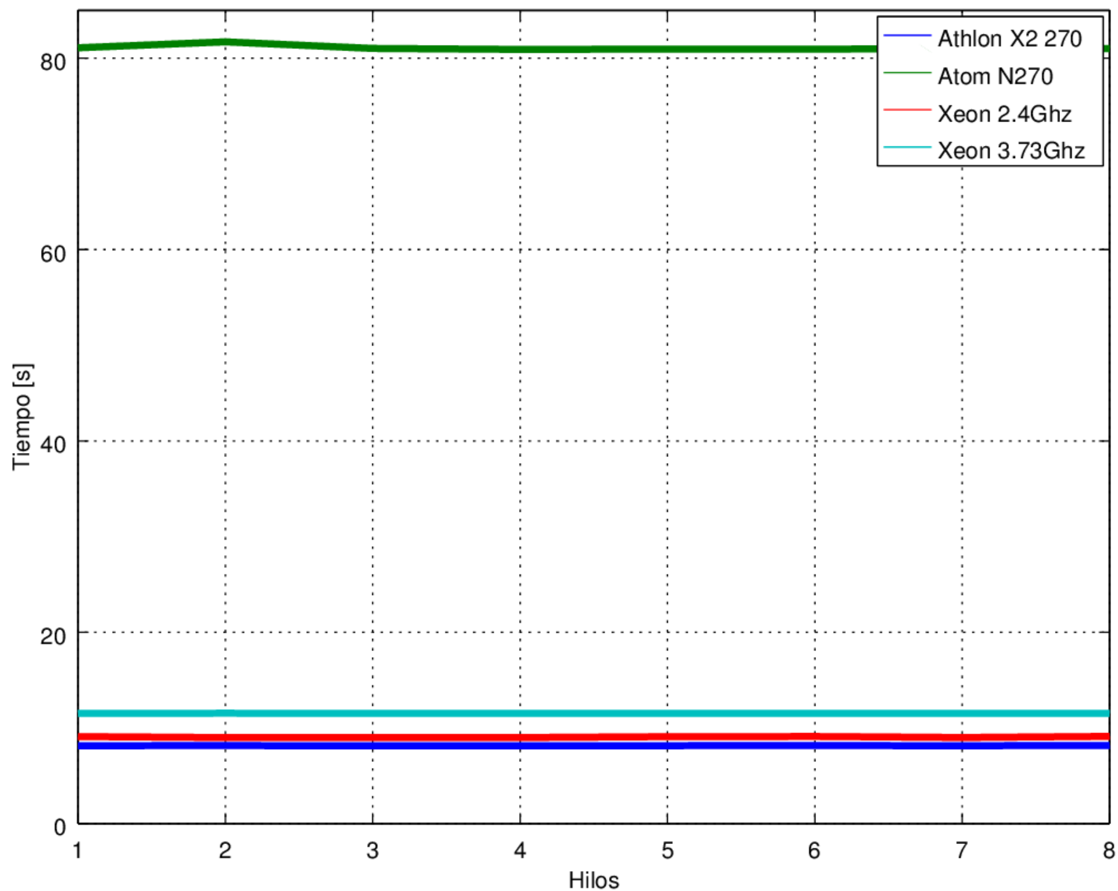


Figura 4.18 Gráfica del tiempo de ejecución del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en múltiples hilos

	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
1 Hilo	1.000	1.000	1.000	1.000
2 Hilos	0.99914	0.99240	1.01111	0.99844
3 Hilos	1.00086	1.00081	1.01122	1.00009
4 Hilos	0.99988	1.00211	1.01066	1.00009
5 Hilos	0.99975	1.00173	1.00099	0.99948
6 Hilos	0.99841	1.00179	0.99868	1.00009
7 Hilos	1.00074	1.00090	1.01032	0.99896
8 Hilos	0.99670	1.00104	0.99748	1.00009

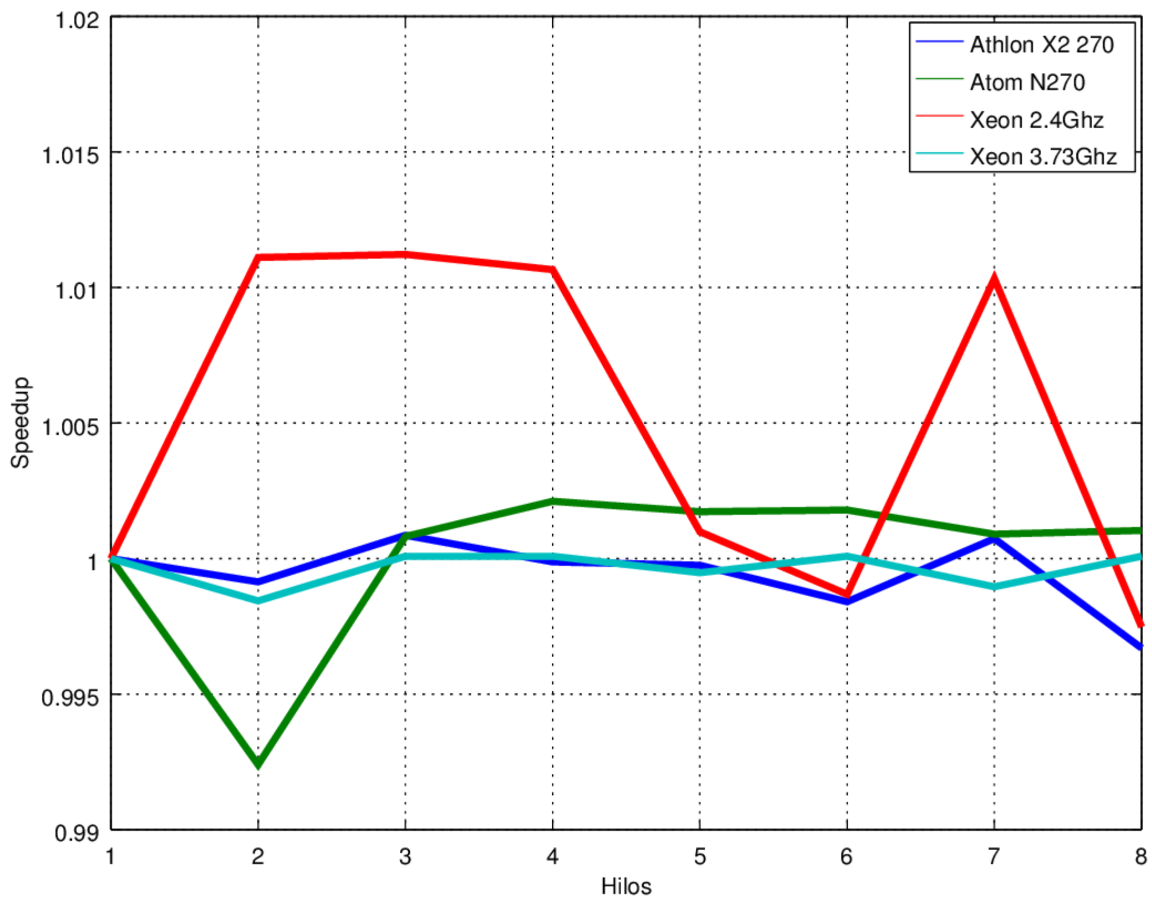


Figura 4.19 Gráfica del Speedup del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en múltiples hilos

	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
1 Lugar	8.152	81.086	9.103	11.528
2 Lugares	8.464	93.662	9.297	11.912
3 Lugares	8.804	130.474	9.309	11.968
4 Lugares	9.008	183.841	9.359	11.995
5 Lugares	9.269	223.945	9.472	12.197
6 Lugares	9.382	279.137	9.690	12.357
7 Lugares	9.632	327.450	9.478	12.460
8 Lugares	9.902	374.857	9.739	12.697

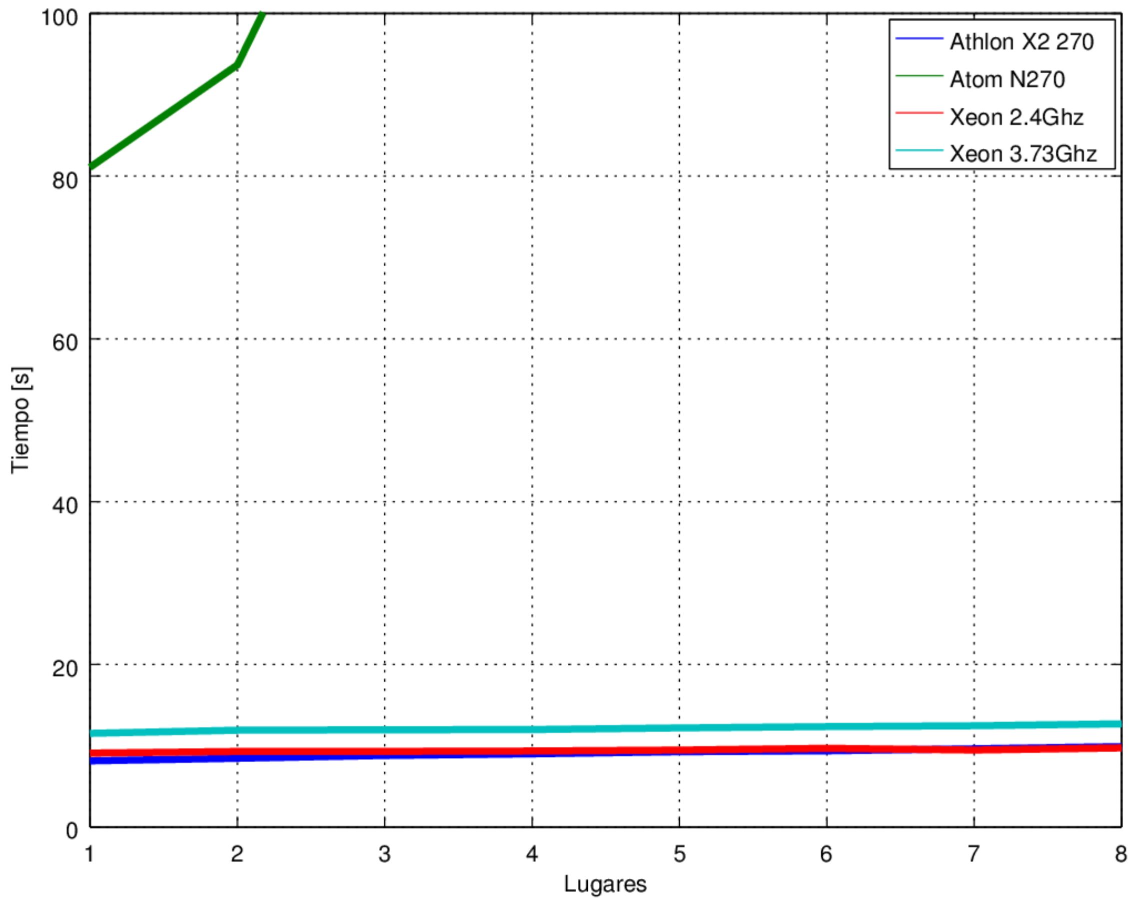


Figura 4.20 Gráfica del tiempo de ejecución del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en múltiples lugares

	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
1 Lugar	1.000	1.000	1.000	1.000
2 Lugares	0.96314	0.86573	0.97913	0.96776
3 Lugares	0.92594	0.62147	0.97787	0.96324
4 Lugares	0.90497	0.44107	0.97265	0.96107
5 Lugares	0.87949	0.36208	0.96104	0.94515
6 Lugares	0.86890	0.29049	0.93942	0.93291
7 Lugares	0.84635	0.24763	0.96043	0.92520
8 Lugares	0.82327	0.21631	0.93470	0.90793

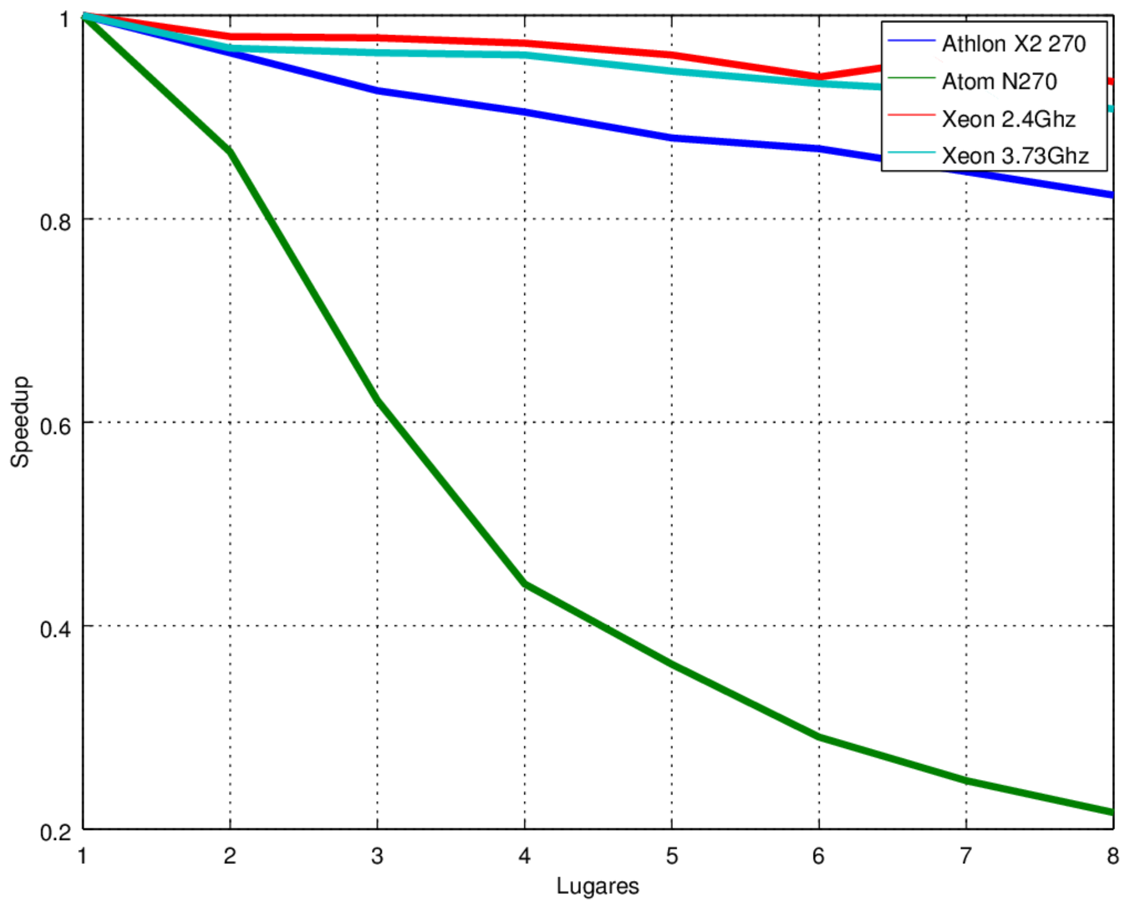


Figura 4.21 Gráfica del Speedup de ejecución del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en múltiples lugares

Muestras	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
(1) 1 Lugar y 1 hilo	8.152	81.086	9.103	11.528
(2) 1 Lugar y 2 hilos	8.164	81.011	9.093	11.543
(3) 2 Lugares y 1 hilo	8.481	96.399	9.293	11.881
(4) 2 Lugares y 2 hilos	8.482	94.467	9.332	11.886
(5) 2 Lugares y 3 hilos	8.480	93.264	9.314	11.883
(6) 2 Lugares y 4 hilos	8.506	93.996	9.224	11.906
(7) 3 Lugares y 1 hilo	8.749	130.155	9.321	11.988
(8) 3 Lugares y 2 hilos	8.643	128.757	9.353	11.964
(9) 3 Lugares y 3 hilos	8.721	131.601	9.338	11.934
(10) 4 Lugares y 1 hilo	8.914	185.101	9.378	11.984
(11) 4 Lugares y 2 hilos	8.883	185.879	9.413	11.987

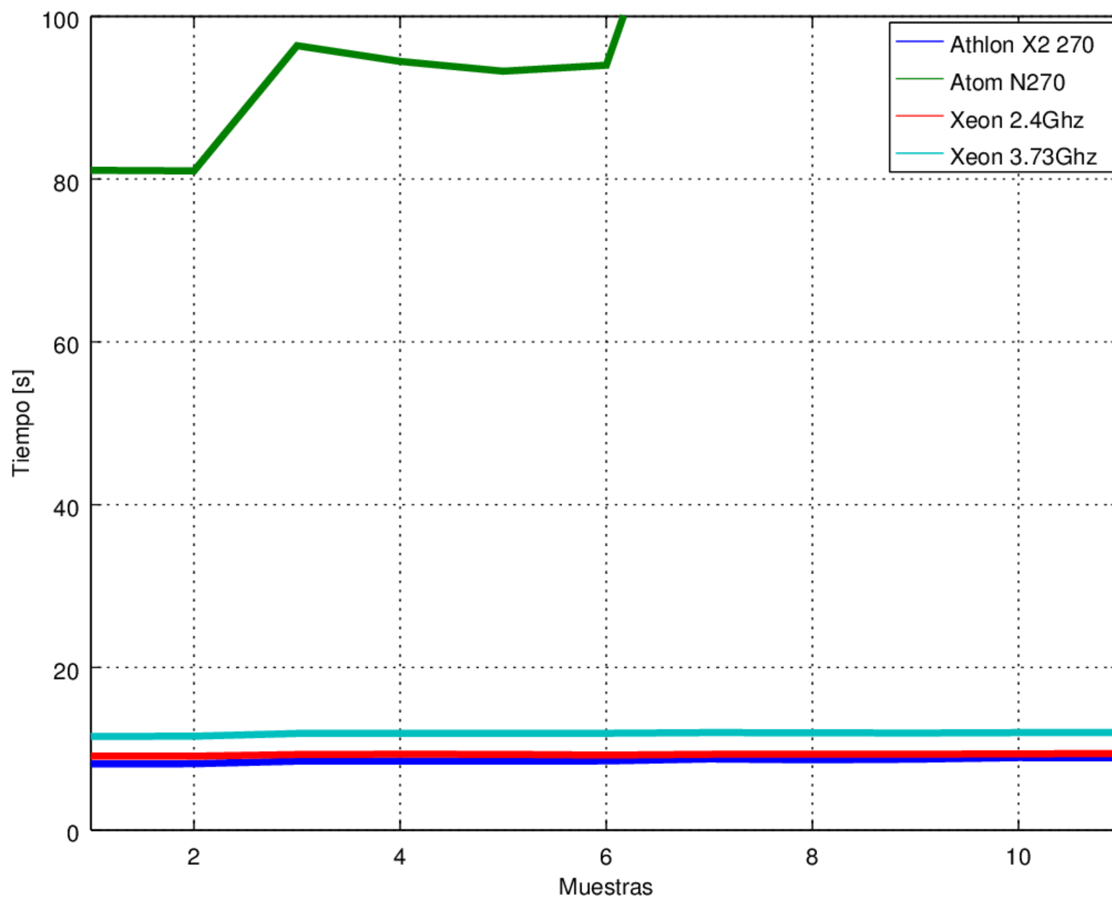


Figura 4.22 Gráfica del tiempo de ejecución del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en múltiples hilos y lugares

Muestras	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
(1) 1 Lugar y 1 hilo	1.000	1.000	1.000	1.000
(2) 1 Lugar y 2 hilos	0.99853	1.00093	1.00110	0.99870
(3) 2 Lugares y 1 hilo	0.96121	0.84115	0.97955	0.97029
(4) 2 Lugares y 2 hilos	0.96109	0.85835	0.97546	0.96988
(5) 2 Lugares y 3 hilos	0.96132	0.86942	0.97735	0.97013
(6) 2 Lugares y 4 hilos	0.95838	0.86265	0.98688	0.96825
(7) 3 Lugares y 1 hilo	0.93176	0.62300	0.97661	0.96163
(8) 3 Lugares y 2 hilos	0.94319	0.62976	0.97327	0.96356
(9) 3 Lugares y 3 hilos	0.93476	0.61615	0.97483	0.96598
(10) 4 Lugares y 1 hilo	0.91452	0.43806	0.97068	0.96195
(11) 4 Lugares y 2 hilos	0.91771	0.43623	0.96707	0.96171

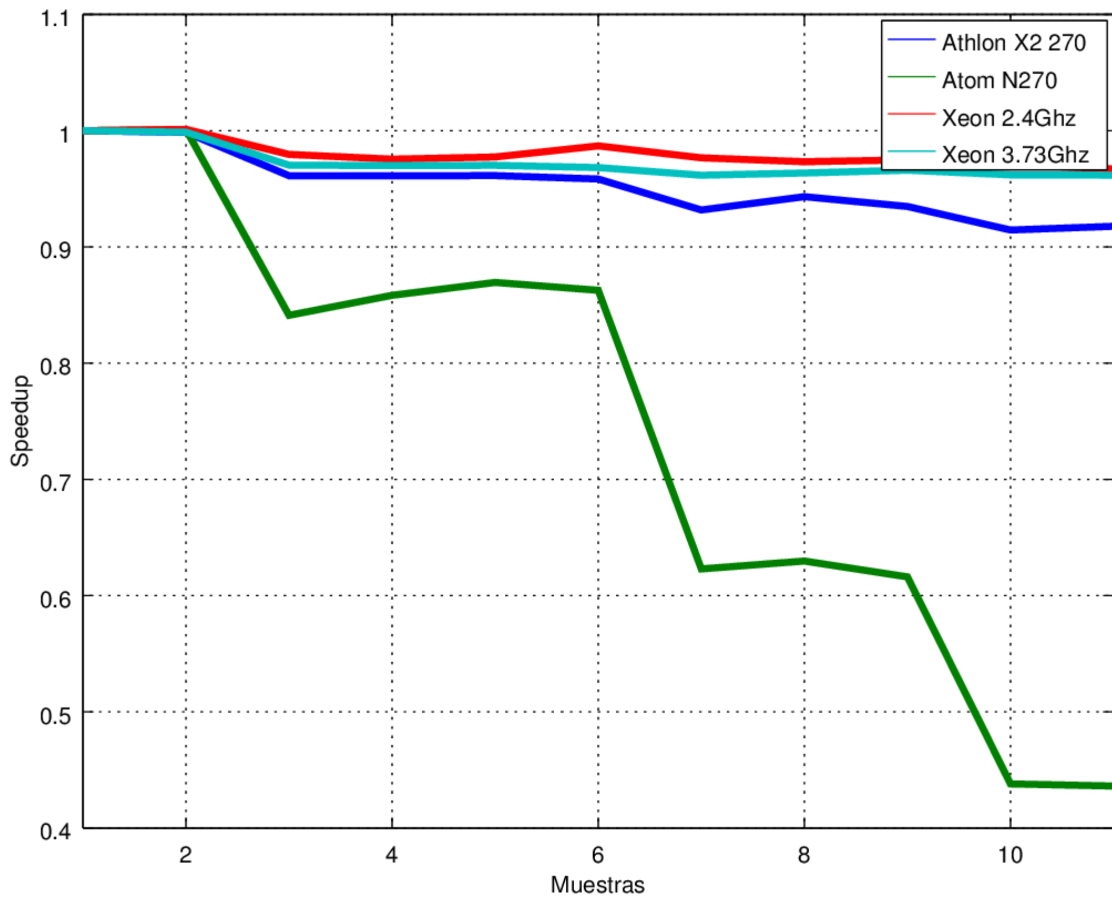


Figura 4.23 Gráfica del Speedup del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en múltiples hilos y lugares

```
xterm
bash-4.2# export X10_NPLACES=2
bash-4.2# export X10_NTHREADS=1
bash-4.2# time /root/x10/bin/x10 pi
PI es: 3.1415926535899708

real    1m36.399s
user    2m16.121s
sys     0m40.243s
bash-4.2#
```

Figura 4.24 Resultado de la ejecución del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en 2 lugares y 1 hilo, en Atom N270

```
xterm
top - 22:51:16 up  1:02,  2 users,  load average: 0.55, 2.31, 3.57
Tasks:  61 total,   1 running,  60 sleeping,   0 stopped,   0 zombie
Cpu(s): 76.6%us, 23.4%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.(
Mem:   2061712k total,   196900k used,  1864812k free,       12k buffers
Swap:        0k total,        0k used,        0k free,   105784k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3664 root      20   0  647m  18m 7728 S   96  0.9   0:12.74 java
 3665 root      20   0  646m  18m 7632 S   95  0.9   0:12.79 java
 2356 root      20   0 32712  11m 5032 S    5  0.6   1:42.18 X
 3690 root      20   0 36488  11m 9296 S    2  0.6   0:00.58 screenshot
 2374 root      20   0 10644 5128 3768 S    1  0.2   0:42.54 e16
 3592 root      20   0  2636  996  824 R    1  0.0   0:00.72 top
 3635 root      20   0 76084  34m  15m S    1  1.7   0:07.80 gimp
 2482 root      20   0  5032 1308 1084 S    0  0.1   0:08.11 E-Load.epplet
 2522 root      20   0 11752 5560 2292 S    0  0.3   0:00.48 xterm
    1 root      20   0  2072  564  500 S    0  0.0   0:01.50 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S    0  0.0   0:00.01 ksoftirqd/0
    4 root      20   0     0    0    0 S    0  0.0   0:00.11 kworker/0:0
    6 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    7 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
    8 root      20   0     0    0    0 S    0  0.0   0:00.07 kworker/1:0
    9 root      20   0     0    0    0 S    0  0.0   0:00.00 ksoftirqd/1
```

Figura 4.25 Salida del comando top durante la ejecución del cálculo del número PI por la aproximación a la integral definida de 0 a 1 en 2 lugares y 1 hilo, en Atom N270

Cálculo de PI utilizando el Algoritmo de Montecarlo

El método de Montecarlo consiste en dibujar dentro de un cuadrado de lado 1 , una circunferencia con radio 1, se trata de generar coordenadas x,y aleatorias de rango 0 a 1 .Y calcular pi, utilizando la siguiente fórmula.

$$\frac{Númerosdentrodelcírculo}{TotaldeNúmeros}=\frac{Areadelcírculo}{Areacuadrado}$$

$$\frac{Númerosdentrodelcículo}{TotaldeNúmeros}=\frac{\pi r^2}{4\mathrm{r}^2}$$

$$\pi=4\,\frac{Númerosdentrodelcírculo}{TotaldeNúmeros}$$

De forma gráfica podemos ver una ejecución del algoritmo en la figura 4.26.

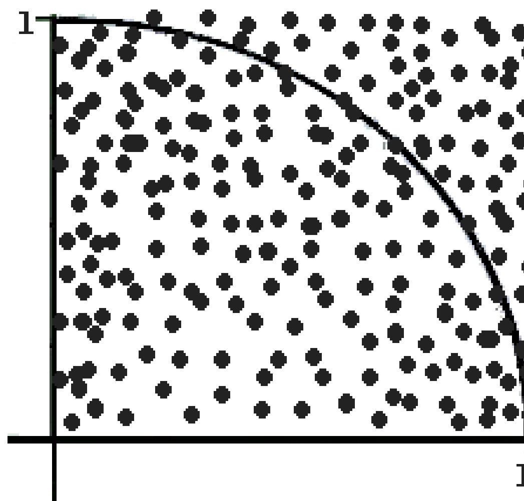


Figura 4.26 Cálculo del número PI por el algoritmo de Montecarlo

La implementación paralela en X10 del algoritmo se muestra a continuación. (véase código 4.4)

Código 4.4

```
import x10.io.Console;
import x10.util.Random;

public class pi_mont{

public static def main(args:Rail[String]){

val r= new Random( );
```

```
var resultado:Double=0;
var i:Long;

val N = (args.size > 0) ? Long.parse(args(0)) : 100000000;
Console.OUT.println("Calculando PI con "+N + " iteracciones");


finish{
        async{
                for(i=0;i<N;i++)
                {
                      val x=r.nextDouble();
                      val y=r.nextDouble();
                      if(x*x + y*y <=1)resultado++;
                }
        }
}


val pi=4*resultado/N;

Console.OUT.println("PI: "+pi);
}

}
```

Para el cálculo de pi existen muchos métodos, el primer método se utilizó en la materia de cómputo de alto desempeño y el segundo se utiliza en la documentación oficial de X10.Los 2 métodos se pueden paralelizar obteniendo resultados similares. Ambos no tienen dependencia de datos, y comparten la barrera explícita para mostrar el resultado correcto, ya que los algoritmos son Seriales-Paralelos.

Resultados del cálculo

	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
1 Hilo	2.053	25.762	2.369	4.049
2 Hilos	2.040	25.757	2.315	4.044
3 Hilos	2.067	25.746	2.310	4.042
4 Hilos	2.063	25.769	2.334	4.042
5 Hilos	2.073	25.753	2.315	4.051
6 Hilos	2.074	26.203	2.332	4.058
7 Hilos	2.065	25.825	2.337	4.054
8 Hilos	2.057	25.774	2.337	4.057

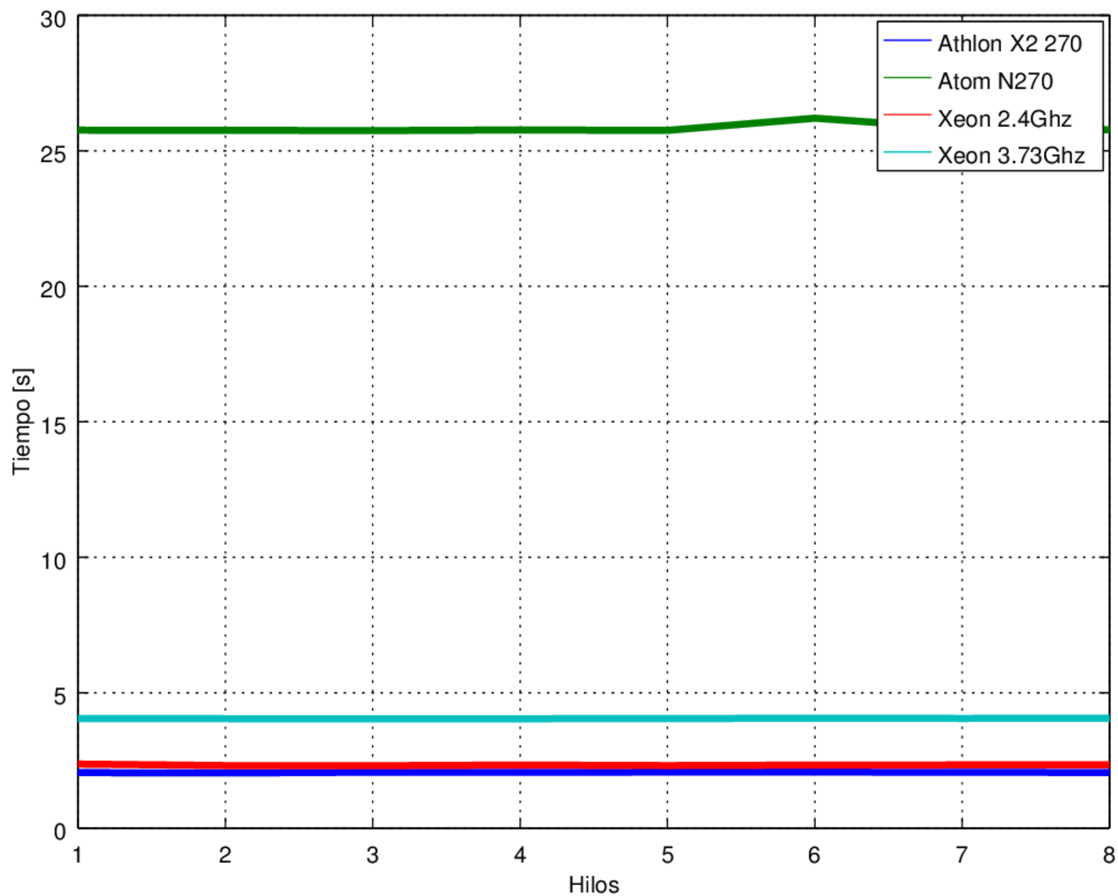


Figura 4.27 Gráfica del tiempo de ejecución del cálculo del número PI por el algoritmo de Montecarlo en múltiples hilos

	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
1 Hilo	1.000	1.000	1.000	1.000
2 Hilos	1.00637	1.00019	1.0233	1.00124
3 Hilos	0.99323	1.00062	1.0255	1.00173
4 Hilos	0.99515	0.99973	1.0150	1.00173
5 Hilos	0.99035	1.00035	1.0233	0.99951
6 Hilos	0.98987	0.98317	1.0159	0.99778
7 Hilos	0.99419	0.99756	1.0137	0.99877
8 Hilos	0.99806	0.99953	1.0137	0.99803

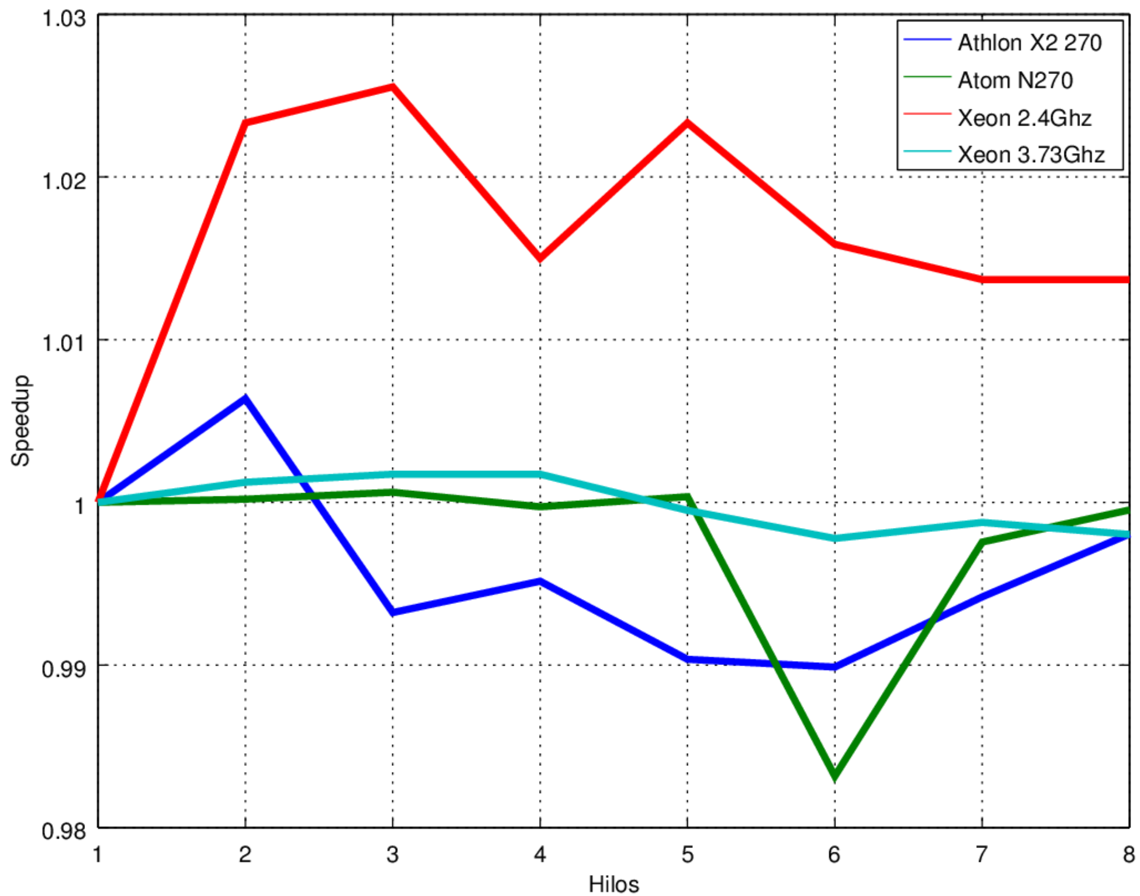


Figura 4.28 Gráfica del Speedup del cálculo del número PI por el algoritmo de Montecarlo en múltiples hilos

	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
1 Lugar	2.053	25.762	2.369	4.049
2 Lugares	2.383	35.581	2.555	4.420
3 Lugares	2.564	49.825	2.560	4.442
4 Lugares	2.910	70.917	2.593	4.552
5 Lugares	3.200	85.599	2.730	4.701
6 Lugares	3.389	106.925	2.844	4.947
7 Lugares	3.500	124.986	2.915	4.947
8 Lugares	3.790	142.934	2.973	5.041

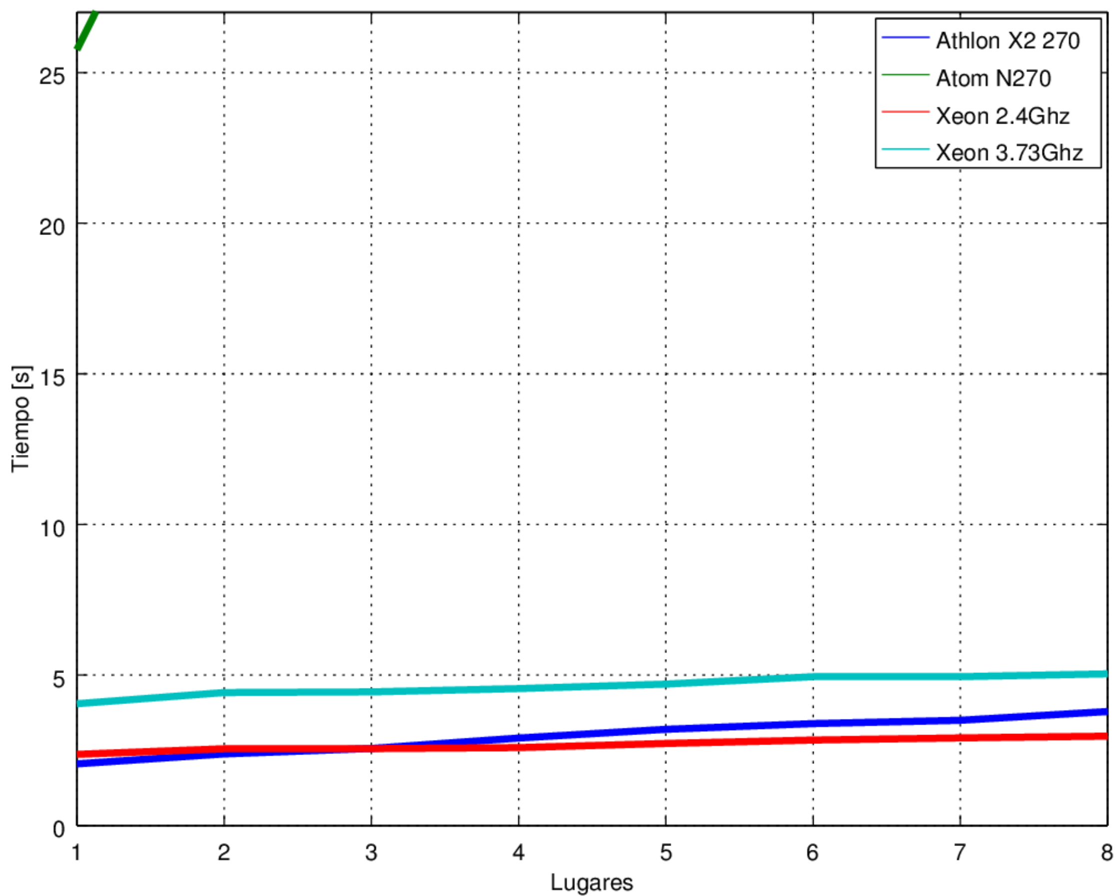


Figura 4.29 Gráfica del tiempo de ejecución del cálculo del número PI por el algoritmo de Montecarlo en múltiples lugares

	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
1 Lugar	1.000	1.000	1.000	1.000
2 Lugares	0.86152	0.72404	0.92720	0.91606
3 Lugares	0.80070	0.51705	0.92539	0.91153
4 Lugares	0.70550	0.36327	0.91361	0.88950
5 Lugares	0.64156	0.30096	0.86777	0.86131
6 Lugares	0.60578	0.24094	0.83298	0.81848
7 Lugares	0.58657	0.20612	0.81269	0.81848
8 Lugares	0.54169	0.18024	0.79684	0.80321

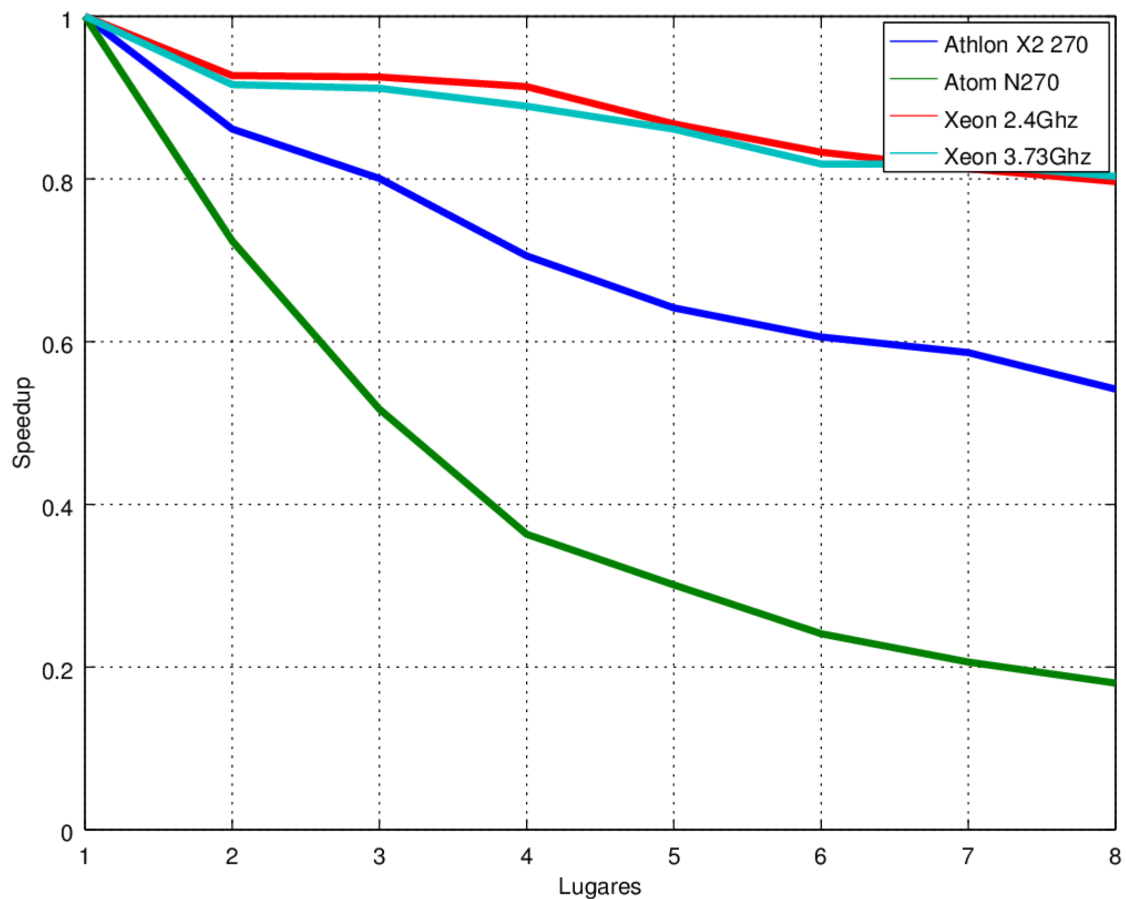


Figura 4.30 Gráfica del Speedup del cálculo del número PI por el algoritmo de Montecarlo en múltiples lugares

Muestras	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
(1) 1 Lugar y 1 hilo	2.053	25.762	2.369	4.049
(2) 1 Lugar y 2 hilos	2.064	25.976	2.331	4.042
(3) 2 Lugares y 1 hilo	2.373	39.329	2.575	4.418
(4) 2 Lugares y 2 hilos	2.363	36.128	2.568	4.424
(5) 2 Lugares y 3 hilos	2.365	36.519	2.588	4.424
(6) 2 Lugares y 4 hilos	2.403	35.740	2.612	4.427
(7) 3 Lugares y 1 hilo	2.597	48.964	2.568	4.480
(8) 3 Lugares y 2 hilos	2.626	49.512	2.614	4.474
(9) 3 Lugares y 3 hilos	2.610	52.493	2.612	4.485
(10) 4 Lugares y 1 hilo	2.771	72.164	2.671	4.526
(11) 4 Lugares y 2 hilos	2.805	69.599	2.615	4.522

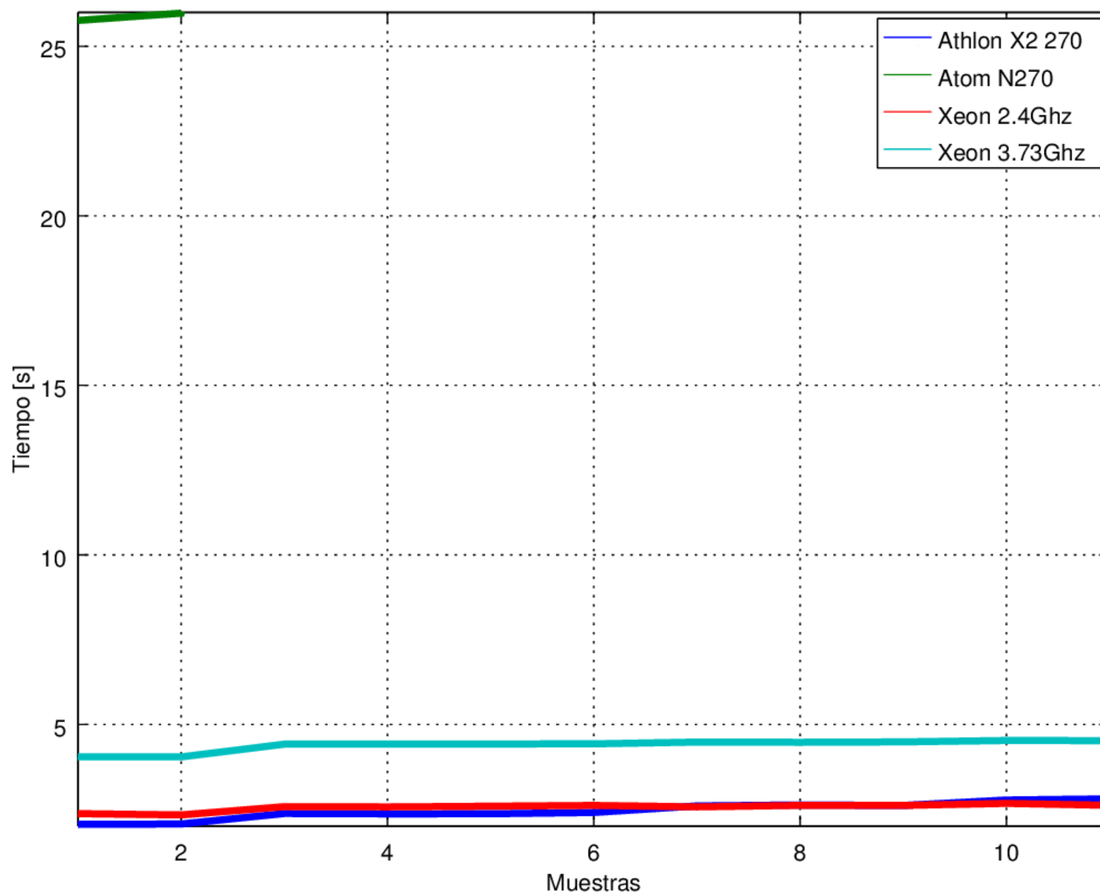


Figura 4.31 Gráfica del tiempo de ejecución del cálculo del número PI por el algoritmo de Montecarlo en múltiples hilos y lugares

Muestras	**Speedup en AMD Athlon x2 270**	**Speedup en Intel Atom N270 HT**	**Speedup en Intel Xeon 2.4Ghz**	**Speedup en Intel Xeon 3.73Ghz**
(1) 1 Lugar y 1 hilo	1.000	1.000	1.000	1.000
(2) 1 Lugar y 2 hilos	0.99467	0.99176	1.01630	1.00173
(3) 2 Lugares y 1 hilo	0.86515	0.65504	0.92000	0.91648
(4) 2 Lugares y 2 hilos	0.86881	0.71308	0.92251	0.91524
(5) 2 Lugares y 3 hilos	0.86808	0.70544	0.91538	0.91524
(6) 2 Lugares y 4 hilos	0.85435	0.72082	0.90697	0.91461
(7) 3 Lugares y 1 hilo	0.79053	0.52614	0.92251	0.90379
(8) 3 Lugares y 2 hilos	0.78180	0.52032	0.90627	0.90501
(9) 3 Lugares y 3 hilos	0.78659	0.49077	0.90697	0.90279
(10) 4 Lugares y 1 hilo	0.74089	0.35699	0.88693	0.89461
(11) 4 Lugares y 2 hilos	0.73191	0.37015	0.90593	0.89540

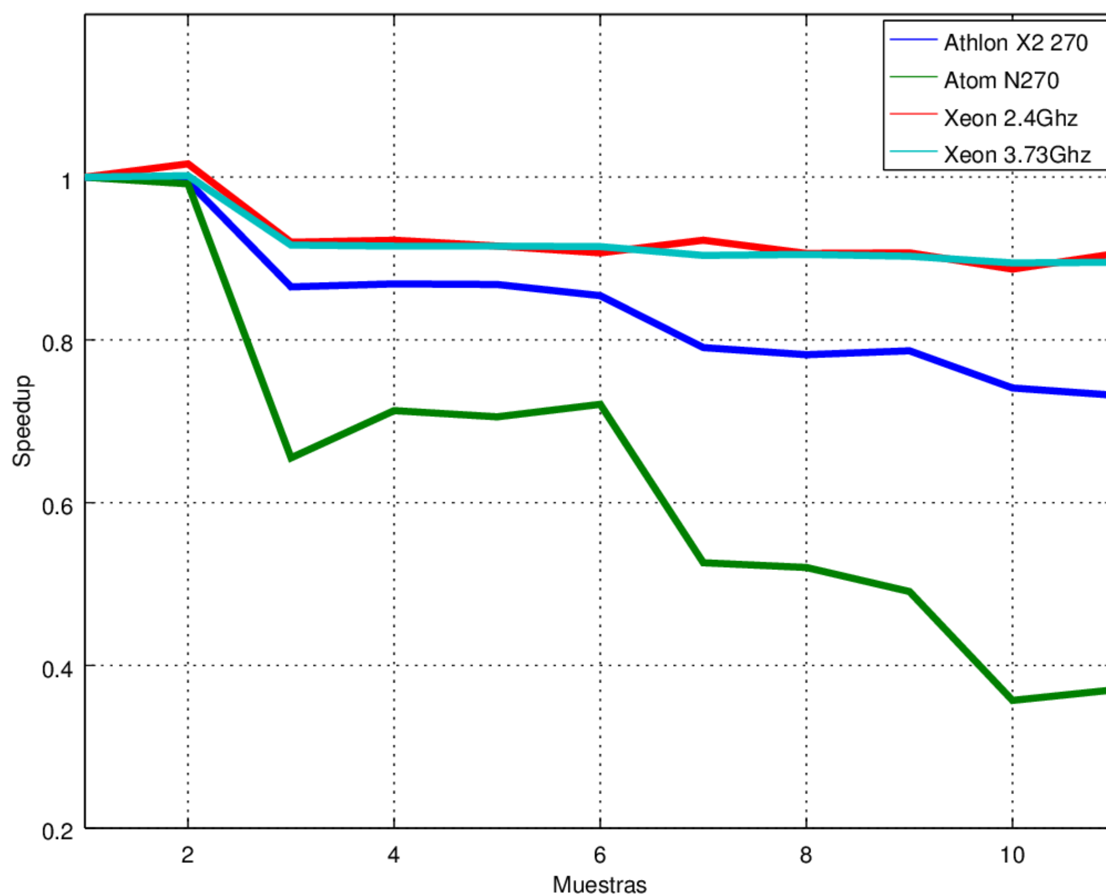


Figura 4.32 Gráfica del Speedup del cálculo del número PI por el algoritmo de Montecarlo en múltiples hilos y lugares

```
xterm
bash-4.2# time /root/x10/bin/x10 pi_mont
Calculando PI con 100000000 iteracciones
PI: 3.1413768

real    0m39.329s
user    0m50.247s
sys     0m14.505s
bash-4.2# 
```

Figura 4.33 Resultado de la ejecución del cálculo del número PI por el algoritmo de Montecarlo en 2 lugares y 1 hilo, en Atom N270

```
xterm
top - 00:07:34 up  2:18,  2 users,  load average: 1.64, 3.46, 2.72
Tasks:  61 total,   1 running,  60 sleeping,   0 stopped,   0 zombie
Cpu(s): 77.3%us, 22.7%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,
Mem:   2061712k total,   245248k used,  1816464k free,       12k buffe
Swap:        0k total,        0k used,        0k free,   141836k cache

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 5566 root      20   0  646m  18m 7740 S   90  0.9   0:05.27 java
 5567 root      20   0  646m  17m 7644 S   90  0.9   0:05.25 java
 5592 root      20   0 36488  11m 9296 S    9  0.6   0:00.55 screensho
 2356 root      20   0 34112  12m 5228 S    7  0.6   3:50.63 X
 2374 root      20   0 10688 5192 3772 S    2  0.3   1:36.01 e16
 3635 root      20   0  114m  42m  16m S    1  2.1   0:15.34 gimp
 2482 root      20   0  5032 1308 1084 S    0  0.1   0:17.75 E-Load.ep
 2522 root      20   0 11752 5560 2292 S    0  0.3   0:05.38 xterm
 5541 root      20   0  3416  852  736 S    0  0.0   0:00.03 X10Launch
    1 root      20   0  2072  564  500 S    0  0.0   0:01.57 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S    0  0.0   0:00.02 ksoftirq
    4 root      20   0     0    0    0 S    0  0.0   0:00.36 kworker/0
    6 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration
    7 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration
    8 root      20   0     0    0    0 S    0  0.0   0:00.48 kworker/1
    9 root      20   0     0    0    0 S    0  0.0   0:00.01 ksoftirq
```

Figura 4.34 Salida del comando top durante la ejecución del cálculo del número PI por el algoritmo de Montecarlo en 2 lugares y 1 hilo, en Atom N270

Suma de 2 Matrices cuadradas

La suma de matrices cuadradas es un ejemplo muy utilizado,ya que no existe ninguna dependencia de datos ni control, porque cada operación es independiente. Aunque se utiliza una barrera explícita para asegurar que todos los procesos terminen su ejecución.

La suma de 2 matrices se realiza sumando elementos de la misma posición en la matriz, como se puede observar en la figura 4.35.

$$A = \begin{pmatrix} 2 & 0 & 1 \\ 3 & 0 & 0 \\ 5 & 1 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

$$A + B = \begin{pmatrix} 2+1 & 0+0 & 1+1 \\ 3+1 & 0+2 & 0+1 \\ 5+1 & 1+1 & 1+0 \end{pmatrix} = \begin{pmatrix} 3 & 0 & 2 \\ 4 & 2 & 1 \\ 6 & 2 & 1 \end{pmatrix}$$

Figura 4.35 Suma de 2 matrices cuadradas

En el siguiente ejemplo en X10 (véase código 4.5), primero se crean 2 matrices de dimensión 10x10 con elementos aleatorios,y posteriormente se realiza la suma.

Código 4.5

```
import x10.io.Console;
import x10.array.*;
import x10.util.Random;

public class suma_arreglos {


 public static def main(args:Rail[String]) {

val a=new Array_2[Long](10,10);
val b=new Array_2[Long](10,10);
val c=new Array_2[Long](10,10);
var i:Long;
var j:Long;
val r = new Random(); //número aleatorio


	//llenamos arreglos
	for(i=0;i<10;i++)
	{

		for(j=0;j<10;j++)
		{
```

```
                a(i,j)=r.nextLong();
                b(i,j)=r.nextLong();
                Console.OUT.println("a("+i+","+j+")=" + a(i,j));
                Console.OUT.println("b("+i+","+j+")=" + b(i,j));
            }

    }

    //procesar la suma en paralelo
    for(i=0;i<10;i++)
    {
        finish{                                     //barrera explícita
            async{                                      //paralelizar
                for(j=0;j<10;j++)
                {
                c(i,j)=a(i,j)+ b(i,j);
                }
            }
        }
    }

    //imprimir resultado
    for(i=0;i<10;i++)
    {

        for(j=0;j<10;j++)
        {
        Console.OUT.println("c("+i+","+j+")=" + c(i,j));
        }
    }


}
}
```

Resultados del cálculo

	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
1 Hilo	0.400	2.239	0.335	0.601
2 Hilos	0.398	2.259	0.332	0.595
3 Hilos	0.391	2.272	0.331	0.595
4 Hilos	0.417	2.247	0.337	0.597
5 Hilos	0.385	2.294	0.338	0.601
6 Hilos	0.391	2.283	0.338	0.598
7 Hilos	0.405	2.271	0.335	0.605
8 Hilos	0.408	2.264	0.338	0.607

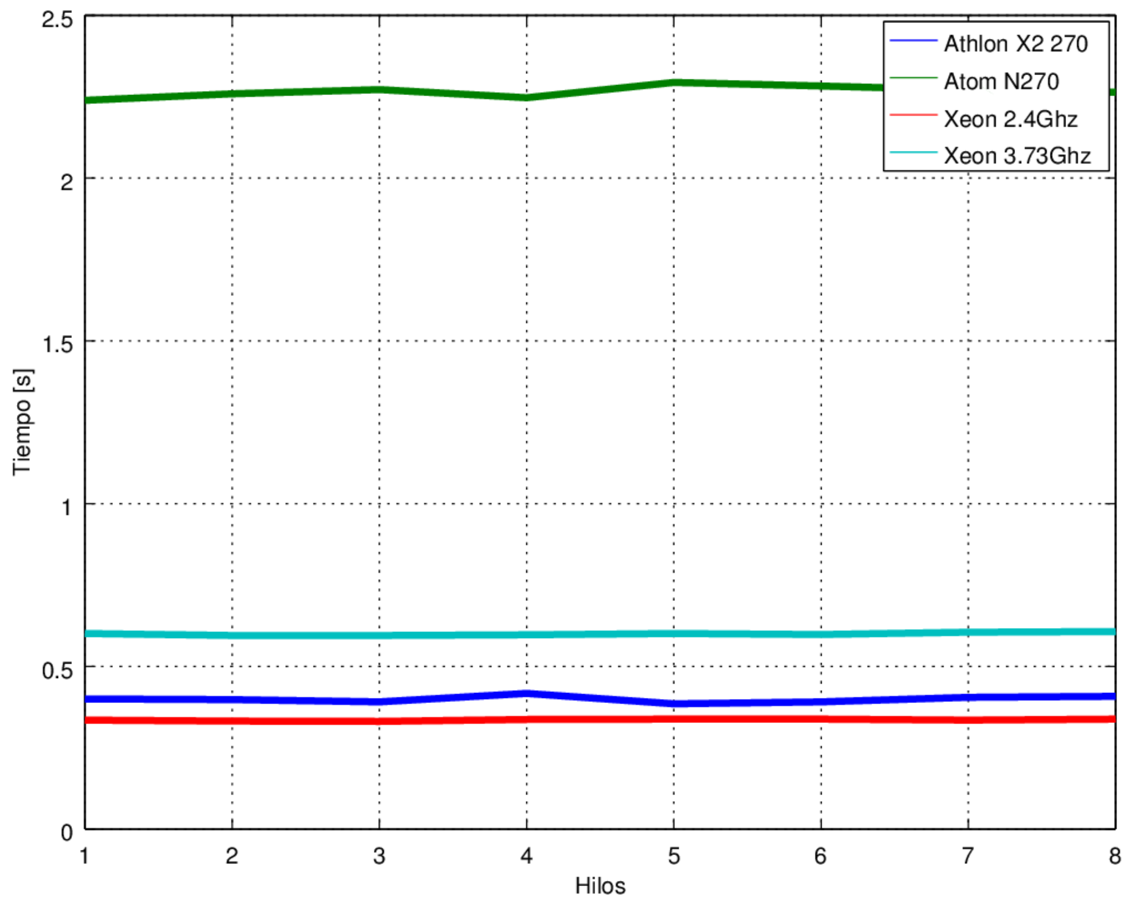


Figura 4.36 Gráfica del tiempo de ejecución del cálculo de la suma de 2 matrices cuadradas en múltiples hilos

	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
1 Hilo	1.000	1.000	1.000	1.000
2 Hilos	1.00503	0.99115	1.00904	1.01008
3 Hilos	1.02302	0.98548	1.01208	1.01008
4 Hilos	0.95923	0.99644	0.99407	1.00670
5 Hilos	1.03896	0.97602	0.99112	1.0000
6 Hilos	1.02302	0.98073	0.99112	1.00502
7 Hilos	0.98765	0.98591	1.0000	0.99339
8 Hilos	0.98039	0.98896	0.99112	0.99012

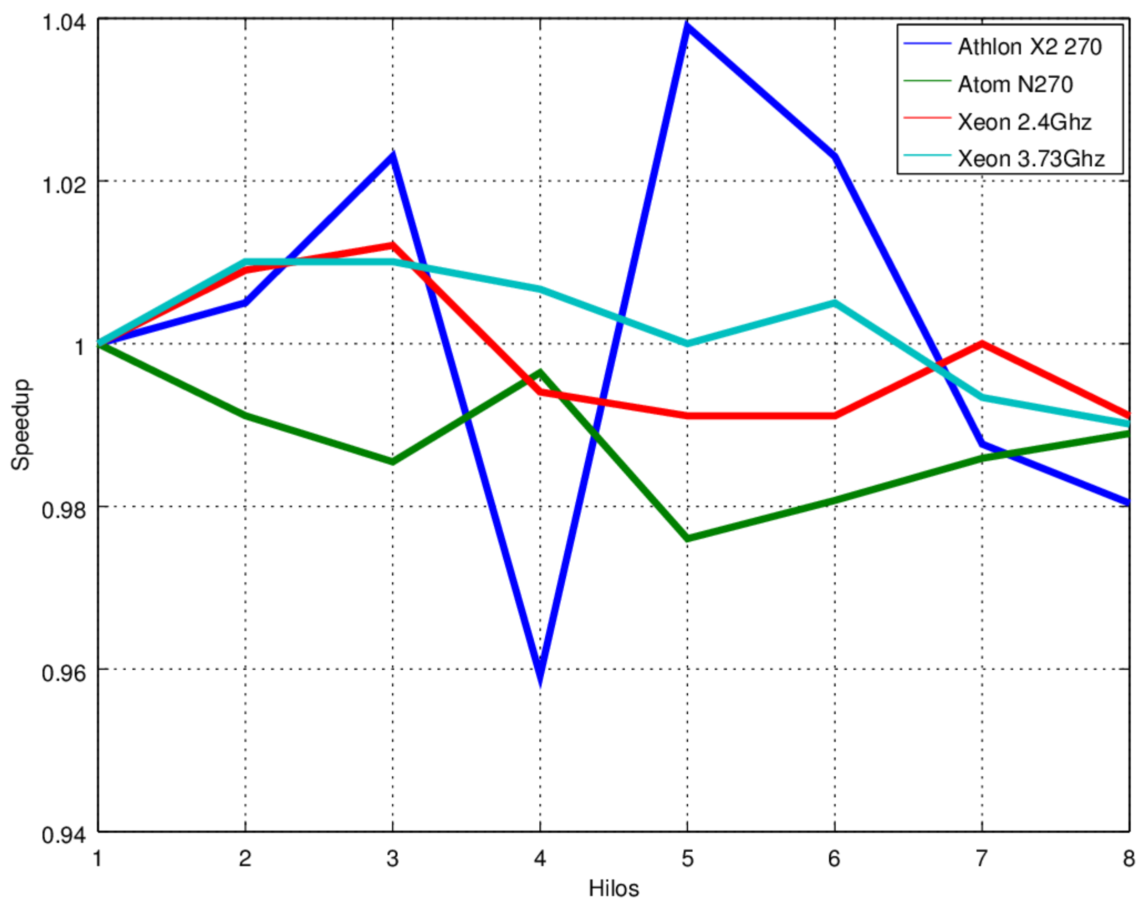


Figura 4.37 Gráfica del Speedup del cálculo de la suma de 2 matrices cuadradas en múltiples hilos

	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
1 Lugar	0.400	2.239	0.335	0.601
2 Lugares	0.726	4.959	0.562	0.961
3 Lugares	0.903	6.253	0.580	0.996
4 Lugares	1.254	7.441	0.634	1.076
5 Lugares	1.426	8.139	0.771	1.214
6 Lugares	1.823	9.928	0.867	1.393
7 Lugares	1.915	10.539	0.912	1.437
8 Lugares	2.211	12.518	1.073	1.488

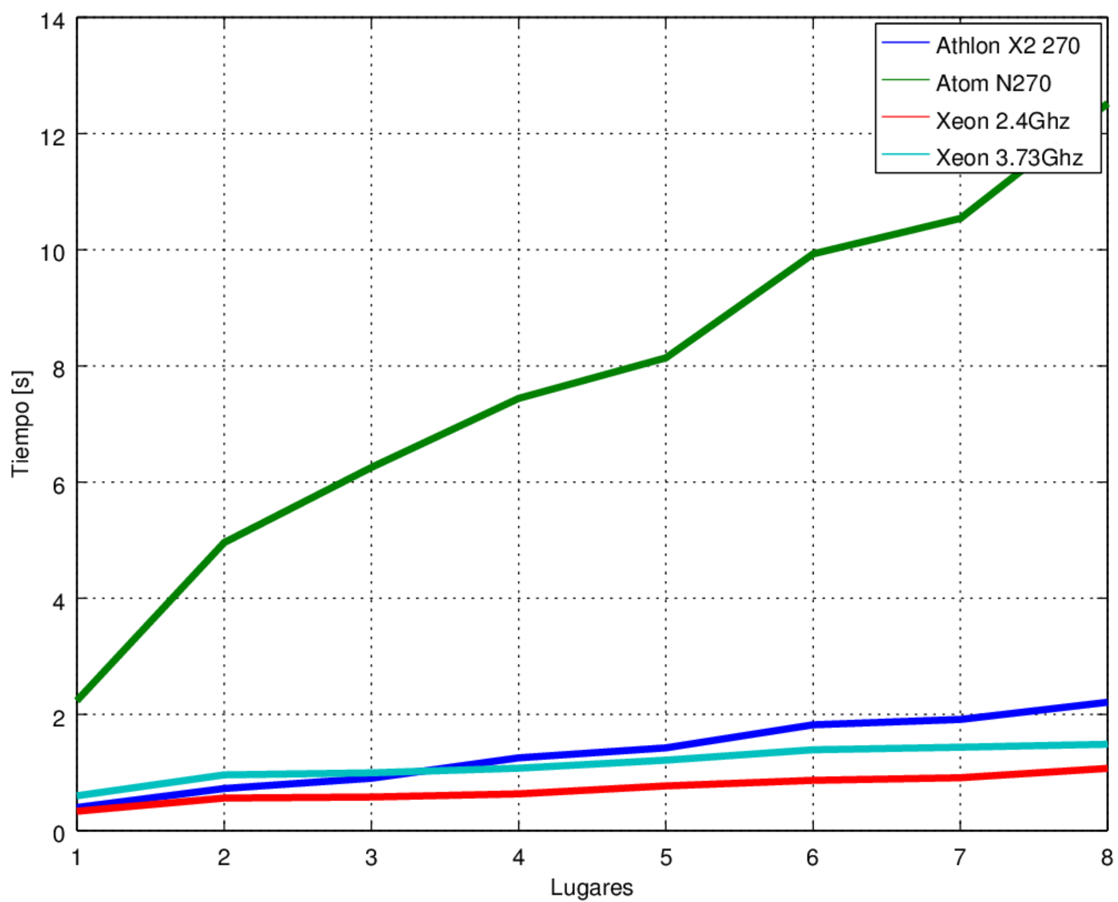


Figura 4.38 Gráfica del tiempo de ejecución del cálculo de la suma de 2 matrices cuadradas en múltiples lugares

	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
1 Lugar	1.000	1.000	1.000	1.000
2 Lugares	0.55096	0.45150	0.59609	0.62539
3 Lugares	0.44297	0.35807	0.57759	0.60341
4 Lugares	0.31898	0.30090	0.52839	0.55855
5 Lugares	0.28050	0.27510	0.43450	0.49506
6 Lugares	0.21942	0.22552	0.38639	0.43144
7 Lugares	0.20888	0.21245	0.36732	0.41823
8 Lugares	0.18091	0.17889	0.31221	0.40390

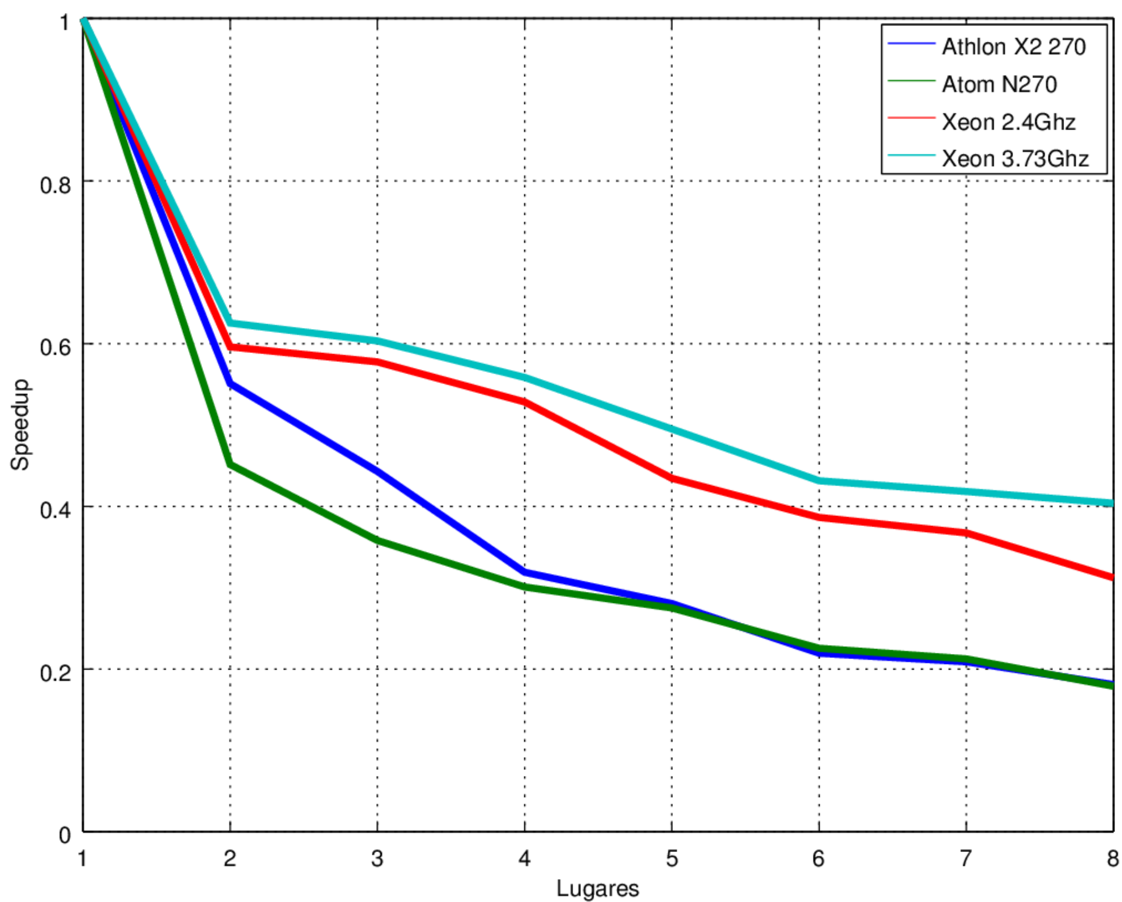


Figura 4.39 Gráfica del Speedup del cálculo de la suma de 2 matrices cuadradas en múltiples lugares

Muestras	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
(1) 1 Lugar y 1 hilo	0.400	2.239	0.335	0.601
(2) 1 Lugar y 2 hilos	0.398	2.261	0.336	0.593
(3) 2 Lugares y 1 hilo	0.706	4.679	0.561	0.963
(4) 2 Lugares y 2 hilos	0.721	4.906	0.564	0.955
(5) 2 Lugares y 3 hilos	0.726	5.370	0.567	0.957
(6) 2 Lugares y 4 hilos	0.730	4.937	0.564	0.961
(7) 3 Lugares y 1 hilo	1.012	6.081	0.591	1.018
(8) 3 Lugares y 2 hilos	0.999	6.542	0.594	1.003
(9) 3 Lugares y 3 hilos	0.928	6.301	0.609	1.009
(10) 4 Lugares y 1 hilo	1.155	7.109	0.629	1.059
(11) 4 Lugares y 2 hilos	1.332	7.699	0.637	1.050

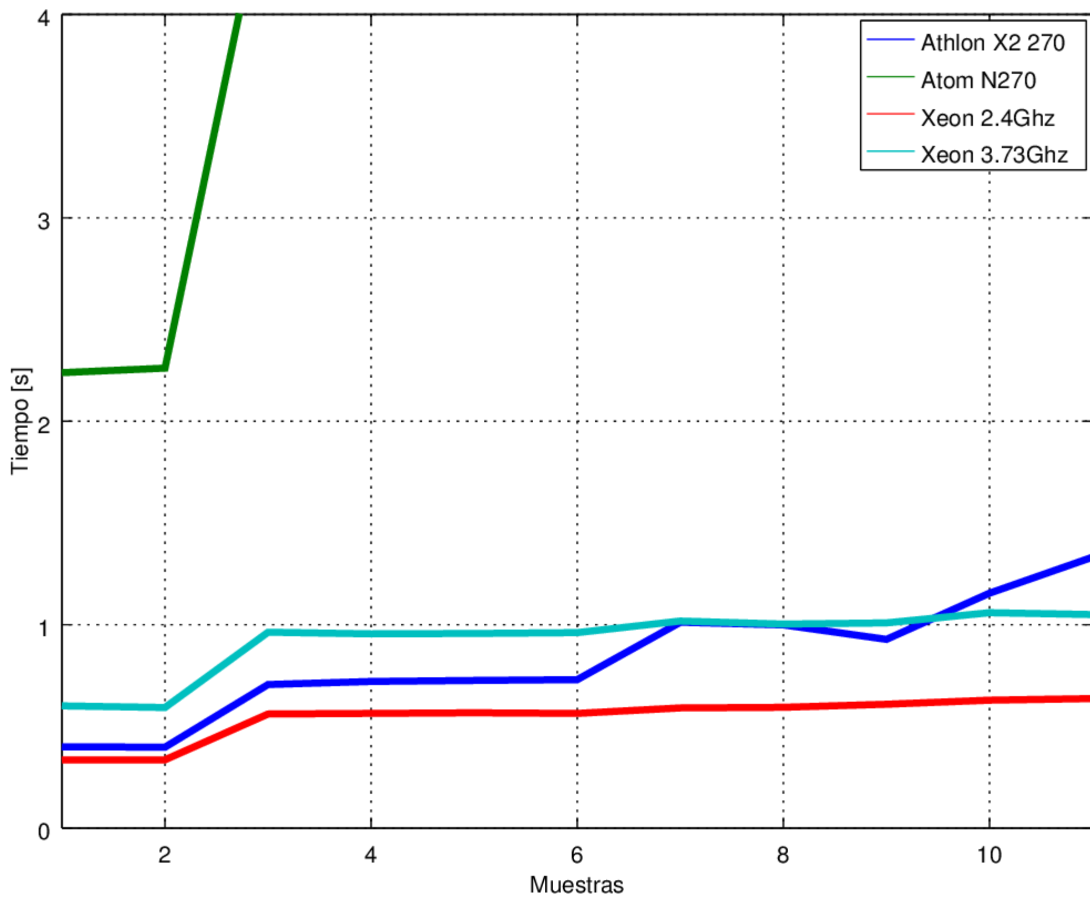


Figura 4.40 Gráfica del tiempo de ejecución del cálculo de la suma de 2 matrices cuadradas en múltiples hilos y lugares

Muestras	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
(1) 1 Lugar y 1 hilo	1.000	1.000	1.000	1.000
(2) 1 Lugar y 2 hilos	1.00503	0.99027	0.99702	1.01349
(3) 2 Lugares y 1 hilo	0.56657	0.47852	0.59715	0.62409
(4) 2 Lugares y 2 hilos	0.55479	0.45638	0.59397	0.62932
(5) 2 Lugares y 3 hilos	0.55096	0.41695	0.59083	0.62800
(6) 2 Lugares y 4 hilos	0.54795	0.45351	0.59397	0.62539
(7) 3 Lugares y 1 hilo	0.39526	0.36820	0.56684	0.59037
(8) 3 Lugares y 2 hilos	0.40040	0.34225	0.56397	0.59920
(9) 3 Lugares y 3 hilos	0.43103	0.35534	0.55008	0.59564
(10) 4 Lugares y 1 hilo	0.34632	0.31495	0.53259	0.56752
(11) 4 Lugares y 2 hilos	0.30030	0.29082	0.52590	0.57238

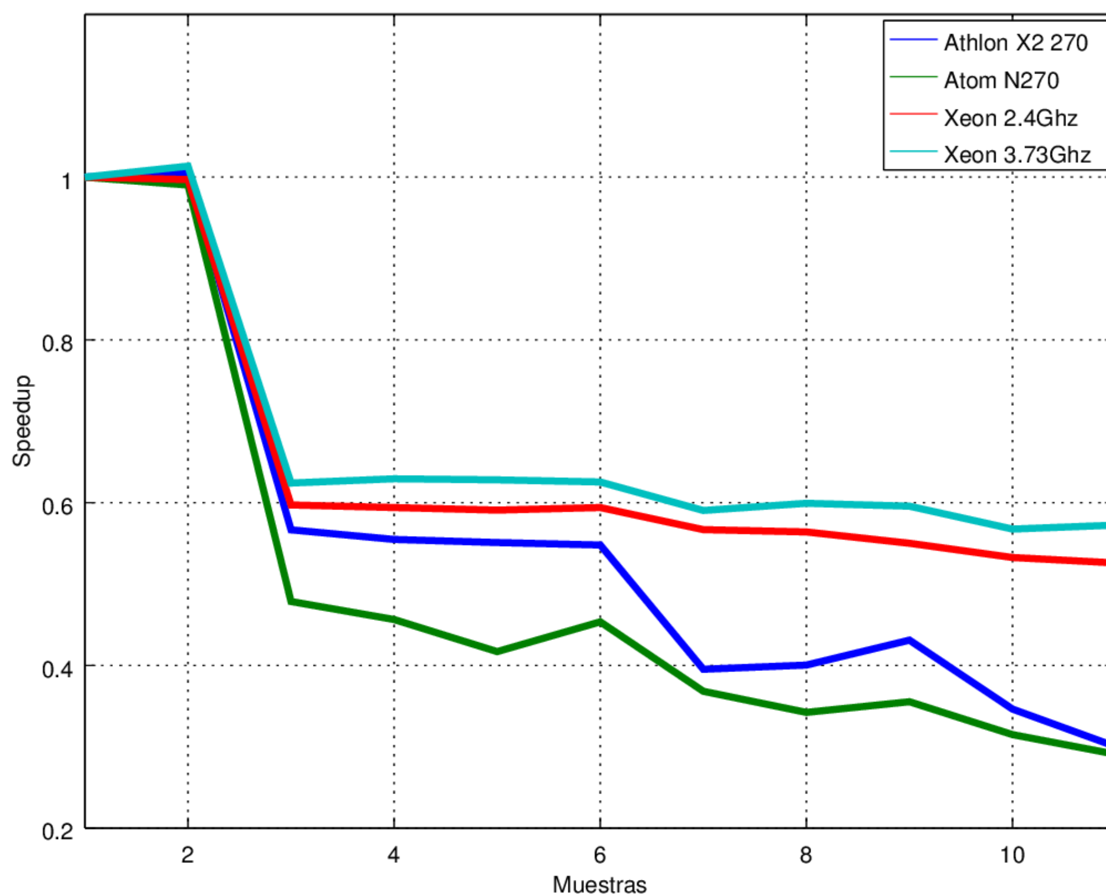


Figura 4.41 Gráfica del Speedup del cálculo de la suma de 2 matrices cuadradas en múltiples hilos y lugares

```
xterm
c(8,1)=5839203804852794241
c(8,2)=-8422232521828626657
c(8,3)=-3847794324029435169
c(8,4)=1635733028421411197
c(8,5)=-2220682719112388O3
c(8,6)=1061157356516183777
c(8,7)=5263416525644626949
c(8,8)=-7435433672580062244
c(8,9)=1595733265438024813
c(9,0)=-5020336327834857705
c(9,1)=8341926020283388907
c(9,2)=3584402088733433783
c(9,3)=-2736459563423702110
c(9,4)=-8538614197565551076
c(9,5)=214509134954911227
c(9,6)=8327885180237926398
c(9,7)=-3643694831886996393
c(9,8)=4145058826557759675
c(9,9)=6240987168299747376

real    0m4,959s
user    0m3,968s
sys     0m0,464s
bash-4,2#
```

Figura 4.42 Resultado de la ejecución del cálculo de la suma de 2 matrices cuadradas en 2 lugares, en Atom N270

```
xterm
top - 00:29:50 up  2:41,  2 users,  load average: 2,28, 1,64, 1,73
Tasks:  73 total,   1 running,  72 sleeping,   0 stopped,   0 zombie
Cpu(s): 54,2%us, 11,3%sy,  0,0%ni, 34,3%id,  0,0%wa,  0,0%hi,  0,2%si,  (
Mem:   2061712k total,   278204k used,  1783508k free,       12k buffers
Swap:        0k total,        0k used,        0k free,   143664k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 7267 root      20   0  644m  11m 6136 S   13  0,6   0:00,40 java
 7270 root      20   0  644m  11m 6136 S   13  0,6   0:00,40 java
 7273 root      20   0  644m  11m 6136 S   13  0,6   0:00,40 java
 7278 root      20   0  644m  11m 6136 S   13  0,6   0:00,40 java
 7281 root      20   0  644m  11m 6136 S   13  0,6   0:00,40 java
 7276 root      20   0  644m  11m 6136 S   13  0,6   0:00,39 java
 7277 root      20   0  644m  11m 6136 S   13  0,6   0:00,39 java
 7279 root      20   0  644m  11m 6136 S   13  0,6   0:00,39 java
 2356 root      20   0 34056  12m 5244 S    6  0,6   4:33,00 X
 2374 root      20   0 10684 5192 3772 S    2  0,3   1:52,19 e16
 7240 root      20   0 36488  11m 9296 S    1  0,6   0:00,43 screenshot
 7241 root      20   0  3416  856  736 S    1  0,0   0:00,02 X10Launcher
 2482 root      20   0  5032 1308 1084 S    0  0,1   0:20,58 E-Load.epple
 3635 root      20   0  119m  46m  17m S    0  2,3   0:31,75 gimp
 6788 root      20   0  2636  996  824 R    0  0,0   0:00,20 top
    1 root      20   0  2072  564  500 S    0  0,0   0:01,59 init
    2 root      20   0     0    0    0 S    0  0,0   0:00,00 kthreadd
```

Figura 4.43 Salida del comando top durante la ejecución del cálculo de la suma de 2 matrices cuadradas en 8 lugares, en Atom N270

Multiplicación de 2 Matrices Cuadradas

La multiplicación de 2 matrices cuadradas, se realiza multiplicando cada elemento de una fila por cada elemento de una columna y sumando los resultados, en la siguiente figura 4.44. se muestra la multiplicación de 2 matrices de 3x3.

$$A \cdot B = \begin{pmatrix} 2 & 0 & 1 \\ 3 & 0 & 0 \\ 5 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 0 \end{pmatrix} =$$

$$= \begin{pmatrix} 2\cdot 1+0\cdot 1+1\cdot 1 & 2\cdot 0+0\cdot 2+1\cdot 1 & 2\cdot 1+0\cdot 1+1\cdot 0 \\ 3\cdot 1+0\cdot 1+0\cdot 1 & 3\cdot 0+0\cdot 2+0\cdot 1 & 3\cdot 1+0\cdot 1+0\cdot 0 \\ 5\cdot 1+1\cdot 1+1\cdot 1 & 5\cdot 0+1\cdot 2+1\cdot 1 & 5\cdot 1+1\cdot 1+1\cdot 0 \end{pmatrix} =$$

$$= \begin{pmatrix} 3 & 1 & 2 \\ 3 & 0 & 3 \\ 7 & 3 & 6 \end{pmatrix}$$

Figura 4.44 Multiplicación de 2 matrices cuadradas

En el siguiente código 4.6 primero se crean 2 matrices de dimensión 11x11 con elementos aleatorios,y posteriormente se realiza la multiplicación.

Código 4.6

```
import x10.io.Console;
import x10.array.*;
import x10.util.Random;

public class multiplicacion_arreglos {


 public static def main(args:Rail[String]) {

val a=new Array_2[Long](11,11);
val b=new Array_2[Long](11,11);
val c=new Array_2[Long](11,11);
var i:Long;
var j:Long;
var k:Long;
val r = new Random(); //número aleatorio


        //llenamos arreglos
        for(i=0;i<11;i++)
        {
```

```
            for(j=0;j<11;j++)
            {
                  a(i,j)=r.nextLong();
                  b(i,j)=r.nextLong();
                  Console.OUT.println("a("+i+","+j+")=" + a(i,j));
                  Console.OUT.println("b("+i+","+j+")=" + b(i,j));
            }

      }

      //procesar la suma en paralelo
      for(i=0;i<11;i++)
      {
        finish{                                    //barrera explícita
                  async{


                        for(j=0;j<11;j++)
                        {
                              for(k=0;k<11;k++)
                              {
                             c(i,j)+=a(i,k)*b(k,j);
                               Console.OUT.println("c("+i+","+j+")=a( "+i+","+k+")*b(" + k + "," + j
+")"    );

                              }
                        }
                      }
              }
      }

      //imprimir resultado

   for(i=0;i<11;i++)
      {

            for(j=0;j<11;j++)
            {
            Console.OUT.println("c("+i+","+j+")=" + c(i,j));
            }
      }


}
}
```

De nueva cuenta el algoritmo es Serial-Paralelo, tanto la carga de números aleatorios y la impresión se realizan de manera serial, y el cálculo de forma paralela.

El algoritmo contiene dependencia de datos, ya que los diferentes procesos necesitan leer las mismas posiciones de alguna de las 2 matrices. Para resolver esto, el lenguaje realiza copias de las variables y las migra hacia el procesador que requiere la información.

En este código se paraleliza fila por fila. Ya que los datos son números enteros, y si se realizara la paralelización elemento a elemento, se tardaría más la ejecución; debido al tiempo de migración de los datos a los diferentes procesadores.

Resultados del cálculo

	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
1 Hilo	0.457	2.817	0.398	0.700
2 Hilos	0.466	2.815	0.396	0.690
3 Hilos	0.487	2.887	0.400	0.691
4 Hilos	0.462	2.891	0.399	0.694
5 Hilos	0.485	2.904	0.400	0.683
6 Hilos	0.483	2.862	0.403	0.694
7 Hilos	0.480	2.883	0.401	0.692
8 Hilos	0.461	2.863	0.404	0.695

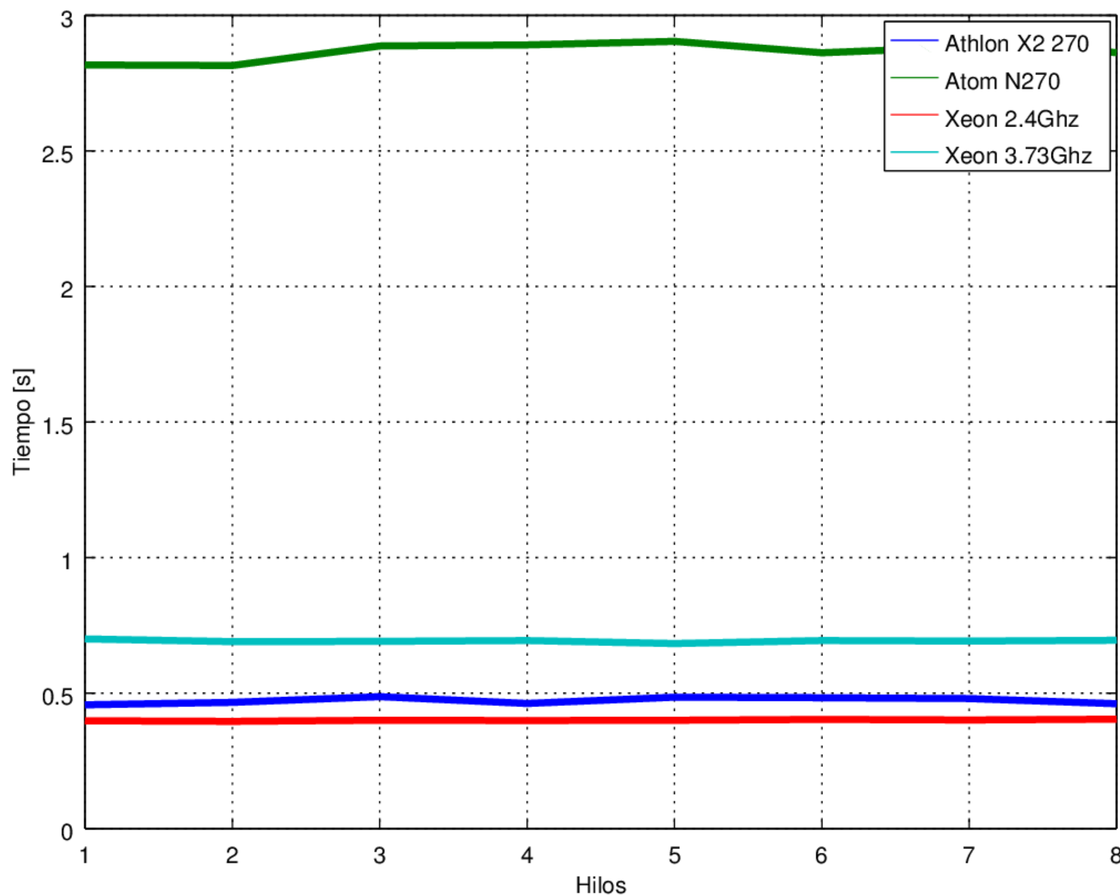


Figura 4.45 Gráfica del tiempo de ejecución del cálculo de la multiplicación de 2 matrices cuadradas en múltiples hilos

	Speedup en AMD	Speedup en Intel	Speedup en Intel	Speedup en Intel

	Athlon x2 270	Atom N270 HT	Xeon 2.4Ghz	Xeon 3.73Ghz
1 Hilo	1.000	1.000	1.000	1.000
2 Hilos	0.98069	1.0071	1.00505	1.0145
3 Hilos	0.93840	0.97575	0.99500	1.0130
4 Hilos	0.98918	0.97440	0.99749	1.0086
5 Hilos	0.94227	0.97004	0.99500	1.0249
6 Hilos	0.94617	0.98428	0.98759	1.0086
7 Hilos	0.95208	0.97711	0.99252	1.0116
8 Hilos	0.99132	0.98393	0.98515	1.0072

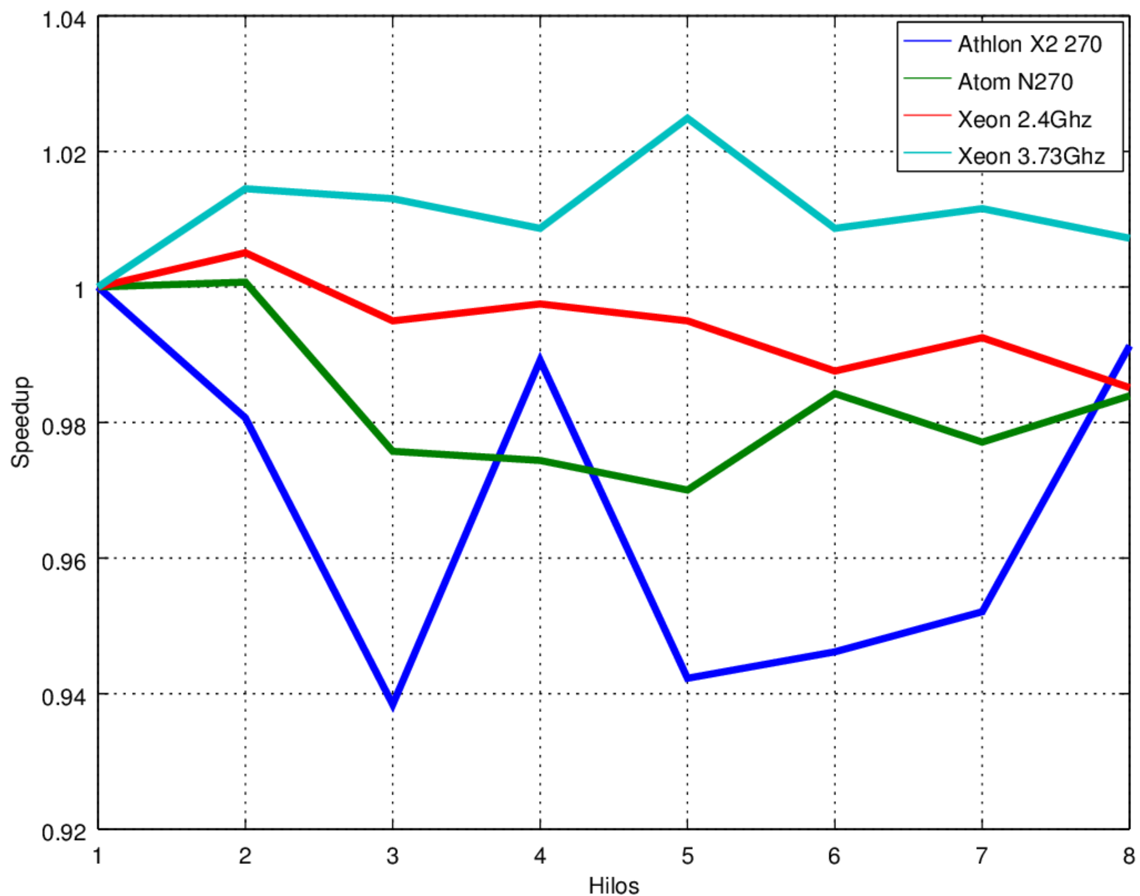


Figura 4.46 Gráfica del Speedup del cálculo de la multiplicación de 2 matrices cuadradas en múltiples hilos

	Tiempo en	Tiempo en Intel	Tiempo en Intel	Tiempo en Intel

	AMD Athlon x2 270	Atom N270 HT	Xeon 2.4Ghz	Xeon 3.73Ghz
1 Lugar	0.457	2.817	0.398	0.700
2 Lugares	0.791	6.145	0.613	1.066
3 Lugares	0.984	8.303	0.654	1.114
4 Lugares	1.350	8.793	0.672	1.197
5 Lugares	1.474	10.538	0.803	1.308
6 Lugares	1.706	12.252	0.985	1.503
7 Lugares	2.004	13.517	1.001	1.543
8 Lugares	2.239	14.435	1.108	1.708

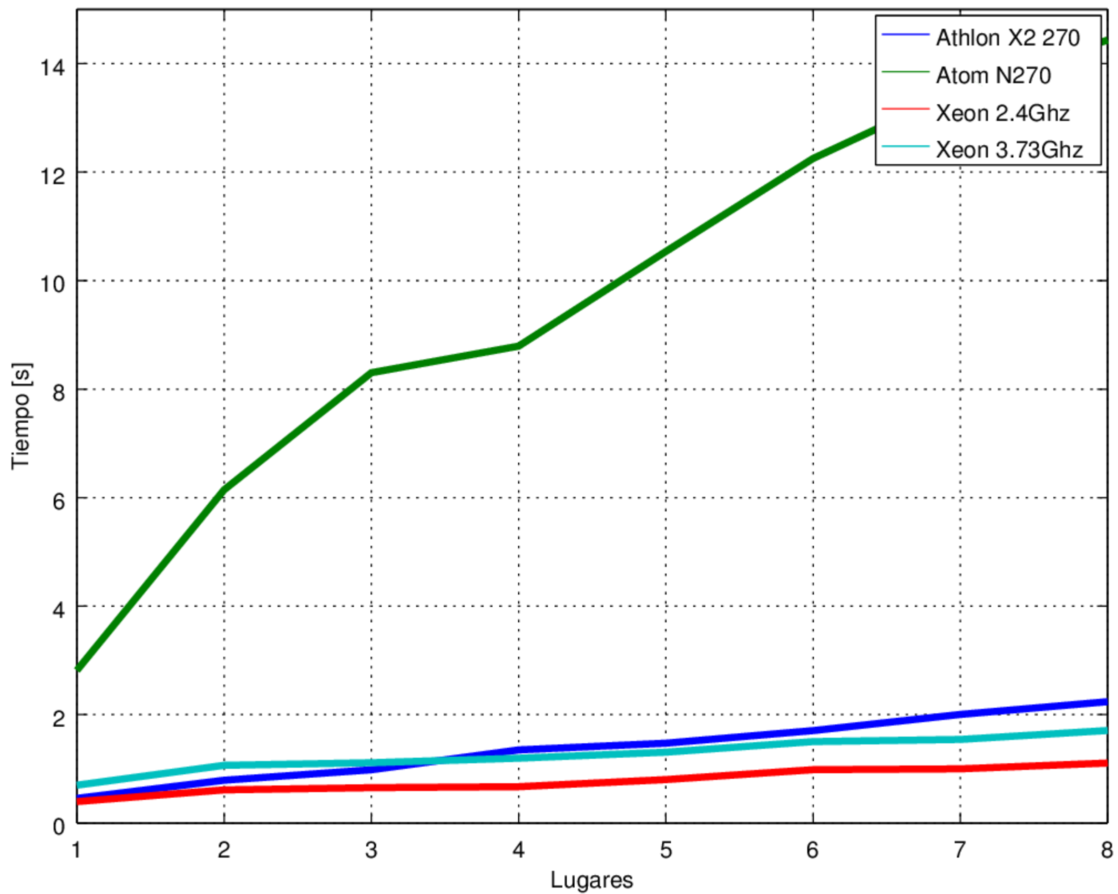


Figura 4.47 Gráfica del tiempo de ejecución del cálculo de la multiplicación de 2 matrices cuadradas en múltiples lugares

	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
1 Lugar	1.000	1.000	1.000	1.000
2 Lugares	0.57775	0.45842	0.64927	0.65666
3 Lugares	0.46443	0.33927	0.60856	0.62837
4 Lugares	0.33852	0.32037	0.59226	0.58480
5 Lugares	0.31004	0.26732	0.49564	0.53517
6 Lugares	0.26788	0.2292	0.40406	0.46574
7 Lugares	0.22804	0.20840	0.39760	0.45366
8 Lugares	0.20411	0.19515	0.35921	0.40984

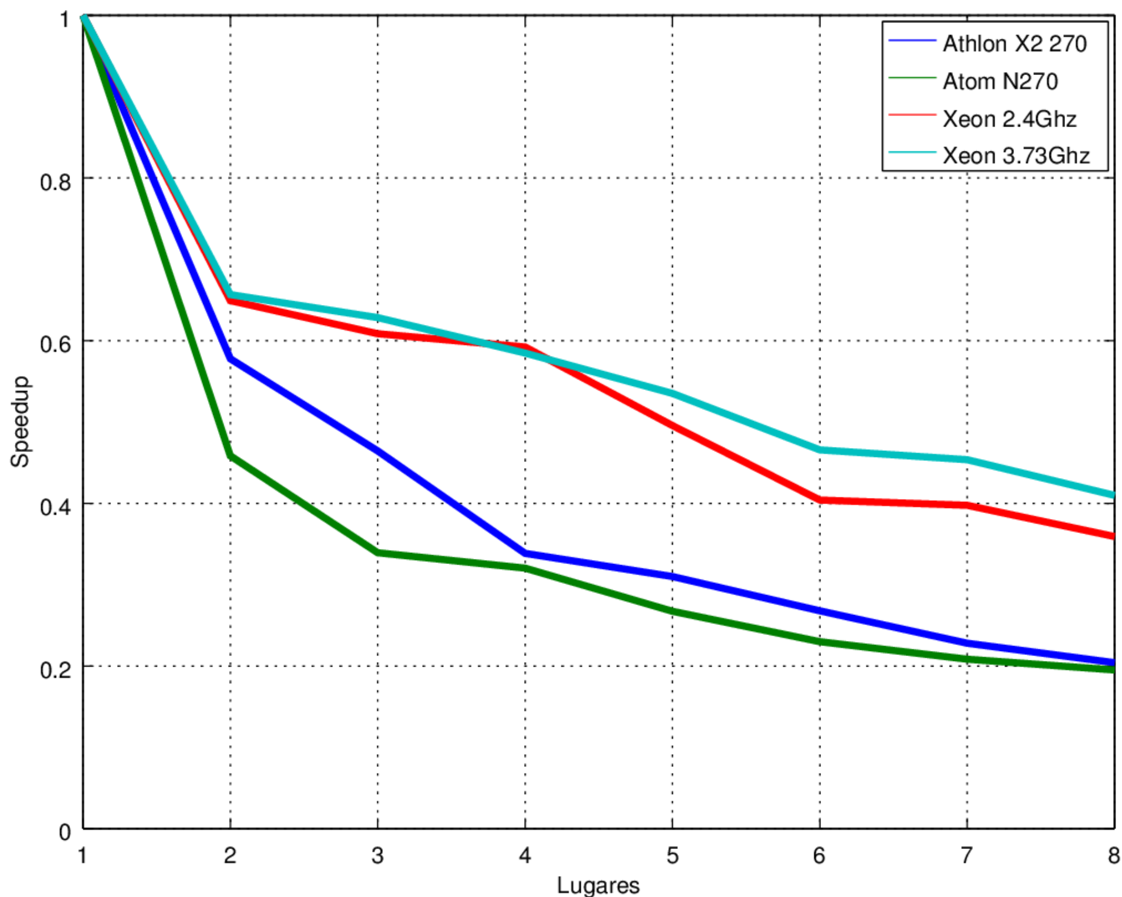


Figura 4.48 Gráfica del Speedup del cálculo de la multiplicación de 2 matrices cuadradas en múltiples lugares

Muestras	Tiempo en AMD Athlon x2 270	Tiempo en Intel Atom N270 HT	Tiempo en Intel Xeon 2.4Ghz	Tiempo en Intel Xeon 3.73Ghz
(1) 1 Lugar y 1 hilo	0.457	2.817	0.398	0.700
(2) 1 Lugar y 2 hilos	0.484	2.875	0.397	0.690
(3) 2 Lugares y 1 hilo	0.799	6.346	0.614	1.073
(4) 2 Lugares y 2 hilos	0.781	6.153	0.613	1.064
(5) 2 Lugares y 3 hilos	0.800	6.018	0.614	1.071
(6) 2 Lugares y 4 hilos	0.822	6.447	0.615	1.069
(7) 3 Lugares y 1 hilo	1.023	7.461	0.635	1.125
(8) 3 Lugares y 2 hilos	1.113	7.247	0.648	1.125
(9) 3 Lugares y 3 hilos	1.004	8.028	0.635	1.122
(10) 4 Lugares y 1 hilo	1.237	8.794	0.689	1.156
(11) 4 Lugares y 2 hilos	1.232	8.311	0.681	1.187

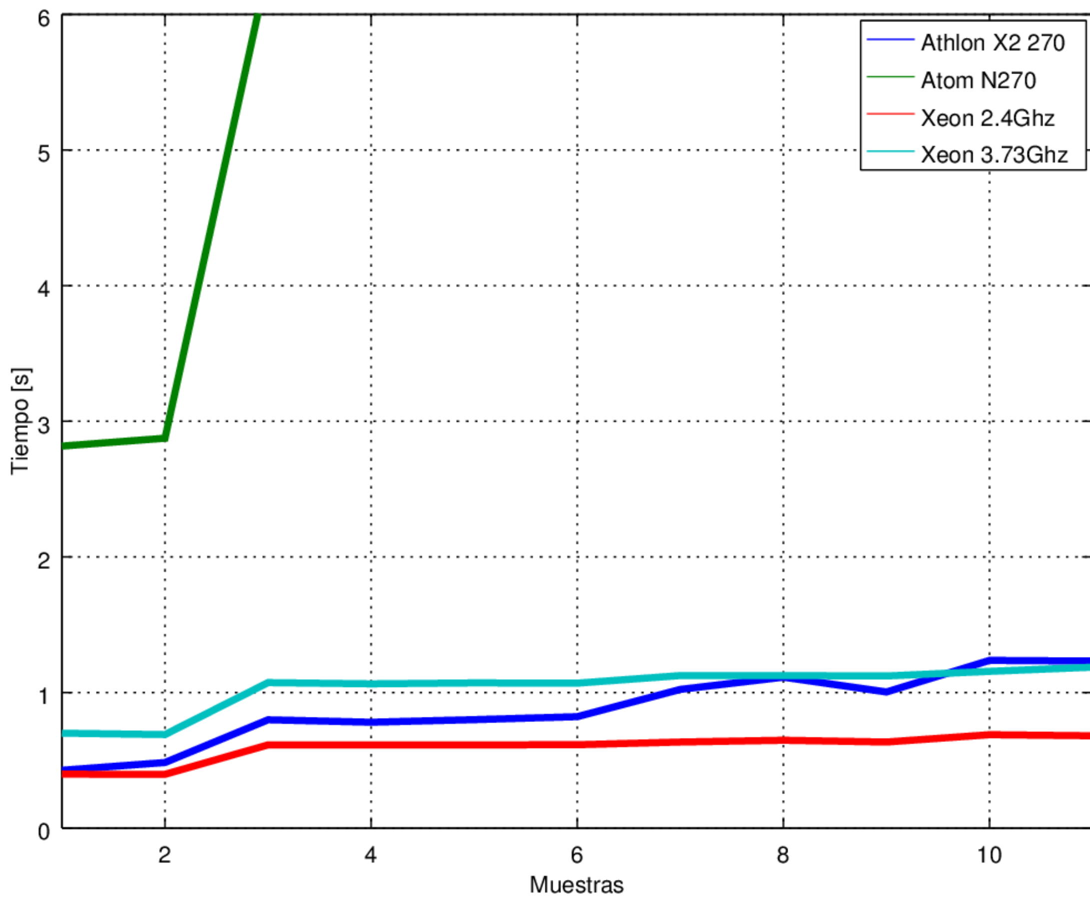


Figura 4.49 Gráfica del tiempo de ejecución del cálculo de la multiplicación de 2 matrices cuadradas en múltiples hilos y lugares

Muestras	Speedup en AMD Athlon x2 270	Speedup en Intel Atom N270 HT	Speedup en Intel Xeon 2.4Ghz	Speedup en Intel Xeon 3.73Ghz
(1) 1 Lugar y 1 hilo	1.000	1.000	1.000	1.000
(2) 1 Lugar y 2 hilos	0.94421	0.97983	1.00252	1.01449
(3) 2 Lugares y 1 hilo	0.57196	0.44390	0.64821	0.65238
(4) 2 Lugares y 2 hilos	0.58515	0.45783	0.64927	0.65789
(5) 2 Lugares y 3 hilos	0.57125	0.46810	0.64821	0.65359
(6) 2 Lugares y 4 hilos	0.55596	0.43695	0.64715	0.65482
(7) 3 Lugares y 1 hilo	0.44673	0.37756	0.62677	0.62222
(8) 3 Lugares y 2 hilos	0.41060	0.38871	0.61420	0.62222
(9) 3 Lugares y 3 hilos	0.45518	0.35090	0.62677	0.62389
(10) 4 Lugares y 1 hilo	0.36944	0.32033	0.57765	0.60554
(11) 4 Lugares y 2 hilos	0.37094	0.33895	0.58443	0.58972

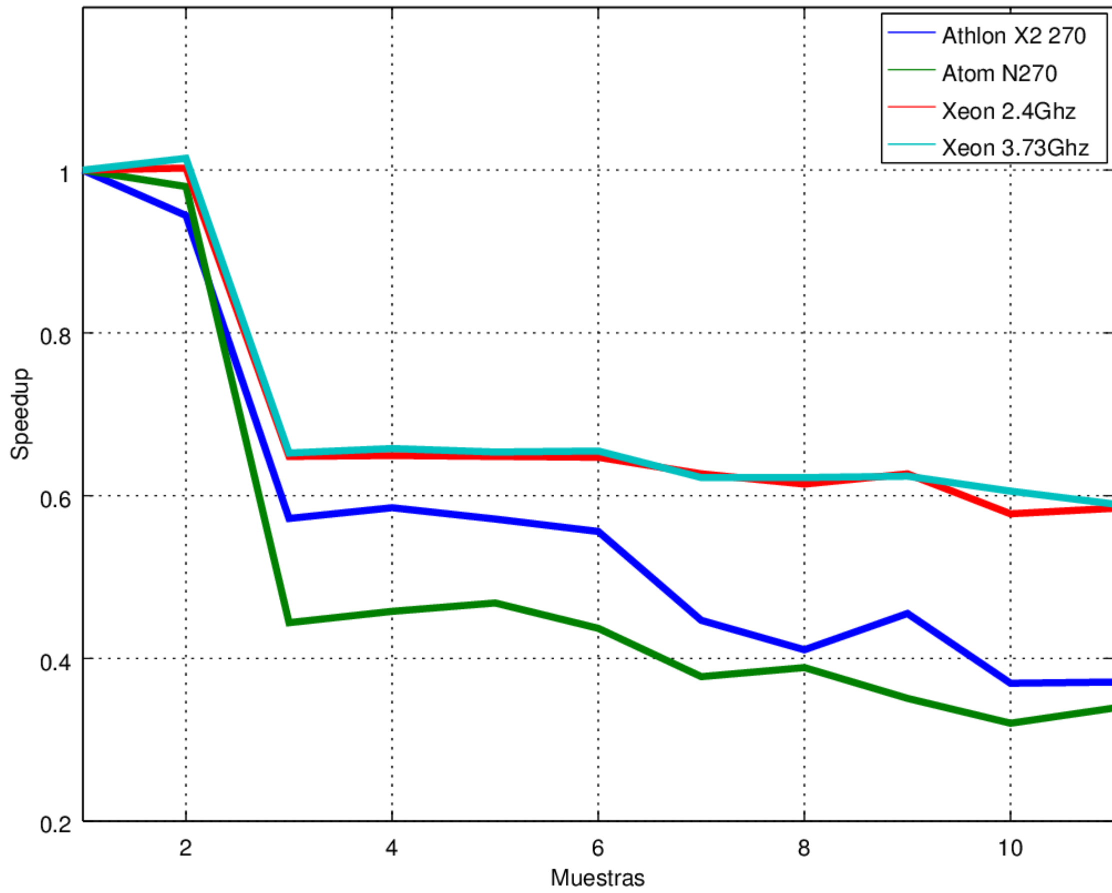


Figura 4.50 Gráfica del Speedup del cálculo de la multiplicación de 2 matrices cuadradas en múltiples hilos y lugares

```
xterm
c(9,3)=4460429773650981183
c(9,4)=2090158619598545285
c(9,5)=5716988885098282339
c(9,6)=7599905170340640541
c(9,7)=-5430301021396674223
c(9,8)=4820343623731031596
c(9,9)=6840674393462243873
c(9,10)=-1049857760209572740
c(10,0)=-6507156049165528490
c(10,1)=-9917267571070587 36
c(10,2)=-2553612049938937 65
c(10,3)=-7819265133299749396
c(10,4)=-2355097544371027464
c(10,5)=575954304769076 4737
c(10,6)=-6671374564907863766
c(10,7)=-7898851027868513294
c(10,8)=826095233001747 0345
c(10,9)=885262789523075 6508
c(10,10)=2527182367793318698

real    0m10.538s
user    0m12.485s
sys     0m2.680s
bash-4.2# █
```

Figura 4.51 Resultado de la ejecución del cálculo de la multiplicación de 2 matrices cuadradas en 5 lugares, en Atom N270

```
xterm
top - 00:37:04 up  2:48,  2 users,  load average: 0.68, 1.13, 1.49
Tasks:  67 total,   1 running,  66 sleeping,   0 stopped,   0 zombie
Cpu(s): 56.9%us,  8.5%sy,  0.0%ni, 34.4%id,  0.0%wa,  0.0%hi,  0.2%si,  0.(
Mem:   2061712k total,   279024k used,  1782688k free,       12k buffers
Swap:        0k total,        0k used,        0k free,   142620k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 8712 root      20   0  645m  16m 7392 S   28  0.8   0:00.99 java
 8715 root      20   0  646m  16m 7484 S   27  0.8   0:00.95 java
 8716 root      20   0  645m  16m 7440 S   25  0.8   0:00.90 java
 8718 root      20   0  645m  16m 7488 S   24  0.8   0:00.86 java
 8719 root      20   0  645m  16m 7440 S   22  0.8   0:00.84 java
 2356 root      20   0 33556  12m 5244 S    4  0.6   4:52.55 X
 2374 root      20   0 10684 5192 3772 S    2  0.3   1:57.68 e16
 2482 root      20   0  5032 1308 1084 S    0  0.1   0:21.51 E-Load.epplet
 6788 root      20   0  2636  996  824 R    0  0.0   0:01.24 top
 8686 root      20   0  3416  852  732 S    0  0.0   0:00.03 X10Launcher
    1 root      20   0  2072  564  500 S    0  0.0   0:01.59 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S    0  0.0   0:00.03 ksoftirqd/0
    4 root      20   0     0    0    0 S    0  0.0   0:00.60 kworker/0:0
    6 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    7 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
    8 root      20   0     0    0    0 S    0  0.0   0:00.90 kworker/1:0
```

Figura 4.52 Salida del comando top durante la ejecución del cálculo de la multiplicación de 2 matrices cuadradas en 5 lugares, en Atom N270

5)Conclusiones

A partir del inicio de mi estudio y desarrollo de esta Tesis, tuve la oportunidad de dar 3 talleres de introducción a la programación paralela utilizando X10, 2 de ellos a alumnos de la Universidad Distrital Francisco José de Caldas en Bogotá Colombia en el año 2013 y 2014,durante su semana tecnológica. Y uno a profesores y alumnos de la Universidad Autónoma de Tlaxcala durante el evento de su 35 aniversario de la Facultad de Ciencias Básicas, Ingeniería y Tecnología en el año 2013.

Durante el desarrollo de los talleres, la aceptación del lenguaje X10 fue muy buena y fácil de utilizar por sus similitudes al lenguaje Java. Sin embargo algunos conceptos no eran claros, tal vez porque la teoría muchas veces resulta tediosa si no se cuenta con ejemplos prácticos adecuados. Por esta razón se desarrollaron ejemplos sencillos que ilustran los conceptos y son fáciles de implementar en el lenguaje X10. El lenguaje tiene ventajas como son el no saber necesariamente en que arquitectura se ejecutará el programa, no utilizar llamadas o funciones complejas para distribuir los procesos y/o hilos de ejecución, ser orientado a objetos y tener una alta similitud a lenguajes ya conocidos.

En mi opinión X10 cumple con su objetivo de ser un lenguaje fácil de entender e implementar, para dar un acercamiento a la programación paralela. Sin embargo al ser un lenguaje de alto nivel y tener un nivel de abstracción similar al de Java, el rendimiento no es el óptimo de manera automática; claro que se puede utilizar el compilador X10c++ y optimizar el código de manera manual obteniendo un rendimiento casi al clásico C/MPI.
X10 permite al programador enfocarse en la resolución del problema y no en la arquitectura donde se ejecutará, además soporta utilizar otras herramientas y protocolos como MPI, CUDA, PAMI, DCMF y TCP/IP.

Otra ventaja si se decide compilar (X10C) para utilizar la ejecución por máquina virtual, es la de compilar una sola vez, compartir y ejecutar en diferentes arquitecturas.

En conclusión el objetivo de desarrollar un documento que ayude en la enseñanza y entendimiento de la programación paralela, utilizando el lenguaje de programación X10 mediante ejemplos prácticos y teóricos, se realizó con éxito. Claro el documento es introductorio y perfectible, ya que la tecnología cambia de manera muy rápida. Pero contiene la información necesaria para introducirse en la programación paralela.

6) Apéndices

Durante el desarrollo de los siguientes apéndices se utilizó la distribución GNU\Linux JarroNegro versión 3.0.0 . Sin embargo los procedimientos pueden ser ocupados en cualquier distribución GNU\Linux.

Para poder utilizar el entorno de desarrollo y X10 con todas sus funcionalidades, es necesario contar con la máquina virtual de Java JRE (Entorno de ejecución) o JDK (Entorno de desarrollo), para lo cual mostraremos los pagos a continuación

Primero se descarga el entorno de ejecución de Java desde la página principal y seleccionaremos el archivo con terminación .tar.gz , el cual no necesita ningún empaquetador como RPM, DPKG,etc.. (véase figura 6.1)

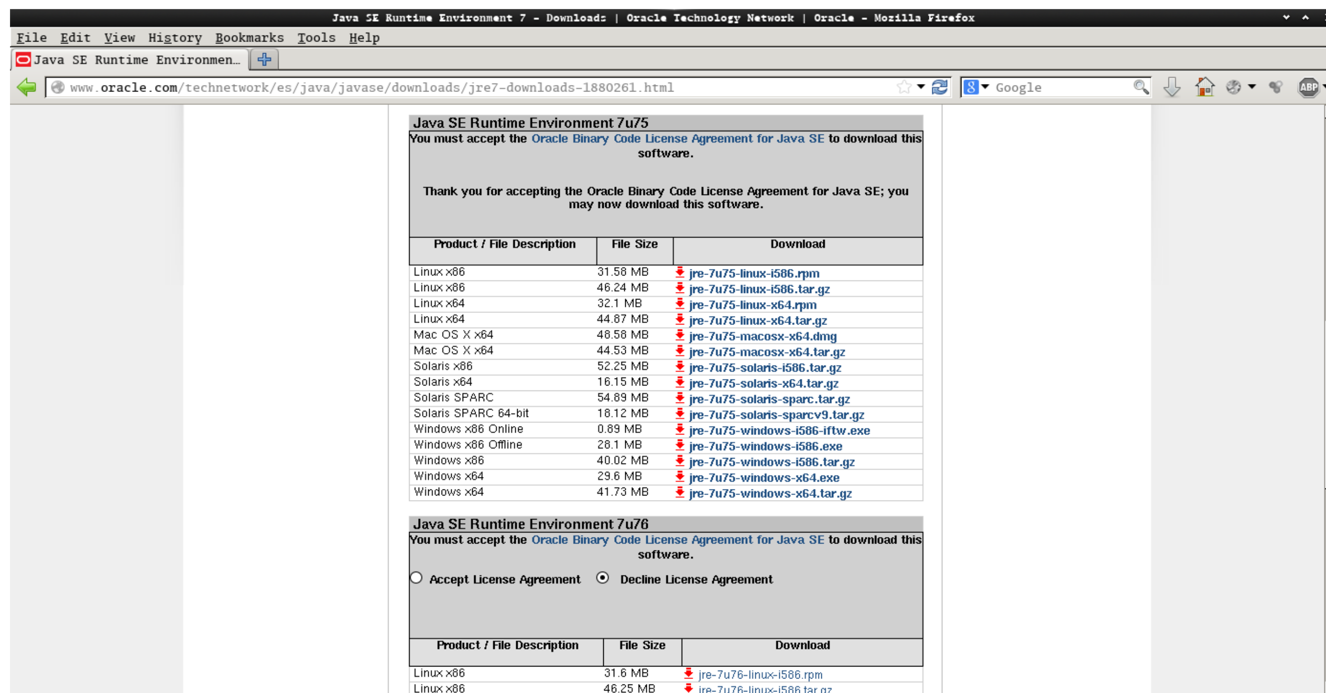


Figura 6.1 Página web de descarga de la máquina virtual de Java

Ya que lo tengamos descargado, es necesario descomprimirlo como se muestra en la siguiente figura 6.2.

Figura 6.2 Descomprimir la máquina virtual de Java

Al finalizar se creará un directorio el cual contendrá la máquina virtual de Java,a continuación lo listaremos como se aprecia en la figura 6.3.

Figura 6.3 Listado del directorio de la máquina virtual de Java

Para su instalación es necesario dar permisos de lectura y ejecución a todos los archivos. (véase figura 6.4)

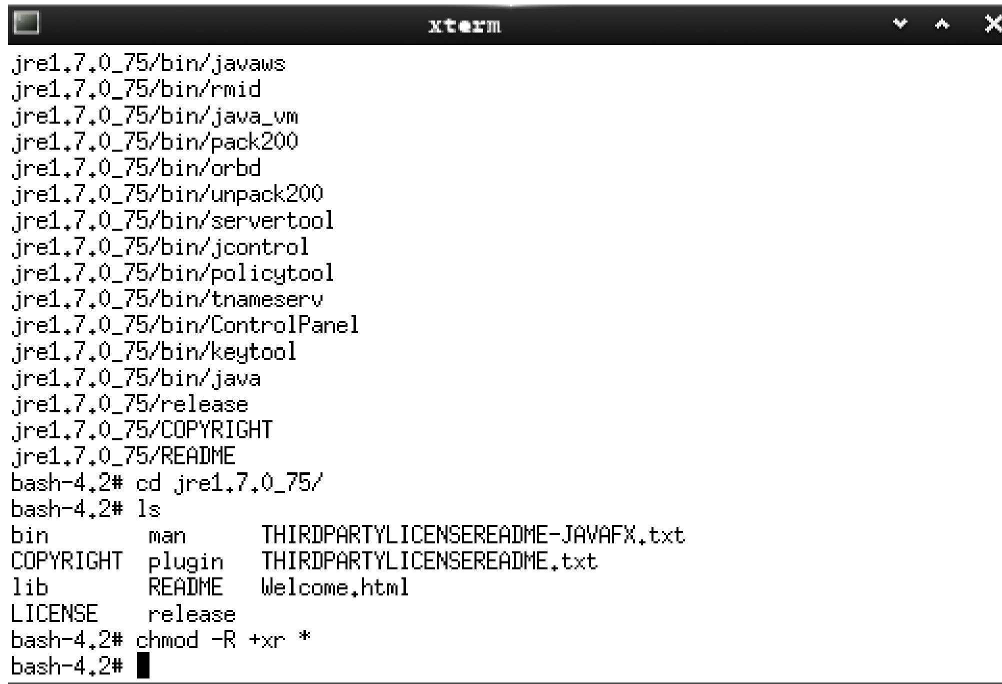


Figura 6.4 Conceder permisos de lectura y escritura al directorio de la máquina virtual de Java

Por último es necesario copiar los directorios *bin* y *lib* al directorio */usr* para que Java pueda ser ejecutado por cualquier usuario.(véase figura 6.5)

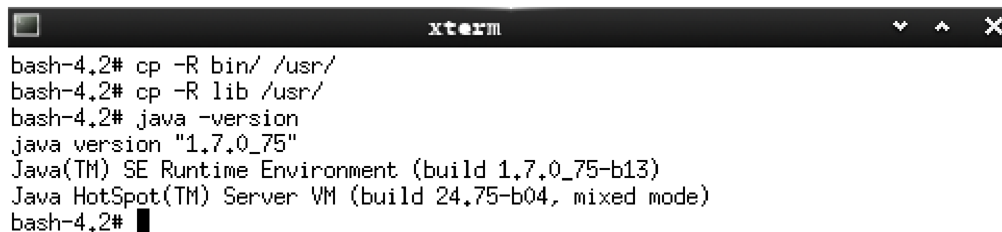


Figura 6.5 Instalación de la máquina virtual de Java

En caso de querer instalar el plugin de Java para el navegador web Firefox, sólo basta con copiarlo con el siguiente comando.(véase figura 6.6)

Figura 6.6 Instalación del plug-in de la máquina virtual de Java para el navegador Firefox

6.1) Instalación y configuración de un entorno de desarrollo

Para la instalación del entorno de desarrollo de X10, es necesario descargarlo desde la página oficial *x10-lang.org* (véase figura 6.7)

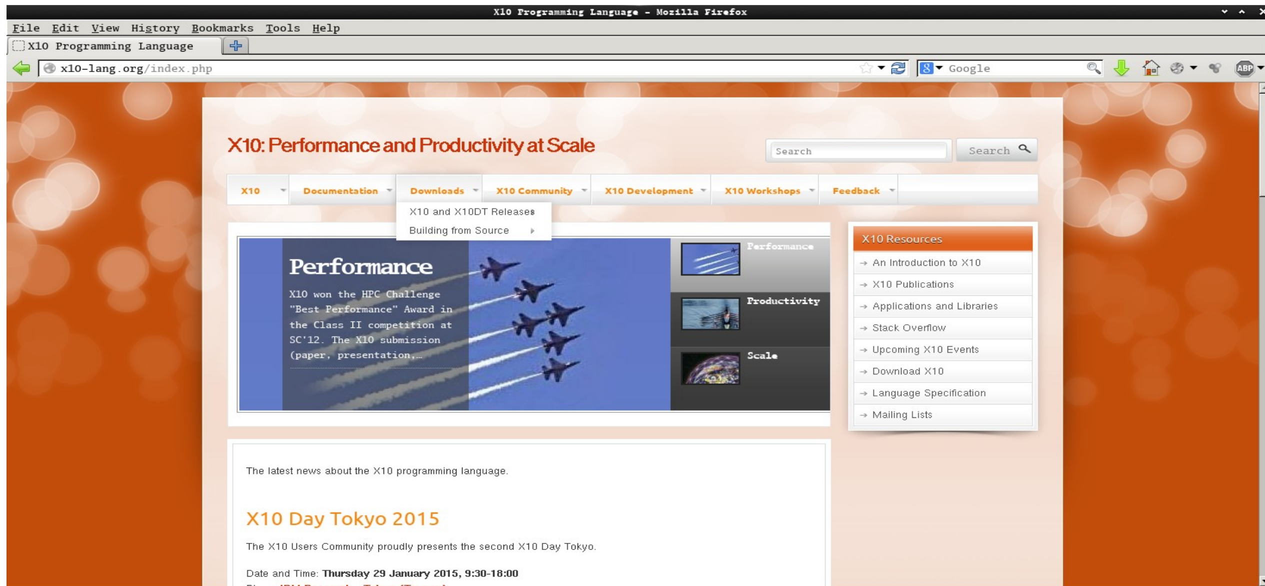


Figura 6.7 Página del lenguaje X10

Dentro de la página se encuentran varios ejecutables y el código fuente. El entorno de desarrollo que se descarga es X10DT para X86.(véase figura 6.8)

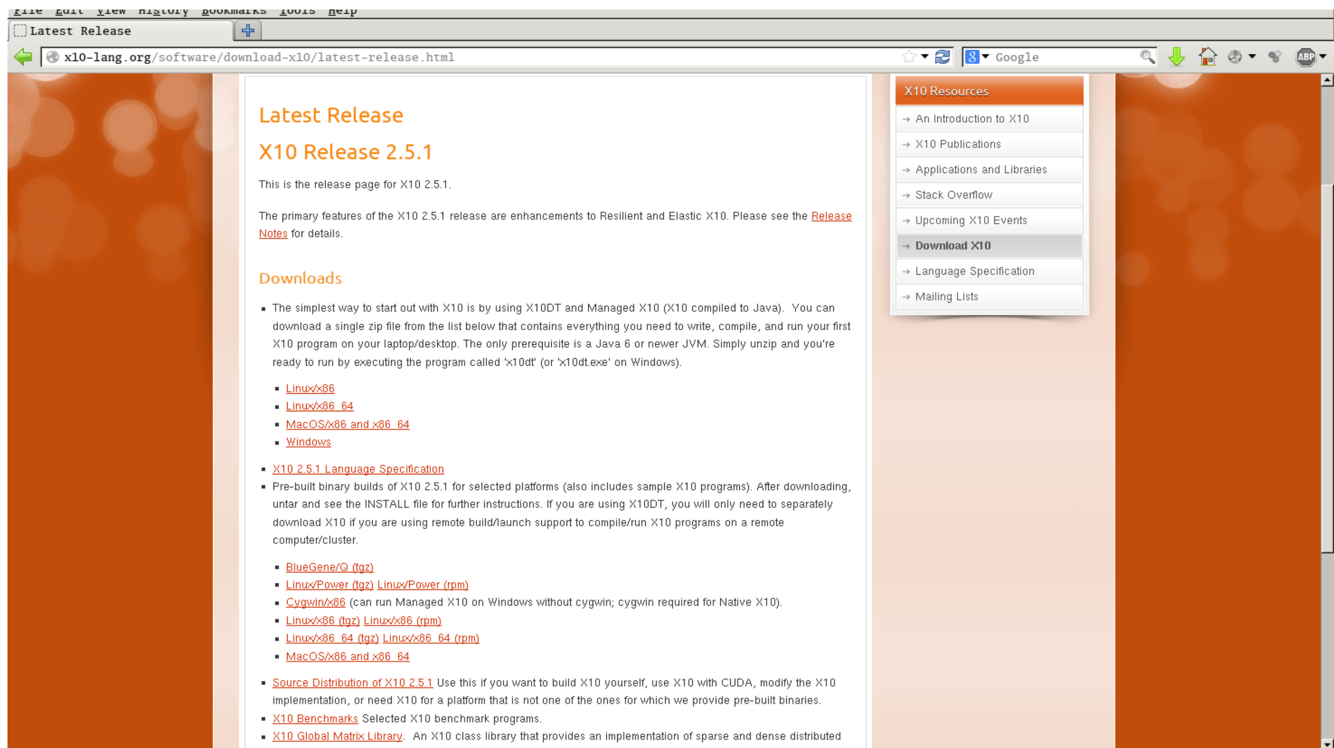


Figura 6.8 Página de descarga de X10dt

Ya que tengamos el archivo, es necesario descomprimirlo con el siguiente comando. (véase figura 6.9)

Figura 6.9 Descompresión del archivo de X10dt

Al terminar la descompresión, se creará el directorio *x10dt* en el cual se encuentra el entorno de ejecución, incluyendo los compiladores y la máquina virtual *x10*.(véase figura 6.10)

Figura 6.10 Detalles de la descompresión del archivo X10dt

Para finalizar, ejecutamos el entorno de desarrollo (IDE) como se muestra en la figura 6.11.

```
xterm
bash-4.2# cd x10dt
bash-4.2# ls
about.html     configuration  features  notice.html  plugins  x10dt
artifacts.xml  epl-v10.html   icon.xpm  p2           readme   x10dt.ini
bash-4.2# ./x10dt
```

Figura 6.11 Ejecución del entorno de X10dt

El entorno de desarrollo se encuentra basado en eclipse.(véase figura 6.12)

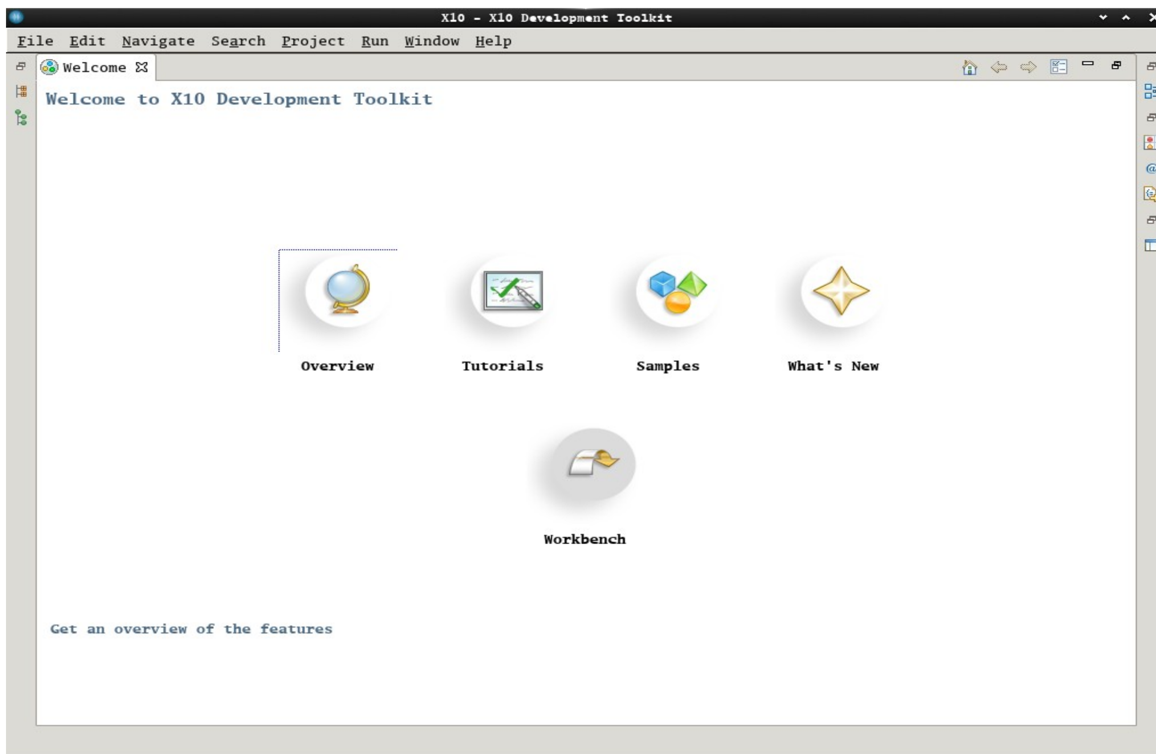


Figura 6.12 Bienvenida al entorno de X10dt

Al crear un nuevo proyecto, podemos elegir si utilizaremos Java o C para su compilación y

ejecución. Por defecto el entorno de desarrollo nos creará el ejemplo de HolaMundo como se muestra en la figura 6.13.

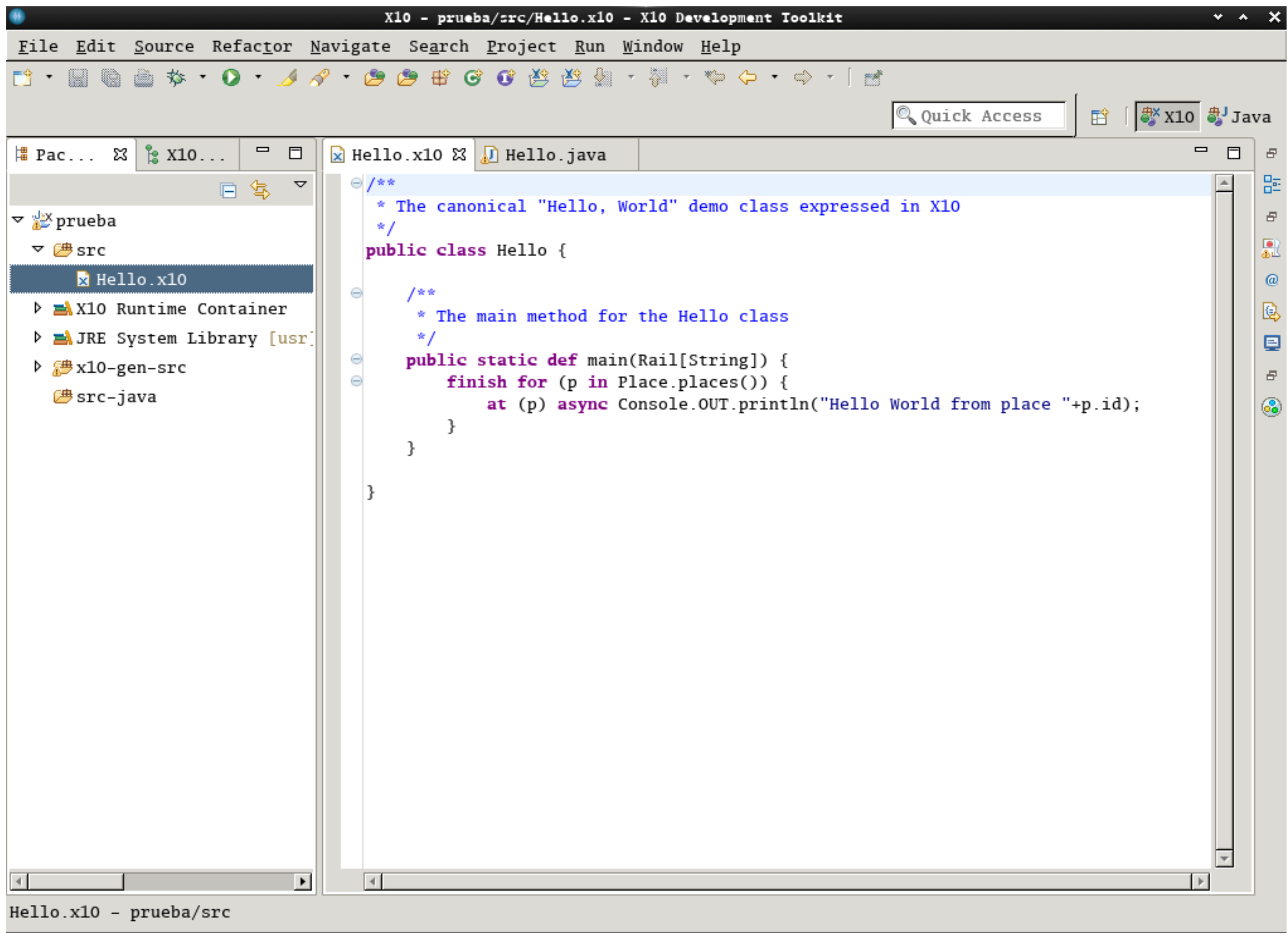


Figura 6.13 Hola Mundo en el entorno de desarrollo de X10

6.2) Instalación y configuración de X10 en arquitectura x86 y Sistema Operativo GNU\Linux

En caso de que no se quiera utilizar el entorno de desarrollo de X10,podemos escribir el código fuente en cualquier editor o procesador de texto y utilizar el compilador de X10.
Para instalar el compilador de X10 y la máquina virtual X10, se descarga el archivo comprimido desde la página oficial *x10-lang.org* (véase figura 6.14)

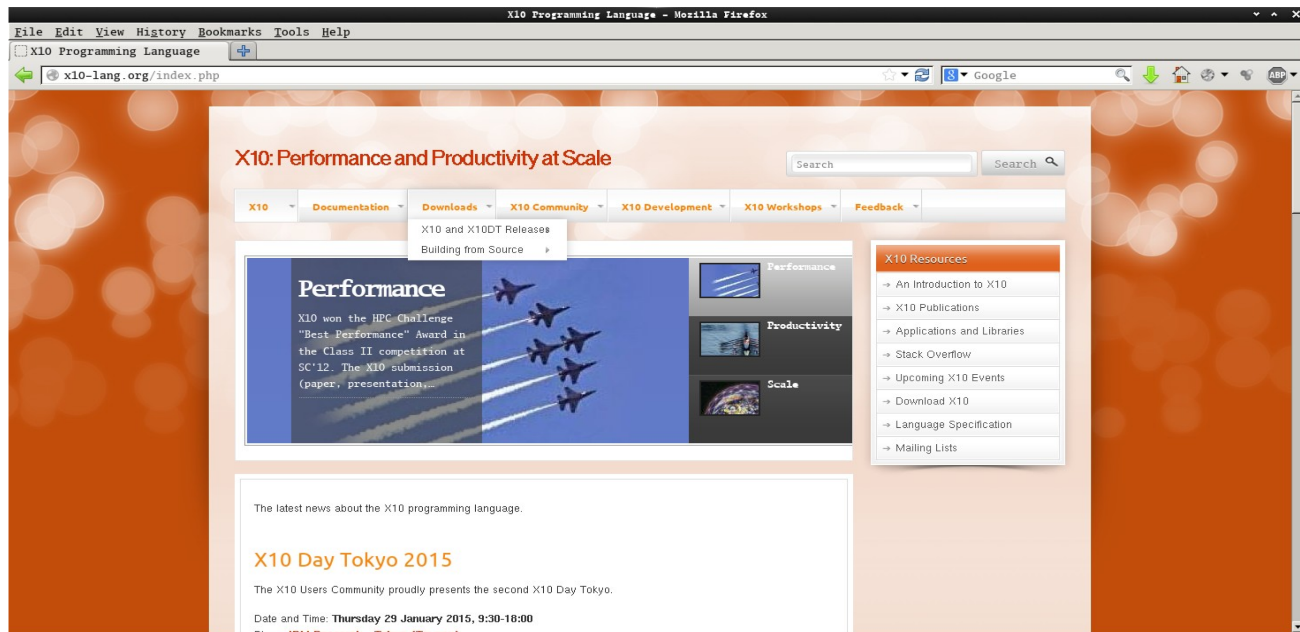


Figura 6.14 Página de X10

Descargamos el archivo X10 para arquitectura X86. (véase figura 6.15)

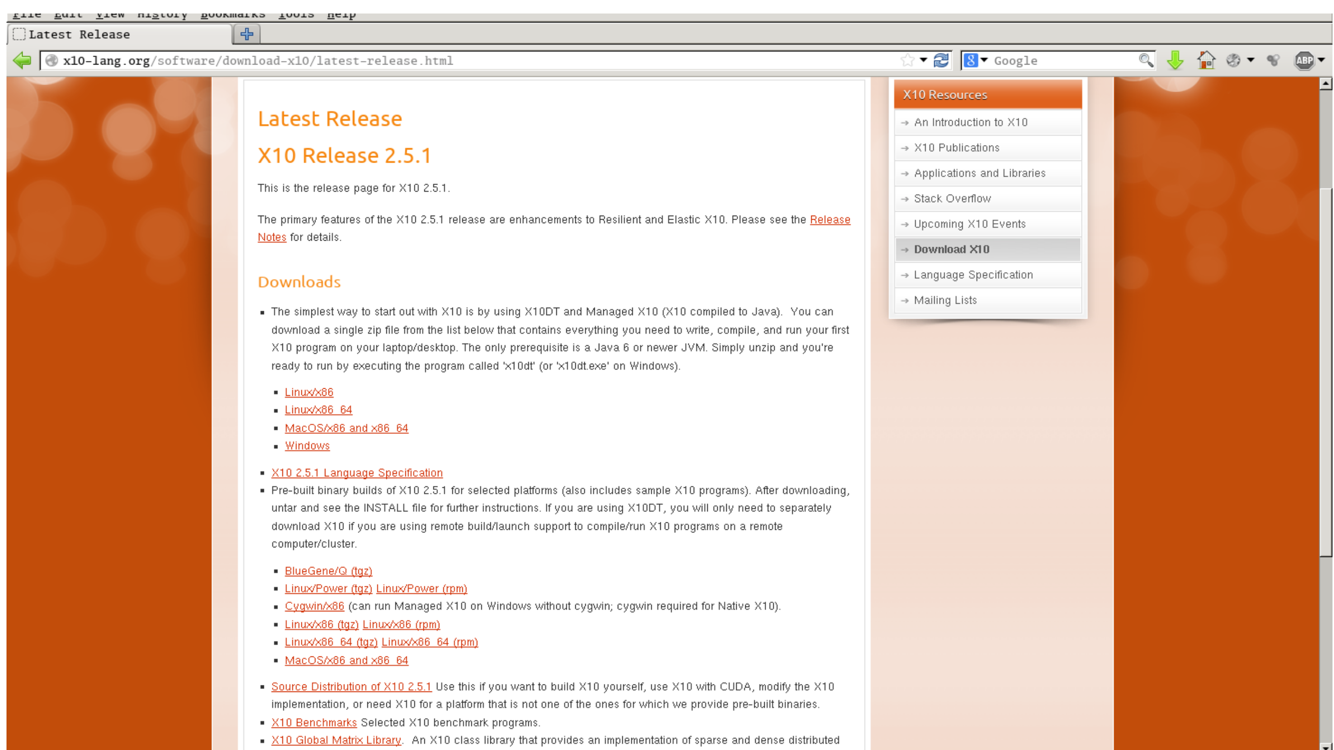


Figura 6.15 Página de descarga de X10

Una vez que se tenga el archivo de X10, se necesita crear un directorio y descomprimir el contenido de X10 en el. Lo que se muestra en la figura 6.16.

Figura 6.16 Descompresión de X10

El archivo contiene los ejecutables, bibliotecas y ejemplos de programas (incluyendo algunos con CUDA).(véase figura 6.17)

Figura 6.17 Detalles de la descompresión de X10

Al descomprimir el archivo se creará la siguiente estructura de directorios.(véase figura 6.18)

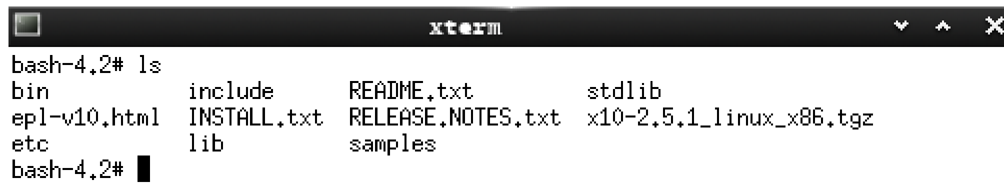


Figura 6.18 Listado del directorio de X10

Al ejecutar el compilador de X10 con Java, se mostrará un error porque no se tiene definida la ruta de la instalación de Java. Para corregir esto se define la variable JAVA_HOME ,como se muestra en la figura 6.19.

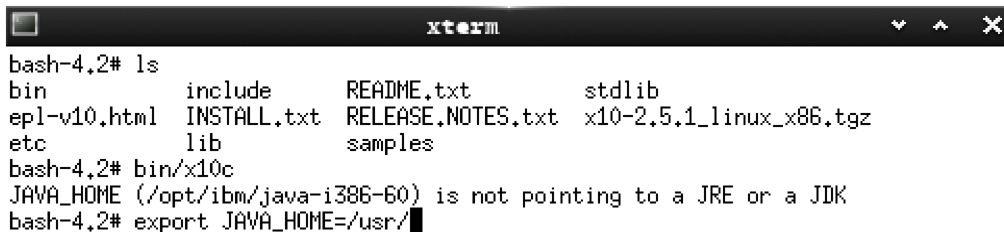


Figura 6.19 Exportar la variable de entorno X10

Por último volvemos a ejecutar el compilador y si todo es correcto, se mostrará la ayuda de los parámetros que recibe.(véase figura 6.20)

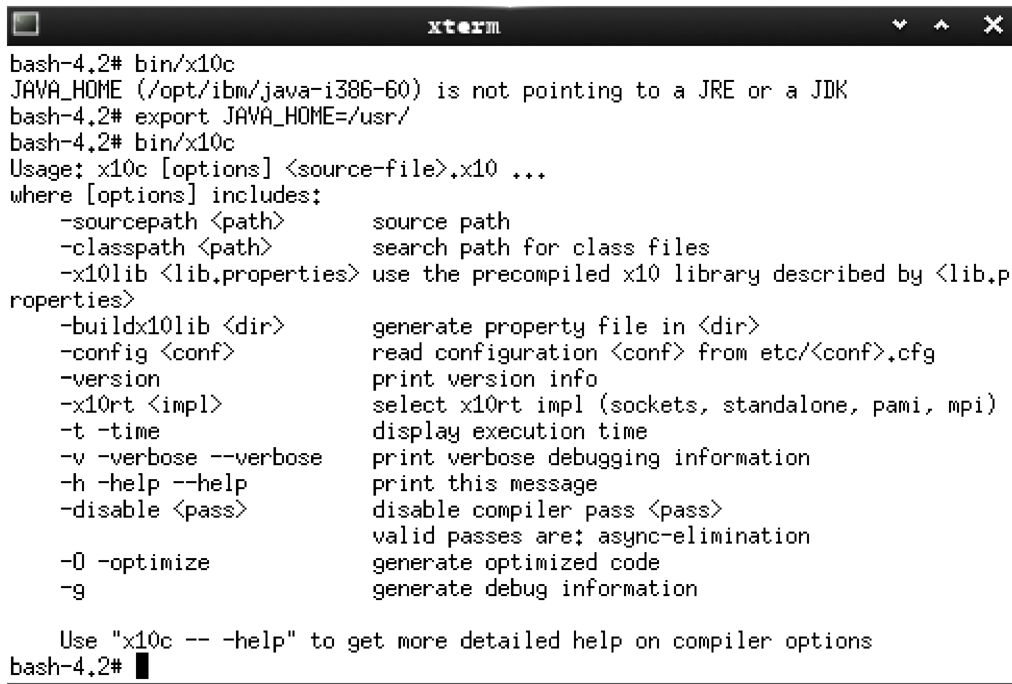


Figura 6.20 Ejecución de X10c

7) Bibliografía

Breshears Clay . **“The Art of Concurrency”**. O'Reilly.2009. ISBN: 978-0-596-52153-0

Rodríguez Clemente, Álvarez Gonzalo ,et al. **“Microprocesadores RISC: Evolución y tendencias”**. RA-MA,2000.230p. ISBN 970-15-0508-5.

Gebali Fayez. **“Algorithms and Parallel Computing”**. John Wiley & Sons, 2011.364p . ISBN: 978-0470902103

Tanenbaum Andrew. **“Sistemas Operativos Distribuidos”**. Prentice Hall.1996. 617p. ISBN: 968-880-627-7

Olivier Tardieu. **“APGAS Programming in X10”.** Hartree Centre Summer School 2013 “Programming for Petascale”.

Poblete Muñoz Tania Xitlali. “**Desarrollo de un sistema de procesamiento distribuido usando bibliotecas para la programación paralela en el modelo de intercambio de mensajes”**. Tesis de Licenciatura. Facultad de Ingeniería. Universidad Nacional Autónoma de México. 2001.

Ramos Palacios Guadalupe Susana y Rosales Madrigal Servando. “**Inducción al cómputo de alto desempeño”.**Tesis de Licenciatura. Facultad de Ingeniería. Universidad Nacional Autónoma de México. 2006.

Ceballos Sierra Fco. Javier. **“Java 2 Curso de Programación”**. Alfaomega. 2003. 778p. ISBN: 970-15-08491

Chapman, Barbara. **“Using OpenMP”**. Massachusetts Institute of Technology. 2008. 353p. ISBN: 978-0-262-53302-7

Amnon Barak and Amnon Shiloh.**”MOSIX Cluster Management System, Administrator's, User's and Programmer's Guides and Manuals”**.2015. 96p.

J.E. Gentle, Wolfgang Hsirdle. ”**Handbook of Computational Statistics”**. Springer. 2004. 900p. ISBN: 978-3540404644

Alan Clements. **“Principles of Computer Hardware”**. Oxford University Press. 2006. 672p. ISBN: 978-0199273133