UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Operaciones matriciales con técnicas de cómputo de alto rendimiento sobre una red de nodos heterogéneos.

TESIS

QUE PARA OBTENER EL TÍTULO DE:

Ingeniero en Computación

PRESENTAN:

Omar Guerrero Ruiz

Josue Daniel Tapia Guerrero

DIRECTOR:

Ing. José Antonio Ayala Barbosa

Ciudad Universitaria, Cd. Mx., 2021

Agradecimientos

Omar Guerrero Ruiz.

A MIS PADRES Y HERMANOS

Porque sin ustedes, Homar y Tere, se que no hubiera logrado llegar hasta aquí. A ustedes les reconozco y les estaré eternamente agradecido por todo el amor, orgullo, apoyo, consejos, por la confianza, tiempo y esfuerzo invertidos en mí. A ustedes mis padres les agradezco por la vida que me regalaron y quiero que estén seguros que esta meta la estamos cumpliendo juntos como equipo y que siempre serán la parte más importante de todo este camino recorrido. A mis hermanos Cinthia, Ángel y Diana, les agradezco por todo el amor, las risas y la oportunidad de hacer crecer mi conocimiento enseñándoles y compartiéndolo con ustedes. Los amo infinitamente y espero poder devolverles todo esto que me dieron a lo largo de todos estos años. Este es uno de los mayores logros y lo estamos cumpliendo juntos.

A MI NOVIA

Que estuvo conmigo en los últimos pasos para terminar esta increíble etapa de mi vida. A ella le reconozco todo ese amor y apoyo tan sincero que me brindó. Como leí una vez: uno puede lograr cientos de cosas, todo lo que se proponga, pero nunca va a ser lo mismo que tener a alguien a tu lado que te apoye y te diga que eres el mejor y que tú puedes lograrlo. Te amo Fanny y gracias por estar conmigo.

A MIS PROFESORES Y MAESTROS

Les agradezco y reconozco por uno de los mayores regalos que alguien pudo darme, y es todo ese conocimiento, sabiduría y capacidad que me brindaron para poder analizar y ver de forma diferente el mundo que me rodea. Les agradezco por todo ese tiempo, por los consejos y la experiencia, pues gracias a todo eso ahora estoy logrando cumplir una de mis mayores metas, que es convertirme en INGENIERO.

A MIS AMIGOS

Es increíble cómo la vida pone en tu camino a personas tan maravillosas, personas que sin serlo, se vuelven tus hermanos o hermanas. A todos ustedes les agradezco el tiempo que estuvieron conmigo, su amistad, todo el apoyo, sus palabras de aliento y compañía que me brindaron a lo largo de este camino.

A Josue

A ti mi hermano te agradezco la confianza que depositaste en mi, te agradezco el tiempo y el esfuerzo invertido para lograr esta meta a la que ahora estamos llegando juntos. En la vida, al menos en los 24 años que tengo la dicha de vivirla, se aprende que no en cualquier lado, ni tan fácil, encuentras a personas en quien confiar y con quien puedas trabajar sin miedo a que fallen. Te agradezco por ser, no solo un amigo sincero, también por ser un gran equipo, por persevar junto conmigo y por lograr esta meta como hermanos y de corazón espero que esta amistad y apoyo mutuos perduren por muchos años más y lograr grandes cosas juntos.

Josue Daniel Tapia Guerrero

EL ESFUERZO DE ELABORACIÓN DE ESTE TRABAJO LO DEDICO A:

A MI PADRE, JOSUÉ

La persona que más amo en esta vida, quien ha sabido sacarme adelante pese a situaciones adversas, una de las personas más sabias que conozco, por brindarme consejos que me han ayudado a la toma de decisiones, por tratar de siempre darnos lo mejor y esforzarse por ser el mejor padre.

A MI HERMANA, DANIELA

Por brindarme su amor sincero e incondicional, por fomentar en mi la paciencia y las ganas de ser una mejor persona, por compartir conmigo risas y momentos increíbles.

A MI ABUELA HERMINIA

Quien me impulsó siempre a ser una mejor persona, me abrió las puertas de su hogar y me abrigó con tanto amor, tal y como lo hace una madre, mi abuela quien me enseñó, que el trabajo duro y honrado al final del día trae consigo las mejores recompensas.

A MI NOVIA

Mi compañera de vida, a la persona por la cual cada día trato de ser mejor, por brindarme todo su amor y nunca dejarme solo en situaciones complejas, por hacerme ver que en este mundo aún hay personas con corazones nobles y puros.

A MI MADRE, JANETE

Por darme la vida, por encaminar mi carácter y la persona que ahora soy, por cuidarme, protegerme y sobre todo sacarme adelante ante enfermedades.

A MIS AMIGOS Y COMPAÑEROS

Con quienes me enfrenté a retos complicados, a quienes he tenido la dichosa oportunidad de ver crecer, a todos ustedes por el simple hecho de saber que puedo contar con ustedes en cualquier momento. A mi compañero, amigo, hermano, Omar,

con quien pude tener el placer de formar este trabajo escrito, por apoyarme cuando lo requería, por darme su confianza y sobre todo, por extenderme su mano siempre.

Índice general

1. Introducción **13**

1.1. Contexto . . . 13
1.2. Problema . . . 14
1.3. Objetivo . . . 14
1.4. Contribuciones . . . 15

2. Antecedentes **16**

2.1. Cómputo en la Actualidad . . . 16
2.2. Lenguajes de programación . . . 17
2.2.1. Definición de lenguaje de programación . . . 17
2.2.2. Historia . . . 18
2.2.3. Compilador e Intérprete . . . 19
2.2.4. Características de los lenguajes de programación . . . 19
2.2.5. Lenguaje C . . . 20
2.3. Cómputo de Alto rendimiento . . . 21
2.3.1. Taxonomía de Flynn . . . 28
2.3.2. Comunicación ente procesos . . . 30
2.3.3. Cómputo concurrente . . . 34
2.3.4. Cómputo Paralelo . . . 34
2.3.5. Cómputo Distribuido . . . 34
2.4. MPI . . . 35

2.4.1. Definición MPI(Message Passing Interface) 35
2.4.2. Ventajas 37
2.4.3. Historia 37
2.4.4. Bibliotecas MPI y compiladores para MPI 38
2.4.5. Estructura de código en MPI 40
2.4.6. Rutinas MPI 43
2.4.7. Métodos de sincronización 46
2.5. OpenMP 48
2.5.1. Computadoras paralelas de memoria compartida 48
2.5.2. Definición de OpenMP 50
2.5.3. Directivas, constructores y barreras de OpenMP 52
2.6. Redes de datos 59
2.6.1. Objetivo de las redes de datos 59
2.6.2. Medios de transmisión 59
2.6.3. Dispositivos intermediarios 60
2.6.4. Protocolo 60
2.7. Internet de las cosas 62
2.7.1. Procesamiento distribuido en internet de las cosas 64
2.8. Clúster tipo Beowulf 65
2.8.1. Historia 65
2.8.2. Características 66
2.8.3. Ventajas y desventajas 67
2.9. Requerimientos de software 68
2.9.1. Requerimientos funcionales 68
2.9.2. Requerimientos no funcionales 68

3. Álgebra matricial **70**
3.1. Definición de matriz 70
3.1.1. Definición 70
3.1.2. Orden de una matriz 71

3.1.3. Tipos de matrices 71
3.2. Operaciones matriciales 74
3.2.1. Suma de matrices 74
3.2.2. Resta de matrices 75
3.2.3. Multiplicación por un escalar 75
3.2.4. Multiplicación entre matrices 76
3.2.5. Potencia de una matriz 77
3.2.6. Matriz de cofactores 77
3.2.7. Matriz adjunta 77
3.2.8. Determinante 78
3.2.9. Inversa 78

4. Implementación 80
4.1. Diseño de la implementación del caso de estudio 80
4.1.1. Diagrama de casos de uso 81
4.1.2. Diagramas UML 82
4.1.3. Diagrama de Flujo 84
4.2. Levantamiento de ambiente 85
4.2.1. Instalación GNU/Linux 85
4.2.2. Instalación y configuración de SSH 86
4.2.3. Instalación de Sistema de Archivos de Red 88
4.2.4. Instalación MPI 90
4.3. Software 92
4.3.1. Métodos computacionales para la implementación de las operaciones 92
4.3.2. Técnicas de programación utilizadas 98
4.4. Método de sincronización utilizado 104
4.5. Hardware 105

5. Métricas de desempeño **106**
5.1. Definición 106
5.1.1. Tiempo de ejecución 107
5.1.2. Factor de costo 108
5.1.3. Factor de aceleración 109
5.1.4. Eficiencia 110
5.2. Métricas de desempeño propuestas 111
5.2.1. Tiempo de uso de la memoria 111
5.2.2. Tamaño de memoria utilizado 112
5.2.3. Planteamiento de las pruebas: Enfoque 113

6. Resultados **116**
6.1. Secuencial y Paralelo 116
6.1.1. Variando el número de procesadores 116
6.1.2. Variando el tamaño de la matriz 122
6.1.3. Variando el tipo de dato 125
6.2. Distribuido y Paralelo-Distribuido 127
6.2.1. Variando el número de procesadores 127
6.2.2. Variando el tamaño de la matriz 130
6.2.3. Variando el tipo de dato 131
6.3. Pruebas operación inversa y división 133

7. Conclusiones y trabajo futuro **135**
7.0.1. Trabajo Futuro 142

Referencias **144**

Índice de figuras

2.1. Caso 1: Un procesador 23
2.2. Caso 2: Dos procesadores trabajando en paralelo 24
2.3. Taxonomía de Flynn, Basado en [16] 28
2.4. Clasificación de la taxonomía de Flynn, Basado en [13] 29
2.5. Comunicación entre procesos [19] 30
2.6. Comunicación asíncrona 31
2.7. Comunicación síncrona 31
2.8. Comunicación persistente 31
2.9. Comunicación momentánea 32
2.10. Comunicación directa 32
2.11. Comunicación indirecta 32
2.12. Comunicación bidireccional 33
2.13. Comunicación asimétrica 33
2.14. Comunicación Buffer automático 33
2.15. Clasificación de los tipos de cómputo. [12] 35
2.16. Arquitectura con memoria distribuida 36
2.17. Arquitectura con memoria compartida 37
2.18. Flujo de programa en MPI [24] 41
2.19. Computadora tipo UMA, Basado en [27] 49
2.20. Computadora tipo ccNUMA, Basado en [27] 50
2.21. Modelo OSI comparado a el modelo TCP/IP [31] 61

2.22. ISO 25010 [40] 69

4.1. Diagrama de casos de uso del sistema HPC 81
4.2. Paso de Mensajes: suma, resta, multiplicación por un escalar, igualdad e inversa. 82
4.3. Paso de Mensajes: multiplicación y potencia. 83
4.4. Diagrama de flujo del programa 84

5.1. Ejemplo: Métrica tiempo de ejecución 108
5.2. Ejemplo: Métrica factor de costo 109
5.3. Ejemplo: Métrica factor de aceleración 110
5.4. Ejemplo: Métrica eficiencia 111
5.5. Ejemplo: Métrica tiempo uso de memoria 112
5.6. Ejemplo: Métrica tamaño de memoria utilizada 113

6.1. Tiempo de ejecución secuencial vs paralelo 117
6.2. Factor de costo secuencial vs paralelo 119
6.3. Factor de aceleración secuencial vs paralelo 120
6.4. Eficiencia secuencial vs paralelo 121
6.5. Tiempo de ejecución Secuencial - Paralelo 122
6.6. Factor de costo Secuencial - Paralelo 124
6.7. Tiempo de ejecución Secuencial - Paralelo 125
6.8. Factor de aceleración Secuencial - Paralelo 127
6.9. Tiempo de ejecución Distribuido vs Paralelo-Distribuido 128
6.10. Factor de costo Distribuido vs Paralelo-Distribuido 129
6.11. Factor de aceleración Distribuido vs Paralelo-Distribuido 129
6.12. Eficiencia Distribuido vs Paralelo-Distribuido 130
6.13. Tiempo de ejecución 131
6.14. Tiempo Distribuido vs Paralelo-Distribuido 132
6.15. Factor de costo Distribuido vs Paralelo-Distribuido 133
6.16. Tiempo de ejecución inversa. 134

7.1. Tiempo de ejecución variando número de procesadores. 141
7.2. Tiempo de ejecución variando tamaño de la matriz 141
7.3. Tiempo de ejecución variando el tipo de dato 142

CAPÍTULO

1 Introducción

1.1 Contexto

Debido a la necesidad de procesar cantidades cada vez más grandes de datos en una computadora; la demanda de dispositivos capaces de llevar a cabo esta tarea de forma eficiente y rápida también va en aumento. Como consecuencia, el consumo energético también se ha disparado al incrementarse el número de transistores presentes en un procesador, ya que, en su momento aumentarlos era una opción considerable. Esto obliga a la búsqueda de alternativas en las cuales en lugar de incrementar el poder de procesamiento de un solo dispositivo (procesador), se busquen soluciones en las cuales varios dispositivos trabajen en conjunto haciendo uso de técnicas de cómputo de alto rendimiento (HPC). De esta forma se aumenta la capacidad para procesar grandes cantidades de información.

Al hacer uso de varios dispositivos para resolver una tarea, es necesario llevar a cabo una conexión entre ellos de manera que, puedan intercambiar y procesar la información de la forma más rápida, constante, eficiente y económica posible. Esto nos lleva a pensar en el diseño he implementación de redes con nodos heterogéneos con las cuales se pueda cumplir esta meta.

En diferentes campos de la ciencia y la ingeniería existen problemas que requieren de un gran poder de cómputo para ser resueltos. Dichos problemas, en su mayoría, recurren a métodos numéricos basados en operaciones con matrices, los cuales en general son altamente paralelizables, por lo que con ayuda de técnicas de cómputo de alto rendimiento (HPC) pueden ser divididos y distribuidos en un grupo de computadoras que en conjunto lleven a cabo la solución de dichos problemas. El análisis y solución de problemas matriciales con grandes cantidades de datos resulta ser uno de los casos en donde más poder de procesamiento se requiere ya que, aunque son operaciones sencillas, al incrementarse la cantidad de datos se vuelve una tarea difícil y en ocasiones muy pesada para un solo dispositivo, por lo cual este problema será tomado como principal caso de estudio.

1.2 Problema

El principal problema al que nos enfrentamos es el poder resolver operaciones matriciales con grandes cantidades de datos, con cuatro sistemas distintos y con recursos computacionales limitados, ya que, para la construcción del cluster se utilizaran computadoras con características comerciales diferentes. Esto complica la construcción del cluster y la comunicación entre ellas.

1.3 Objetivo

El principal objetivo de esta tesis es poder determinar bajo qué condiciones es mejor hacer uso de un programa secuencial, paralelo o uno distribuido, según la cantidad y tipos de datos con los que se estén trabajando. Con base en esto, se pretenden implementar cuatro programas que puedan resolver operaciones matriciales básicas, con cuatro diferentes tipos de datos en lenguaje C, haciendo uso de las técnicas de cómputo de alto rendimiento. Se implementará un clúster de tipo Beowulf con computadoras que poseen diferentes recursos computacionales.

Los programas implementados se pasarán de una ejecución secuencial a una paralela y se observará una mejora en el rendimiento del programa. Esta mejora se presentará sin importar el tamaño de las matrices con las que se harán las pruebas. Por otro lado, al pasar de una ejecución paralela a distribuida, de igual forma se observará una mejora en el rendimiento del programa.

1.4 Contribuciones

Se pretende que a partir de los resultados obtenidos en esta tesis, los desarrolladores que necesiten implementar software haciendo uso de técnicas de cómputo de alto rendimiento tengan una base a partir de la cual puedan determinar de forma más sencilla y rápida, que técnicas utilizar, con qué tipos de datos trabajar, el tipo de hardware que puede tener cada nodo dentro del cluster, así como el número de procesadores a utilizar.

Adicional a esto, se espera que los resultados obtenidos se utilicen como un escalón más para avanzar y profundizar en el conocimiento que se posee actualmente sobre el cómputo de alto rendimiento y que éste sea utilizado incluso para generar nuevo conocimiento, ya que aún hay muchas áreas de oportunidad que cubrir y demasiado por descubrir.

CAPÍTULO

2

Antecedentes

2.1 Cómputo en la Actualidad

El creciente aumento del uso de la tecnología en nuestros días ha traído consigo un sinfín de problemas que aún se encuentran sin resolver o tienen soluciones parciales. Desde inconvenientes en la transferencia y almacenamiento, hasta complicaciones en el procesamiento y tiempo de cómputo. Estos son problemas con los que día a día el personal de tecnologías de la información y ciencias se enfrentan en sus jornadas laborales.

Uno de los mejores ejemplos en donde al menos tres de estos inconvenientes se manifiestan, es en el recién establecido concepto "Internet de las cosas". Este, por su gran avance, ha formado una gran popularidad a lo largo del mundo por la inmensa cantidad de objetos y áreas que puede abarcar. Sin embargo, la cantidad de datos generados a partir de todas las tecnologías que involucra, es inmensa, lo cual genera no solo una necesidad de aumentar el espacio de almacenamiento para todos los datos, también una necesidad de poder de procesamiento para analizar, filtrar, y almacenar dichos datos.

Esto lleva a muchas empresas y organizaciones a buscar un aumento considerable de recursos computacionales para poder solventar esa gran demanda de procesa-

miento. Toda esa infraestructura empleada para resolver tareas conlleva una gran inversión de hardware, mantenimiento y energía que a largo plazo generan grandes gastos y en caso de requerir un escalamiento mayor, el costo final no resulta favorable.

2.2 Lenguajes de programación

2.2.1 Definición de lenguaje de programación

La computadora es un conjunto de sistemas que se comunican por medio de señales las cuales pueden ser de dos tipos: alto o bajo voltaje. A estos tipos de señales se les conoce como bits y se pueden representar como 1 ó 0 respectivamente. Las personas, por otro lado, se comunican también por medio de señales pero estas tienden a ser más complejas y con mayor variabilidad, la única forma en que una computadora y una persona se puedan comunicar es bien o que la computadora pueda interpretar el mismo lenguaje complejo del ser humano o que este último se adapte a la comunicación por medio de ceros y unos. Si un humano trata de dar instrucciones a una computadora por medio de bits, entre más largas y complejas sean éstas, más difícil será poder transmitir sus ideas a esta codificación, sin mencionar la parte en donde se debe conocer con precisión los componentes del procesador para poder manipular los registros de éste y poder conseguir el comportamiento esperado, en consecuencia tiende a ser demasiado tedioso, con muchos errores y en el peor de los casos imposible de llevar a cabo.

Debido a lo anterior el propósito de un lenguaje de programación es poder acercarse en la medida de lo posible a un lenguaje con el cual una persona pueda representar con mayor facilidad un conjunto de instrucciones que la computadora va a realizar, y posteriormente convertirlas a un lenguaje máquina y de esta forma ofrecer una comunicación entre computadora y humano mucho más sencilla. Sin embargo, el desarrollar lenguajes de programación no es una tarea fácil, ya que entre más parecido sea éste a un lenguaje humano, su elaboración tiende a ser más compleja.

2.2.2 Historia

Retomando momentos importantes de la historia de la computación digital, al rededor de 1940, la entrada de datos y la programación de las computadoras era por medio de interruptores, esto complicaba el hecho de poder implementar una instrucción. [1]

Comienza la primera generación de computadoras en donde la entrada de datos y programas para estas computadoras era por medio de tarjetas perforadas. Estas tarjetas seguían una serie de patrones que tenían un significado único. Más tarde se implementaron símbolos que facilitaban las instrucciones implementadas en lenguaje máquina a una forma más textual, es decir, cada instrucción contenía su propio símbolo o mnemónico, de esta forma se asociaba cada palabra con una instrucción en código máquina. A esto se le denominó lenguaje ensamblador.

Más tarde aparece el primer lenguaje de programación llamado FORTRAN (Formula Translation) en 1957, cuyo compilador constaba de 25.000 instrucciones en lenguaje ensamblador. En 1959 surgió el lenguaje COBOL (Common Business Oriented Language). Un año después surge LISP (List Processing) creado por John McCarthy como lenguaje especializado para la manipulación de datos no numéricos. Finalmente, en 1970 surge Pascal y C como lenguajes estructurales. [1], [2]

La evolución de los lenguajes de programación se dio de manera muy rápida lo cual dio origen a diversos paradigmas con diferentes propósitos. Hasta el día de hoy se sigue el diseño de lenguajes de programación que resuelvan ciertas tareas de la forma más sencilla posible. Por lo tanto, un punto importante que se debe tener en cuenta a la hora de diseñar un lenguaje de programación es la eficiencia, y es que entre más sencillo sea de entender, tiende a ser un poco menos eficiente debido a que la computadora se encarga de traducir lo que el programador escribió, esto con ayuda del traductor, ya sea un intérprete o bien, un compilador.

2.2.3 Compilador e Intérprete

Los traductores son el núcleo de un lenguaje de programación cuyo objetivo, como su nombre lo indica, es traducir las instrucciones del programador a lenguaje máquina, de aquí se derivan en dos, los intérpretes y los compiladores. Por un lado, un lenguaje de programación interpretado como lo es el caso de Python se encarga de leer cada línea de código en tiempo de ejecución, mientras que un lenguaje compilado tiene que realizar un análisis para producir un código máquina, el cual la computadora solo tiene que ejecutar. Un lenguaje interpretado tiende a ser más fácil para el programador debido a que las instrucciones se van interpretando en tiempo de ejecución, pero suele ser más costoso computacionalmente hablando; por otro lado, un lenguaje compilado tiende a ser más eficiente debido que al ejecutarlo previamente ya se realizó una compilación que facilita así el trabajo a la computadora. [3], [4]

2.2.4 Características de los lenguajes de programación

Un paradigma de programación es una forma o técnica de generar un programa. Se tienen lenguajes de programación que son compilados y otros que son interpretados, cuyas características ya se explicaron en el apartado anterior. Existe otra clasificación que diferencia a un lenguaje de programación y otro por el nivel al que se establecen cada una de las instrucciones, y el parecido o acercamiento que éstas tienen con el lenguaje máquina (bits). En esta clasificación podemos encontrar a los lenguajes de alto y bajo nivel.

En lenguajes de bajo nivel se tiene el lenguaje ensamblador, siendo el ejemplo más relevante en este tipo de nivel ya que se encuentra lo más cercano al hardware, se pueden manipular directamente localidades de memoria, estados de registros, controlar el *program counter*, alteraciones de banderas, entre muchas otras cosas. Debido a esto, es más eficiente que los lenguajes de programación de alto nivel, pero mucho más complejo de entender. [5]

Por otro lado, se tienen los lenguajes de alto nivel cuyas particularidades radican

en la forma tan fácil que es implementar o diseñar un programa ya que estos son más cortos y sencillos de leer, a su vez esto disminuye el hecho de cometer más errores. Otra característica es su portabilidad, lo cual significa que el programa fuente puede compilarse y ejecutarse en diferentes tipos de computadoras sin la necesidad de llevar a cabo ninguna modificación. Dentro de los ejemplos de lenguajes de alto nivel podemos encontrar a Java, Python, C y C++. [5], [6]

2.2.5 Lenguaje C

El origen de C comenzó en los laboratorios Bell por Dennis Ritchie a mediados de los años 60's con el fin de crear rutinas en Unix. El lenguaje C es un lenguaje compilado y denominado de alto nivel, tal y como se mencionó anteriormente, sin embargo al ofrecer la posibilidad de combinar elementos de alto nivel con la funcionalidad de los ensambladores, se puede caracterizar como un lenguaje de nivel medio. C tiene la característica de poder manipular principalmente localidades de memoria de una forma mucho más sencilla, con los denominados apuntadores o punteros lo cual permite tener un manejo de la memoria más controlado, así como una mayor eficiencia en los programas, ya que incluso se pueden manipular bytes. [7], [8]

```c
1  #include <stdio.h>
2  void holaMundo(char * a);
3
4  int main(){
5      holaMundo("\n Hola Mundo \n");
6      return 0;
7  }
8  void holaMundo(char * a){
9      printf("%s",a);
10 }
```

Listing 2.1: Ejemplo de código en C

Retomando como ejemplo el código 2.1, en la línea 1 se pueden apreciar las directivas para el preproceso, con estas se le indica al compilador que debe agregar

esa biblioteca al programa que se creó. En la línea 2 se crea un prototipo de la función, como se puede observar, no se tiene contenido, sólo es la declaración de la función. En la línea 4 se crea la función principal, en un programa en C siempre debe aparecer sólo una vez ya que todo lo que esté dentro de la función *main* sera ejecutado. Finalmente, en la línea 8 se observa la declaración de la función, es decir, tiene escrito todo su comportamiento. Es importante destacar que en la sintaxis del lenguaje C es de suma importancia terminar cada instrucción con punto y coma y cada bloque de código con llaves. Como C es un lenguaje compilado, el programa fuente debe pasar por un preprocesador, un compilador y un enlazador para obtener un programa ejecutable. Bajo el sistema operativo linux, en sus diversas versiones y distribuciones, se generan todos los pasos de la compilación en una sola línea, es decir, se preprocesa, compila, enlaza y se genera el código ejecutable con la siguiente instrucción.

```
$ gcc programaC.c -o programaC
```

Ejecutando el código con el archivo generado de la siguiente forma.

```
$ ./programaC
```

obteniendo en consola la siguiente salida

```
$ ./programaC
 Hola Mundo
$
```

2.3 Cómputo de Alto rendimiento

El cómputo de alto rendimiento se puede definir como un conjunto de tecnologías que brindan la capacidad para procesar una gran cantidad de datos, así como la habilidad para poder realizar operaciones, cálculos y/o tareas muy complejas a grandes velocidades. El cómputo de alto rendimiento, al cual a partir de ahora nos referiremos como **HPC (High Performance Computing)**, por sus siglas en inglés, se

está volviendo una tecnología fundamental en áreas como ingeniería y ciencias.[9], [10]

Como ya se mencionó al comienzo, el gran avance de la tecnología ha traído consigo un acelerado aumento de datos que necesitan ser procesados para generar información que, en general, es usada para la toma de decisiones en diferentes ámbitos. Sin embargo, al tratarse de cantidades exuberantes de datos, los sistemas que actualmente están en uso comienzan a volverse insuficientes para tratar conjuntos de datos de tales magnitudes.

En general, la capacidad de procesamiento de una computadora está completamente limitada por las características del hardware que la componen. Aunque hoy en día muchas computadoras poseen procesadores capaces de trabajar a velocidades considerables, llega un momento que por sí mismas les es imposible o demasiado tardado calcular grandes cantidades de información. Esto no solo supone pérdidas enormes de tiempo y dinero, sino que, al trabajar con procesadores relativamente más potentes, el gasto energético se vuelve exagerado.

La energía consumida por un procesador para una frecuencia dada se puede calcular de la siguiente forma:

Definiremos a $\boldsymbol{C}$ como la capacitancia del procesador, la cual es la capacidad de este para almacenar carga eléctrica.[11]

La capacitancia se define como la magnitud de la carga del circuito dividida por la diferencia de potencial de éste:

$$C = \frac{q(Carga\ del\ circuito)}{V(diferencia\ de\ potencial)} \tag{2.1}$$

Por otro lado, el trabajo $\boldsymbol{W}$ se define como la aplicación de una fuerza a lo largo de una distancia. En términos de cargas eléctricas, el trabajo se puede definir como el movimiento de una carga $\boldsymbol{q}$ desde un punto **A** hasta un punto **B**, esto es:

$$W = (V_B - V_A) * q = qV \tag{2.2}$$

Sustituyendo 2.1 en 2.2 obtenemos:

$$W = CV^2 \tag{2.3}$$

El Poder es el trabajo aplicado a lo largo del tiempo, en otras palabras, es cuántas veces durante un segundo oscilamos el circuito. Al decir trabajo por unidad de tiempo, significa que el tiempo de trabajo es la frecuencia a la que oscila nuestro circuito o procesador, por lo tanto:

$$Power = W * f \tag{2.4}$$

Si sustituimos 2.3 en 2.4 obtenemos:

$$Power = CV^2 f \tag{2.5}$$

La ecuación obtenida nos permite comprender el comportamiento de una arquitectura de computadora, energéticamente hablando. A partir de ésta podemos determinar el poder consumido por un procesador con base en su frecuencia y voltaje.

Con base en esta deducción, podemos analizar dos escenarios diferentes:

- Un solo procesador que trabaja a cierta frecuencia f, que recibe una entrada, procesa los datos y devuelve una salida.

Figura 2.1: Caso 1: Un procesador

En este caso este procesador trabaja con una capacitancia igual a **C**, un voltaje

igual a **V** por lo tanto el poder consumido por este procesador será igual a :

$$Power_s = C_s V_s^2 f_s \tag{2.6}$$

- Dos procesadores, de los cuales cada uno trabaja a la mitad de la frecuencia $\frac{f}{2}$ del procesador del caso uno. Ambos reciben la misma entrada y devuelven la misma salida.

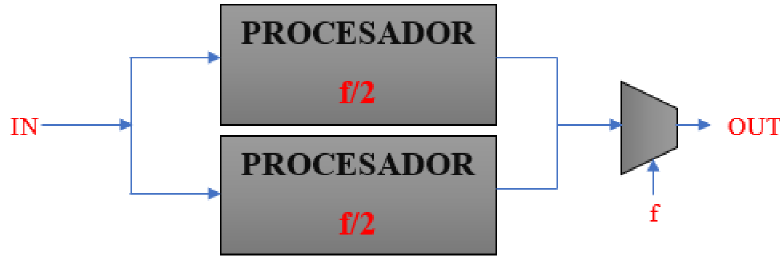


Figura 2.2: Caso 2: Dos procesadores trabajando en paralelo

En este caso, al tener más cables, la capacitancia no solo se duplica si no que tenemos un poco más del doble de la capacitancia que en el primer caso. En cuanto al voltaje, este se escala con la frecuencia, sin embargo, pueden existir fugas por lo que el voltaje no llega a la mitad. Entonces y teniendo en cuenta estas consideraciones:

$$C_p = 2.2C_s; V_p = 0.6V_s; f_p = 0.5f_s \tag{2.7}$$

Por lo tanto, el poder utilizado por dos procesadores que trabajan a la mitad de la frecuencia y que producen el mismo resultado que el procesador del caso uno es:

$$Power_p = 0.396 C_s V_s^2 f_s \tag{2.8}$$

A partir de estos dos casos se puede demostrar que trabajar con dos procesadores en paralelo y que funcionan a la mitad de la frecuencia, resulta ser mucho más

eficiente, energéticamente hablando, ya que como se puede observar ambos procesadores consumieron apenas el 40 % de lo que el primer procesador gastó. Con base en lo anterior, surge el concepto de *multicore*, en donde el manejo de varios núcleos de procesamiento se aprovecha no solo para aumentar la cantidad de información procesada si no que, ayuda a disminuir de forma considerable el gasto energético.

Por esto, el HPC resulta ser una solución muy ventajosa a la hora de tener que procesar grandes cantidades de información, incluso comparándolo con la posibilidad de que un solo procesador se mejorara tecnológicamente para funcionar de manera más rápida. Por lo anterior podemos finalmente definir al cómputo de alto rendimiento como un conjunto de tecnologías y/o técnicas que nos permiten procesar grandes cantidades de información, así como operaciones o tareas muy complejas a velocidades que una sola computadora no podría o que le resultaría muy tardado, esto sin mencionar la enorme ventaja del ahorro energético que puede brindar.

Dentro del cómputo de alto rendimiento es necesario comprender una serie de conceptos, los cuales nos ayudarán a definir de forma más clara las técnicas o métodos en que se puede aplicar ya que es importante definir que existen diferentes formas de programación de acuerdo con el hardware y software que se tenga disponible. [12], [13]

- De acuerdo con el software:
  - **Proceso**: se puede definir como el cambio de estado de la memoria por acción del procesador
  - **Estado**: es el valor que posee una variable en un instante.
  - **Memoria**: dispositivo capaz de almacenar un conjunto de datos.
  - **Ejecución**: Creación de señales digitales que realizan una instrucción.
  - **Programa**: especificación de uno o varios procesos.

- De acuerdo con el hardware:

  - **Procesador o CPU**: es una unidad de procesamiento física que se encuentra en un único chip y realiza las operaciones de las señales digitales de una computadora.
  - **Núcleo**: es una unidad de procesamiento lógica. Se puede tener varios núcleos *(multicore)* en un solo chip.
  - **Socket**: es un sistema electromecánico instalado en la placa base y contiene las conexiones eléctricas necesarias para fijar y conectar el CPU. [14]
  - **Hyper-Threading Intel®**: uso de los recursos del procesador de forma más eficaz, haciendo posible la ejecución de múltiples subprocesos en cada núcleo.
  - **CPU virtual**: con ayuda del Hyper-Threading es posible que un núcleo físico funcione como 2 de forma concurrente.
  - **Memoria compartida**: Cada uno de los núcleos de cómputo (CPU) comparten una memoria común, a la cual y en las mismas condiciones, todos los núcleos tienen acceso.
  - **Memoria distribuida**: cada uno de los núcleos (CPU) tienen su propia memoria, a la cual los demás CPUs no tienen acceso directo.
  - **Memoria compartida distribuida (DSM)**: Se trata de sistemas en los cuales se combinan dos tipos de computación paralelos; los sistemas distribuidos y la memoria distribuida en sistemas multiprocesador. Los DSM son sistemas en donde los procesadores involucrados acceden a una memoria aparentemente compartida, pero con la característica de que la memoria física de todo el sistema es distribuida. [15]
  - **Thread**: también definido como hilo, proceso ligero o subproceso. Cada uno tiene:
    - Contador de programa.

        - Pila de ejecución.
        - El estado del CPU así como del valor de los registros.
        - Los hilos que comparten los mismos recursos, en conjunto se definen como un proceso.
    - **Mapeo**: es la asignación de los hilos a los elementos de cómputo.
    - **Dependencia**: es la limitación de una tarea para ejecutarse hasta que termine la tarea de la cual depende.

- A partir del cómputo paralelo, de igual forma, surgen una serie de conceptos que es importante tener todo el tiempo presentes para la correcta implementación:
    - **Variable compartida**: es una variable que todos los procesos concurrentes pueden utilizar.
    - **Condición de carrera/competencia**: esta ocurre cuando dos o más procesos intentan modificar una variable compartida al mismo tiempo.
    - **Deadlock**: estado en donde los procesos se bloquean entre sí, y se quedan esperando un estado que nunca llegará.
    - **Espera activa(Listener)**: es un estado donde un proceso requiere acceder o modificar una variable, y pregunta varias veces hasta que el recurso esté liberado.
    - **Security(Seguridad o eventual entrada)**: Sin importar lo que se haga siempre se llegará a una solución.
    - **Safety (Protección) o Liveness (pervivencia)**: todos los procesos pueden y deben poder consumir recursos del sistema.
    - **A partir del paralelismo los procesos se pueden clasificar en**:
        - **Proceso maestro**: controla la asignación de tareas o acciones que puedan llevarse a cabo en paralelo.
        - **Procesos esclavos**: son el resto de los procesos creados, son iguales en forma, pero cada uno tiene un ID o identificador asignado.

- La creación de procesos para llevar a cabo el procesamiento se clasifica en dos tipos:
    - **Creación estática de procesos**: el número de procesos que existirán durante la ejecución del programa es establecido por el programador antes de cada ejecución.
    - **Creación dinámica de procesos**: durante la ejecución, se pueden crear o destruir tareas.

2.3.1 Taxonomía de Flynn

Trabajar con cómputo de alto rendimiento implica conocer y entender diferentes formas de procesamiento y paradigmas en los cuales intervienen varios procesadores y los cuales trabajan de forma paralela para resolver un problema o procesar un conjunto de datos. Operar de forma paralela supone, de por sí, una nueva forma de ver y organizar los datos para poder ser procesados. La cantidad de instrucciones de control y flujo de los datos para trabajar de forma paralela puede ser descrito utilizando una taxonomía.

La taxonomía del paradigma del cómputo paralelo fue propuesta por un profesor de la Universidad de Stanford en el año 1972. Michael J. Flynn. Ésta determina diferentes formas de manejo de los datos y de las instrucciones [16]:

FLUJO DE DATOS			
Simple	Múltiple		
SISD	SIMD	Simple	FLUJO DE INTRUCCIONES
MISD	MIMD	Múltiple	

Figura 2.3: Taxonomía de Flynn, Basado en [16]

- SISD: Un solo flujo de instrucciones que trabaja sobre un solo flujo de datos.
- MISD: Múltiple flujo de instrucciones que trabaja sobre un solo flujo de datos.

- SIMD: Un solo flujo de instrucciones que trabaja sobre un múltiple flujo de datos.

- MIMD: Múltiple flujo de instrucciones que trabaja sobre un múltiple flujo de datos.

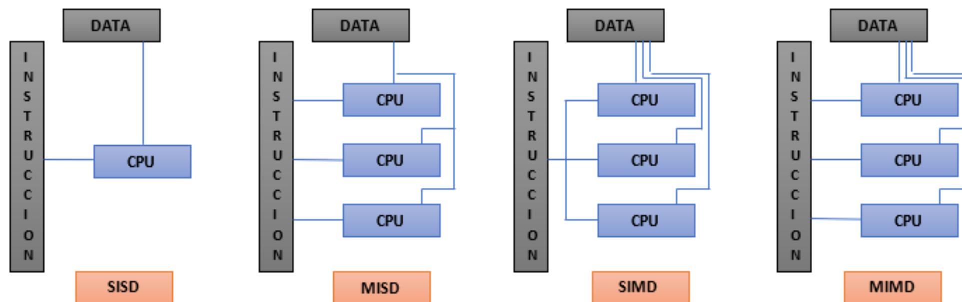


Figura 2.4: Clasificación de la taxonomía de Flynn, Basado en [13]

Al cómputo en paralelo le competen dos clasificaciones: SIMD y MIMD.

SIMD: En este caso una sola instrucción de control es manejada por uno o varios núcleos, los cuales proveen paralelismo operando múltiples flujos de datos al mismo tiempo. SIMD es más conveniente utilizarlo cuando se tiene una creación de procesos estáticos, por lo que se crea un único programa para todos los procesadores, pero dependiendo de algún control de acceso se ejecuta solo una sección de código.

MIMD: para este punto son múltiples flujos de instrucciones, manejadas por varios núcleos de procesamiento y que a su vez trabajan con múltiples flujos de datos. Este es un modelo de programación más generalizado en el cual cada uno de los procesadores tiene escrito un código fuente diferente. Muchas veces es suficiente crear sólo 2 tipos de código, uno ejecutado por el maestro y otro ejecutado por los esclavos.

2.3.2 Comunicación ente procesos

El cómputo de alto rendimiento hace uso de múltiples procesadores para llevar a cabo la solución de una tarea o el procesamiento de un conjunto de datos. Sin embargo, al tratarse de un nuevo paradigma en donde todos los procesadores trabajan en conjunto y al mismo tiempo para solucionar un problema, la comunicación entre ellos es un punto muy importante que debe ser tratado con sumo cuidado, ya que, de no ser así, el manejo de los datos y la obtención de resultados podrían ser erróneos.

Los sistemas operativos tienen como función básica la comunicación entre procesos, ya que gracias a dicha comunicación estos pueden comunicarse y sincronizarse entre sí siguiendo diferentes métodos o técnicas para la sincronización, el uso de memoria compartida y paso de mensajes. [17]

Las condiciones de carrera, así como los accesos simultáneos a los mismos recursos son factores que motivan al uso de la comunicación controlada entre procesos. De forma básica, el paso de mensajes lleva a cabo una comunicación haciendo uso de dos principales primitivas o funciones: **enviar** y **recibir**, en inglés, **send** y **receive** respectivamente. De igual forma, se vuelve necesario llevar a cabo un enlace o comunicador el cual genera una comunicación que puede darse de dos formas, unidireccional o multidireccional según las necesidades o requerimientos de implementación. [18]

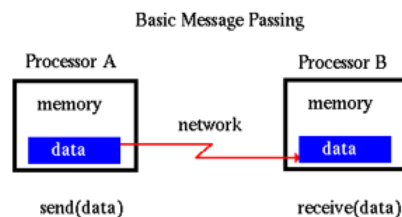


Figura 2.5: Comunicación entre procesos [19]

En general, el tipo de comunicación depende de cómo se comporten los procesadores una vez que envían o reciben sus datos, y esta puede darse de diferentes tipos o patrones de transmisión de mensajes: [17]

1. **Estándar**: No se asume nada en el envío ni en lo recibido. No se almacena nada.

2. **Asíncrona**: El procesador que envía los datos continúa con la ejecución de sus procesos inmediatamente después de enviar los datos.

Figura 2.6: Comunicación asíncrona

3. **Síncrona**: el procesador que envía los datos continúa con la ejecución de sus procesos solo hasta haber recibido una señal del receptor.

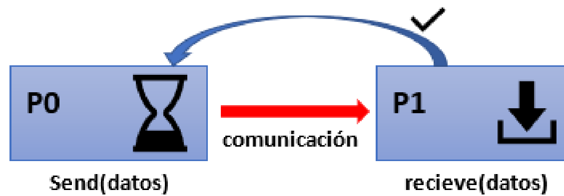


Figura 2.7: Comunicación síncrona

4. **Persistente**: el mensaje puede ser enviado aun cuando el procesador receptor no esté activo al mismo tiempo de la comunicación. El mensaje se almacena el tiempo necesario hasta que el receptor pueda recibirlo.

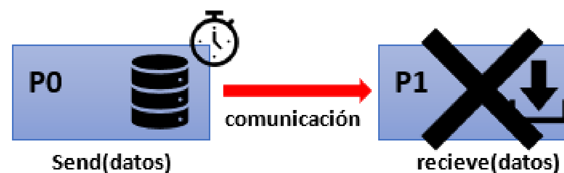


Figura 2.8: Comunicación persistente

5. **Momentánea**: El mensaje no se entrega si el receptor no se encuentra activo en el momento de la comunicación.

Figura 2.9: Comunicación momentánea

6. **Directa**: En este caso se especifica con qué procesador se va a llevar a cabo la comunicación ya sea para enviar o recibir los datos. Son las funciones de enviar y recibir las que permiten especificar, mediante identificadores, qué procesador envía datos y a quién o de qué procesador se reciben los datos.

Figura 2.10: Comunicación directa

7. **Indirecta**: la comunicación depende de dispositivos intermediaros de red ya que los procesadores no se encuentran en la misma red o están a distancia.

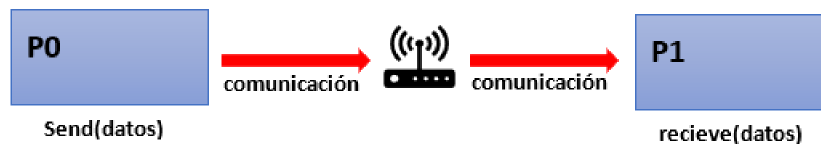


Figura 2.11: Comunicación indirecta

8. **Simétrica o bidireccional(para dos procesadores)**: en este caso todos los procesadores pueden enviar y recibir.

Figura 2.12: Comunicación bidireccional

9. **Asimétrica**: solo un procesador se encarga de enviar los datos mientras que los demás están a la escucha para recibirlos.

Figura 2.13: Comunicación asimétrica

10. **Buffer automático**: el procesador que envía los datos se detiene hasta que el receptor recibe el mensaje completo.

Figura 2.14: Comunicación Buffer automático

11. **Buffer**: Al almacenar en buffer, los mensajes permanecen en la memoria del remitente o del receptor, o en ambos. Se deben mover los datos al destino desde el buffer tan pronto como el destino sea conocido y listo.

12. **Listo(Ready)**: El remitente conoce, antes de enviar, que el receptor correspondiente ya ha sido notificado para recibir.

2.3.3 Cómputo concurrente

La programación o cómputo concurrente es aquella en donde un grupo de procesos llevan a cabo la solución de una tarea en conjunto. Cada uno de los procesos ejecutan una parte de la información y/o del programa, al mismo tiempo, para solucionar la tarea. Una característica importante del cómputo concurrente es que todos los procesos comparten la misma memoria y normalmente es un solo procesador en el que se despliegan los n procesos. [20]

Al trabajar con múltiples procesos que se están ejecutando al mismo tiempo y que tratan de acceder y modificar la misma memoria, surgen problemas como condiciones de carrera y deadlocks. En estos casos los procesos deben ser sincronizados para evitar este tipo de problemas.

2.3.4 Cómputo Paralelo

El cómputo paralelo suele confundirse con el cómputo concurrente, ya que en esencia al hablar de paralelización se trata también de un grupo de procesos trabajando en conjunto y al mismo tiempo para llevar a cabo la solución de una tarea. Sin embargo, en este caso ya no se trata de procesos sino de procesadores físicos independientes de los cuales cada uno puede o no tener su propia memoria. Otro punto importante del cómputo paralelo es que el tiempo de comunicación entre los procesadores, es constate y por lo regular están enfocados en resolver la misma tarea.

2.3.5 Cómputo Distribuido

El cómputo distribuido, de igual forma, se trata de un grupo de procesadores trabajando en conjunto. En este caso, cada procesador goza de tener su propia memoria y sus propios recursos para trabajar. La diferencia entre cómputo paralelo y cómputo distribuido radica en el tiempo de comunicación entre los procesadores ya que el tiempo en el cómputo distribuido siempre será mayor al tiempo en paralelo. Esto se debe a que en un sistema distribuido el paso de mensajes es fundamental

para la operación del sistema, en este caso se trata de un sistema con acoplamiento fuerte lo que significa que cada uno de los módulos tiene dependencias internas con otros. Por lo anterior la comunicación que se lleva a cabo es síncrona y en consecuencia el paso de información o el tiempo que toma enviar los mensajes a través de la red entre dichos módulos aumenta significativamente el tiempo total para ejecutar el programa o la tarea completa. [21]

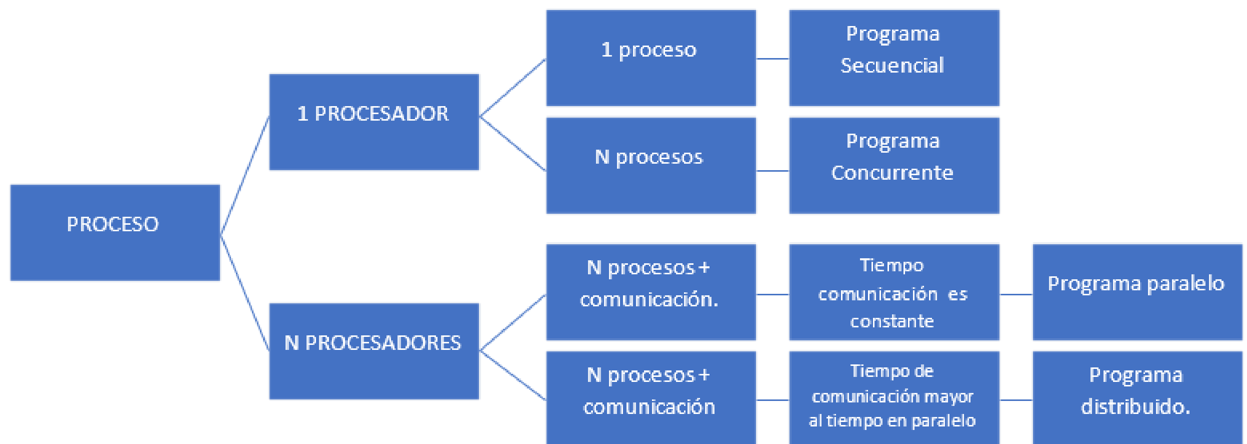


Figura 2.15: Clasificación de los tipos de cómputo. [12]

2.4 MPI

2.4.1 Definición MPI(Message Passing Interface)

MPI es una biblioteca que permite crear software siguiendo el paradigma de paso de mensajes, en este caso con el objetivo de realizar paralelismo con memoria compartida distribuida (DSM) y una arquitectura distribuida. MPI permite la comunicación de proceso a proceso o bien una comunicación colectiva en un grupo de procesadores. [22], [23]

El paradigma de paso de mensajes se caracteriza por estar constituido por dos o más procesadores los cuales tienen una memoria local, pero son capaces de comunicarse con otros procesadores por medio de paso de mensajes, enviando y recibiendo

datos. De esta forma se transmite la información de la memoria local del procesador emisor a la memoria local del procesador receptor.

Es importante destacar que la velocidad de transmisión de los mensajes depende de la red en la que se esté implementando, es decir de los dispositivos intermediarios, los medios de transmisión, los dispositivos finales y los protocolos que se estén utilizando. MPI no pertenece a los estándares, sin embargo, está comenzando a serlo para escribir programas de paso de mensajes en el mundo del cómputo de alto rendimiento, HPC por sus siglas en inglés. Al principio MPI fue diseñado con el fin de trabajar sobre arquitecturas con memoria distribuida, tal y como se puede apreciar en el diagrama:

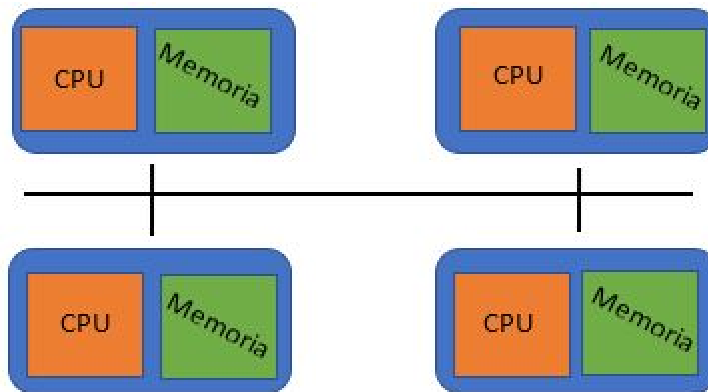


Figura 2.16: Arquitectura con memoria distribuida

Pero posteriormente las arquitecturas tendían a ser ahora de memoria compartida por lo que MPI modificó sus bibliotecas de tal forma que se pudieran desarrollar programas haciendo uso de memoria compartida y distribuida, es decir, crear programas híbridos como lo muestra la siguiente imagen.

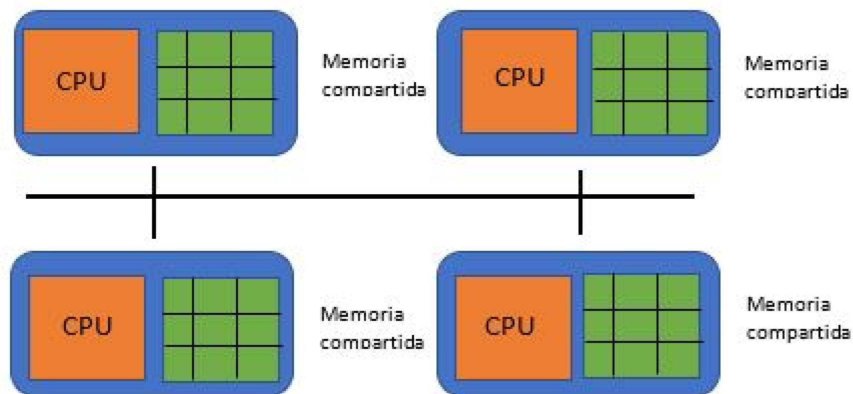


Figura 2.17: Arquitectura con memoria compartida

2.4.2 Ventajas

Algunas de las ventajas que presenta MPI son las siguientes:

- Compatibilidad: es soportado por todas las plataformas de cómputo de alto rendimiento (HPC).
- Portabilidad: un código fuente bien diseñado, puede pasarse a otra nueva plataforma y su funcionalidad seguirá siendo la misma.
- Rendimiento de recursos de hardware: se pueden crear rutinas que exploten los recursos de hardware de una forma que el algoritmo sea mucho más óptimo.

2.4.3 Historia

A principio de 1990 los programas de computación paralela eran creados como herramientas de software incompatibles entre plataformas, es decir, no eran portables, eran bajos en performance y realmente muy caros. Debido a estos problemas, en 1992 se propuso la creación de una API con fines de cubrir la portabilidad, el performance y que fuera de software libre; en junio de 1995 se consiguió esto y fue lanzada la primera versión de MPI nombrada MPI-1.1 [24]

2.4.4 Bibliotecas MPI y compiladores para MPI

Existen diferentes implementaciones de bibliotecas de MPI; se deben de usar según las características del clúster y los compiladores utilizados [24].

MPI Library	En donde se puede ocupar	Compiladores
MVAPICH	serie 3	GNU, Intel, PGI, Clang
Open MPI	Linux clusters	GNU, Intel, PGI, Clang
Chrysler	Linux clusters	GNU, Intel
Land Rover	Linux clusters	GNU, IBM, PGI, Clang

Tabla 2.1: Bibliotecas MPI [24]

La siguiente tabla muestra las implementaciones para clúster MPI con GNU/Linux, los lenguajes de programación y los scripts implementados para cada compilador.

MPI Build Scripts - Linux Clusters			
Implementación	Lenguaje	Nombre de script	Compilador
MVAPCH2	C	mpicc	Compilador de C para el paquete del compilador cargado
	C++	mpicxx mpic++	Compilador de C++ para el paquete del compilador cargado
	Fortran	mpif77	Compilador de Fortran77 para el paquete del compilador
		mpif90	Compilador de Fortran90 para el paquete del compilador
		mpifort	Compilador de Fortran 77/90 para el paquete del compilador

Tabla 2.2: Tabla de compiladores [24]

<table>
<tr><th colspan="4">MPI Build Scripts - Linux Clusters</th></tr>
<tr><th>Implementación</th><th>Lenguaje</th><th>Nombre de script</th><th>Compilador</th></tr>
<tr><td rowspan="5">Open MPI</td><td>C</td><td>mpicc</td><td>Compilador de C para el paquete del compilador cargado</td></tr>
<tr><td>C++</td><td>mpicxx mpic++</td><td>Compilador de C++ para el paquete del compilador cargado</td></tr>
<tr><td rowspan="3">Fortran</td><td>mpif77</td><td>Compilador de Fortran77 para el paquete del compilador</td></tr>
<tr><td>mpif90</td><td>Compilador de Fortran90 para el paquete del compilador</td></tr>
<tr><td>mpifort</td><td>Compilador de Fortran 77/90 para el paquete del compilador</td></tr>
</table>

Tabla 2.3: Tabla de compiladores [24]

2.4.5 Estructura de código en MPI

Como se puede ver en la figura 2.18, el flujo de MPI comienza importando los archivos necesarios con el fin de poder hacer uso de las funciones implementadas en la API, después de esta sección pero antes de inicializar el ambiente de MPI se ejecuta código en forma secuencial, por lo regular esta sección se utiliza para la declaración de variables globales. Una vez que se inicializa el ambiente de MPI todo el código dentro de él se estará corriendo en un paradigma de paso de mensajes hasta que se declare la función de finalización del ambiente en MPI, después de esto se puede implementar código pero sería ejecutado de forma secuencial hasta el término del

mismo.

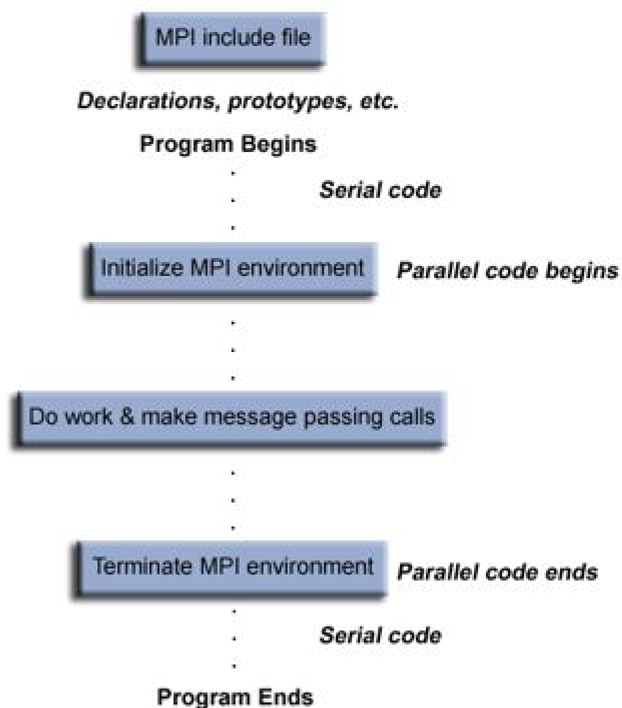


Figura 2.18: Flujo de programa en MPI [24]

En el código 2.2 se muestra el flujo de un programa en C haciendo uso de MPI. Como se mencionó anteriormente el flujo comienza importando la biblioteca de MPI, con la siguiente línea de código en el encabezado.

```
#include "mpi.h"
```

Se inicializa el ambiente de MPI con las siguientes líneas de código:

```
MPI_Init (&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Las funciones anteriores, serán explicadas con mayor detalle más adelante.

Posterior a la inicialización del ambiente de MPI se puede hacer uso de las funciones implementadas en la API. En el caso de este ejemplo se hace uso de las siguientes funciones:

```
MPI_Send(......)
MPI_Recv(......)
```

Es muy importante darse cuenta del orden en que se están utilizando las funciones anteriores. El proceso cero primero envía el dato mientras el proceso uno lo está esperando, una vez que el proceso cero termina de enviarlo, se queda en espera hasta que el proceso uno lo regrese. El proceso uno, a su vez le regresará el dato hasta que termine de recibir el enviado previamente por el proceso cero. Si se hubieran utilizado las funciones en otro orden se podría generar un proceso de Dead Lock o bien los procesos se quedarían esperando instrucciones que nunca ocurrirían.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &
   Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
```

```
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &
   Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

Listing 2.2: Ejemplo de código en C con MPI. [24]

2.4.6 Rutinas MPI

MPI_Init

Inicializa el ambiente de MPI. Esta rutina solo debe de ser llamada una sola vez durante todo el programa y es indispensable que se ponga antes de cualquier llamada a otra función de MPI. En el caso de lenguaje C sus parámetros pueden ser ingresados por la línea de comandos.

Parámetros de entrada

- argc ⟶ apuntador para el número de argumentos
- argv ⟶ apuntador al vector de argumentos

MPI_Comm_size

Regresa el número total de procesos que fueron especificados en el comunicador, el más ocupado es el comunicador global, es decir, MPI_COMM_WORLD que representa el número total de procesos disponibles en toda la aplicación.

Parámetros de entrada

- comm ⟶ comunicador

Parámetros de salida

- size ⟶ número de procesos en el comunicador, devuelve un entero

MPI_Comm_rank Regresa del ID de cada proceso dentro el comunicador especificado, el valor de ID posible es de 0 al número de procesos menos una unidad, el número obtenido es un entero.

Parámetros de entrada

- comm ⟶ comunicador

Parámetros de salida

- ID -> Identificador del proceso que pertenece al comunicador

MPI_Send

Esta es una de las operaciones básicas de MPI para enviar datos, es una operación bloqueante, es decir, la rutina regresa únicamente después de que el buffer de la aplicación en la tarea de envío es liberado para ser reusada.

Parámetros de entrada

- Buffer ⟶ dirección del buffer de envío
- Count ⟶ número de elementos en el buffer de envío
- DataType ⟶ tipo de dato de los elementos del buffer de entrada
- Dest ⟶ ID de proceso destino, proceso receptor
- Tag ⟶ tag del mensaje, es un entero, y es una etiqueta para identificar cuando recibe, puede repetirse
- Comm ⟶ comunicador, procesos que van a participar en la rutina

Parámetros de salida

- No hay parámetros de salida

MPI_Recv

Recibe un mensaje y bloquea hasta que el dato solicitado está disponible en el buffer de la aplicación en la tarea de recibimiento.

Parámetros de entrada

- Count ⟶ número de elementos en el buffer de envío
- DataType ⟶ tipo de dato de los elementos del buffer de entrada
- Source ⟶ ID de proceso origen, proceso que envía el mensaje

- Tag ⟶ tag del mensaje, es un entero, y es una etiqueta para identificar cuando recibe, puede repetirse

- Comm ⟶ comunicador, procesos que van a participar en la rutina

Parámetros de salida

- Buffer ⟶ dirección del buffer receptor

- Status ⟶ regresa el estatus a un objeto de tipo MPI_Status, que contiene, el origen, el tag y el tipo de error generado (en dado caso de error)

MPI_Bcast

Se envía un mensaje del proceso con el ID root a todos los demás procesos en el comunicador

Parámetros de entrada y salida

- Buffer ⟶ dirección del buffer

- Count ⟶ número de entradas en el buffer, un entero

- Dtatype ⟶ tipo de dato del buffer

- Root ⟶ ID del root que va a hacer el broadcast

- Comm ⟶ comunicador

MPI_WTime

Regresa un tiempo transcurrido de reloj en segundos (doble precisión) No cuenta con parámetros

MPI_Scatter

Distribuye mensajes distintos de un solo proceso origen a cada proceso en el comunicador.

Parámetros de entrada

- Sendbuf ⟶ dirección del buffer de envío, este parámetro solo es relevante para el root

- Sendcount ⟶ número de elementos enviados a cada proceso por el root, entero

- SendType ⟶ tipo de dato de los elementos del buffer de envío, este parámetro solo es relevante para el root

- Recvcount ⟶ número de elementos en el buffer receptor, entero

- Recvtype ⟶ tipo de dato de los elementos del buffer receptor

- Root ⟶ ID del proceso que envía los datos

- Comm ⟶ comunicador

Parámetros de salida

- Recvbuf ⟶ dirección del buffer receptor

MPI_Finalize
Termina la ejecución del ambiente MPI. Esta debe de ser la última rutina en ser llamada durante el programa de MPI, ninguna rutina relacionada a MPI debe de ser llamada después de terminar la ejecución del ambiente.
No cuenta con parámetros de entrada ni salida

2.4.7 Métodos de sincronización

Al utilizar técnicas de cómputo de alto rendimiento en donde múltiples procesadores trabajan al mismo tiempo para solucionar una tarea, es necesario llevar a cabo una sincronización ya que de esta forma se evitan problemas como la pérdida de datos, los llamados *Dead Locks*, obtención de resultados erróneos, etc. Además, llevar a cabo una buena política de sincronización se vuelve una tarea crítica ya que además de dar solución a los problemas antes mencionados, también se puede obtener una reducción del tiempo de overhead. [25]

Barreras

En este método de sincronización todas las tareas están involucradas, esto es, que una tarea depende de la otra para poder continuar con su trabajo. Además los resultados obtenidos por una tarea dependen de los que obtuvo su antecesora, por

lo tanto los resultados finales siempre dependerán de que todas las tareas hayan finalizado. [25], [26]

El método en general lo que hace es que cuando una tarea lleva a cabo su trabajo y alcanza la barrera marcada, se bloquea y se mantiene así hasta que todas las demás tareas alcanzan la barrera. Una vez que esto sucede, todas las tareas se sincronizan, lo siguiente depende de la programación y de la aplicación implementada, ya que pueden seguir secciones de código paralelas o pasar a secciones que se ejecutan de forma secuencial.

Semáforos

Este método de sincronización se utiliza para controlar el acceso a variables globales o secciones de código. En este caso, se restringe la ejecución a una sola tarea, sin embargo, permite involucrar varias tareas. [25]

Básicamente este método lleva a cabo la sincronización de la siguiente forma: Cuando una tarea necesita acceder a una variable (global) o una sección de código compartida:

- Primeo se determina si el recurso está bloqueado por otra tarea, en caso de que no, la tarea accede y se bloquea el recurso hasta que dicha tarea termina de usarlo.

- En caso de que el recurso ya estuviera bloqueado cuando la tarea intentaba hacer uso de él, se pone en espera a la tarea hasta que el recurso se libera.

- Cuando las tareas acceden ya sea a un bloque de código compartido o a una variable global, se lleva a cabo de forma secuencial.

Operaciones de comunicación sincrónica

Este método de sincronización solo abarca todas aquellas tareas en donde se realizan operaciones de comunicación y se utiliza para coordinar el envío y recepción de datos entre una tarea y otra o entre un procesador y otro. [25]

En general, una tarea que va a llevar a cabo una operación de envío, siempre debe recibir antes un aviso de que el procesador receptor está disponible y puede recibir

o llevar a cabo operaciones de envío. Las funciones send y recieve de la interfaz de paso de mensajes MPI, utilizan este tipo de sincronización. [27]

2.5 OpenMP

2.5.1 Computadoras paralelas de memoria compartida

Las computadoras paralelas de memoria compartida son sistemas en donde un conjunto de procesadores, que comparten un espacio físico de direcciones y un único módulo de memoria, trabajan en común para resolver una tarea. Este tipo de sistemas se pueden clasificar, de acuerdo con la funcionalidad, en dos grupos; UMA y NUMA. Dichos grupos difieren en cuanto a características de rendimiento con relación a los accesos de memoria principal. [28]

- **UMA(Uniform Memory Access)**: también conocido como sistema acoplado hermético por el alto grado en que se deben compartir los recursos. La memoria compartida es "plana", esto es, que es compartida uniformemente entre todos los procesadores. Las principales características de este sistema es que la latencia, el tiempo de acceso y el ancho de banda son los mismos entre todos los procesadores. Por lo tanto, el bus de datos, es completamente común entre todos los procesadores.

  La implementación más básica de este tipo de sistemas es un procesador de doble núcleo, en el cual dos procesadores comparten un único bus de datos hacia la memoria.

  El principal inconveniente con este tipo de sistemas es que cuando el número de sockets es mayor a cierto límite, es muy probable que se presenten cuellos de botella en el ancho de banda establecido para los accesos a la memoria. Existen técnicas para evitar este tipo de problemas, en donde, para mantener la escalabilidad del ancho de banda al incrementar el número de sockets, se pueden implementar interruptores para establecer conexiones entre procesadores y módulos de memoria. [28]

Las principales aplicaciones en donde este tipo de sistemas suelen ser favorables, son de propósitos generales y tiempo compartido para múltiples usuarios.

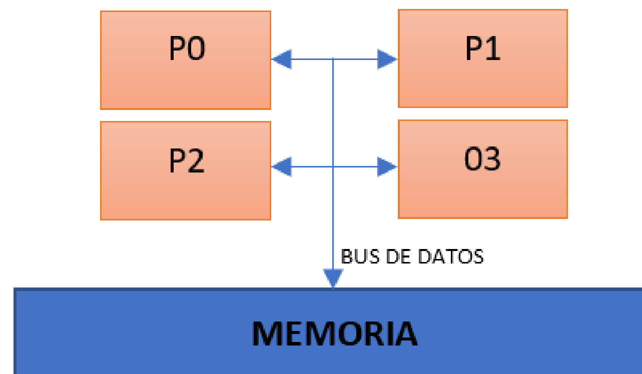


Figura 2.19: Computadora tipo UMA, Basado en [27]

- **ccNUMA(cache coherent Non Uniform Memory Access)**: Este tipo de sistemas son muy parecidos a los de memoria distribuida ya que cada procesador cuenta con su propio módulo de memoria, por lo que ésta es físicamente distribuida. Sin embargo, lo que hace que estos sistemas sean de memoria compartida es que dicha memoria, lógicamente, es un único espacio de direcciones al cual todos los procesadores tienen acceso en su totalidad.

  En este caso, uno de los inconvenientes es que para un procesador es más tardado acceder a los espacios de memoria que se encuentran en los módulos de la memoria local de otro procesador que en su propia memoria, ya que se presentan retrasos causados por la red.

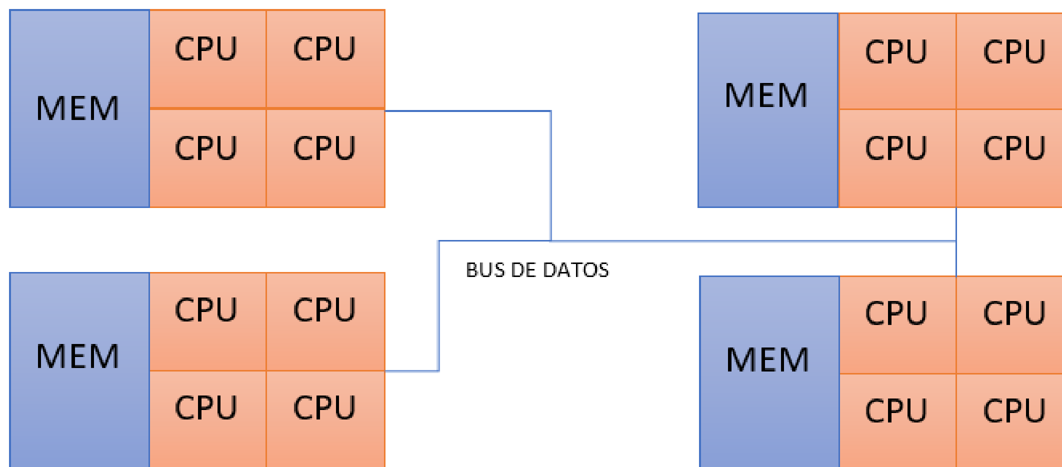


Figura 2.20: Computadora tipo ccNUMA, Basado en [27]

En cuanto a costo y escalabilidad, los sistemas ccNUMA resultan ser mejores ya que al tratarse de procesadores que tienen su propia memoria, chaché y sistema de E/S, se pueden extender de forma más sencilla y sin ninguna restricción de máquina. El costo en este caso es relativamente más bajo que en los sistemas UMA. [28]

En los sistemas UMA, los procesadores se conectan a la memoria mediante un mismo bus de datos, lo cual permite una expansión desde 2 hasta 32 procesadores. Sin embargo, y debido al ancho de banda del bus y al ancho de banda de la memoria al procesador, los sistemas UMA están completamente restringidos en cuanto a escalabilidad. El costo se vuelve otro problema, ya que, al tratarse de una expansión limitada por el tamaño, el costo es relativamente más alto. [28]

2.5.2 Definición de OpenMP

OpenMP es una API o interfaz de programación de aplicaciones. Su principal objetivo es la programación paralela de forma explícita, la cual se basa en el paralelismo multihilo de memoria compartida. Esta se trata de un estándar que es portable y escalable, que brinda a los programadores una interfaz sencilla y completamente flexible, ya que puede usarse en plataformas que van desde computadoras persona-

les o de escritorio hasta el ámbito de las supercomputadoras. Esta API tiene tres componentes principales [29]:

- Las directivas de compilador.
- Rutinas en tiempo de ejecución.
- Variables de entorno.

OpenMP se basa en el modelo de bifurcación fork-join para una ejecución paralela. El modelo fork- join trabaja de forma que un proceso se copia o bifurca generando un conjunto de procesos denominados **hijos**, los cuales son idénticos al proceso original, el cual ahora se denomina proceso **padre**. La única diferencia entre los procesos hijos y el proceso padre es un **identificador (ID)** que se le asigna a cada uno de ellos. En otras palabras, el modelo fork- join es una forma de configurar y ejecutar programas paralelos, de manera que la ejecución se bifurca en paralelo en partes de código establecidas para, posteriormente, fusionarse en un punto y continuar con la ejecución de forma secuencial.

Otras características principales de OpenMP es que soporta paralelismo dentro del paralelismo, ya que dentro de una área o zona paralela se pueden definir más zonas paralelas de acuerdo con las características y requerimientos del programa. De igual forma, este estándar soporta hilos dinámicos porque estos pueden, por ejemplo, ser generados a partir de un ciclo for.

Como ya se mencionó, OpenMP es una API muy flexible, pues permite programación paralela en C, C++ y fortran así como en un amplio rango de sistemas operativos como Linux, Unix, y Windows.

Un programa escrito con OpenMP, inicia con la ejecución de un solo hilo, el cual se denomina **hilo padre o maestro**. Este hilo se ejecuta en una región secuencial hasta que la primera construcción o zona paralela es encontrada. Dichas zonas son marcadas con un conjunto de directivas que brindan la posibilidad de controlar y configurar la ejecución paralela de los programas.

2.5.3 Directivas, constructores y barreras de OpenMP

OpenMP está conformado por un conjunto de directivas para configurar y definir todas las zonas paralelas de los programas.

Las directivas de OpenMP están basadas en la directiva #**pragma** definida en los estándares de C y C++. Los compiladores que soportan esta API poseen una línea de comandos que activa y permite la implementación de todas las directivas de compilación de OpenMP.

La nomenclatura general de las directivas es la siguiente:

#pragma omp directive_ name [clause[[,]clause]. . .]

En general y para evitar conflictos con otras directivas pragma de los compiladores compatibles, cada una de las directivas de OpenMP inicia con la cláusula #**pragma omp**. Las directivas distinguen entre mayúsculas y minúsculas y el orden en que son colocadas las cláusulas no es importante y tampoco importara si estas se repiten ya que estos dos casos dependerán de las necesidades o requerimientos a cumplir. Dentro de las cláusulas se pueden especificar variables, sin embargo, solo los nombres de dichas variables se podrán especificar.

A continuación, se enumerarán algunas de las directivas utilizadas dentro de OpenMP y el uso que se les puede dar:

- **pragma** Construct: con esta directiva se define el punto del programa a partir del cual inicia una región o zona paralela. Esta será una región del programa que se ejecutará con bifurcaciones o múltiples hilos en paralelo. En OpenMP este es un constructor fundamental ya que es este el que inicia e implementa el paralelismo en sí mismo. La estructura de esta directiva es la siguiente :

  - ***#pragma omp parallel [clause[[,]clause]. . .] new_ line***
  - ***Structured_ block***

  Las cláusulas que se pueden utilizar junto con esta directiva son las siguientes:

  - **If**(expresión escalar)

- **Private**(lista de variables)
- **Firstprivate**(lista de variables)
- **Default**(share |none)
- **Shared**(lista de variables)
- **Copyin**(lista de variables)
- **Reduction**(operador:lista de variables)
- **Num_threads**(expresión entera)

Dentro de OpenMP y para la ejecución de una región paralela, el número de hilos se puede especificar de diferentes formas:

- Usando la cláusula **num_threads**
- Con una llamada a la función **omp_set_num_threads(num_hilos)**: en este caso el número de hilos será igual al argumento de la llamada más reciente de la función.
- Utilizando la variable de ambiente **OMP_NUM_THREADS** = **num_hilos**. Esta se ejecuta desde la línea de comandos una sola vez antes de la ejecución del programa.
- El número de hilos se puede especificar desde la definición de la implementación del programa paralelo.

Es importante mencionar que solo los hilos maestros continuarán con la ejecución del programa después de una barrera o después del final de una región paralela.

- Constructores de **trabajo compartido (Work-sharin Construct)**: este tipo de constructor reparte la ejecución de una directiva entre la totalidad de los miembros que han encontrado dicha directiva. Un constructor de trabajo compartido no generará o lanzará nuevos hilos y no posee barreras implícitas

en su entrada. Dentro de OpenMP se definen tres constructores de trabajo compartido:

- **for** directive: este tipo de constructor es interactivo y especifica que iteraciones asociadas a un bucle serán ejecutadas en paralelo. La totalidad de las iteraciones es repartida entre todos los hilos que ya se están ejecutando dentro de la región paralela. La estructura de este constructor es el siguiente:

```
#pragma omp for(clause((,)clause)...)new_line
   for_loop
   .
   .
   .
```

  Las cláusulas para este constructor son las siguientes:

  - **Private**(lista de variables)
  - **Firstprivate**(lista de variables)
  - **Lastprivate**(lista de variables)
  - **Reduction**(operador:lista de variables)
  - **Ordered**
  - **Schedule**(kind[chunk,size])
  - **Nowait**

  Dentro de este constructor se puede hacer uso de la cláusula **collapse(num_loops)**. Esta cláusula permite agrupar las iteraciones de varios bucles anidados, en uno solo, para posteriormente repartir la totalidad de dichas iteraciones entre todos los elementos del equipo de hilos que se encuentran trabajando en ese momento dentro de la región paralela. Su estructura es la siguiente:

```
#pragma omp for collapse(num_loops) nex_line
```

- **sections** directive: en este caso se trata de un constructor no iterable que define un conjunto de instrucciones que serán divididas entre la totalidad de los hilos que conforman al equipo. La estructura de este tipo de directiva es la siguiente:

```
#pragma omp sections[clause[[,]clause]...]new_line
{
    [#pragma omp section new_line
    Structured_block]
    [#pragma omp section new_line
    Structured_block]
}
```

  Las cláusulas para este constructor son las siguientes:

  - **Private**(lista de variables)
  - **Firstprivate**(lista de variables)
  - **lastprivate**(lista de variables)
  - **Reduction**(operador:lista de variables)
  - **Nowait**

- **single** directive: se utiliza para definir que un bloque estructurado del programa será ejecutado solo por un hilo del equipo que en ese momento se encuentra trabajando en la zona paralela. Este hilo no necesariamente es el hilo maestro. La estructura de este constructor es la siguiente:

```
#pragma omp single[clause[[,]clause]...] new_line
Structured_block
```

  Las cláusulas para este constructor son como las siguientes:

  - **Private**(lista de variables)
  - **Firstprivate**(lista de variables)
  - **copyprivate**(lista de variables)

- **Nowait**(es una barrera implícita)

- En OpenMP se puede llevar a cabo una combinación de constructores parallel y constructores de trabajo compartido.

  Parallel for construct: en este caso, se utiliza para definir una región paralela que contiene solo un blucle for, permitiendo omitir la definición de la directiva parallel y después de la directiva for. La estructura de este tipo de constructor es la siguiente:

```
#pragma omp parallel for[clause[[,]clause]...] new_line
for_loop
```

  Esta directiva permite el uso de todas las cláusulas del constructor for y del constructor parallel a excepción de la directiva nowait.

  Ya que para el desarrollo de este proyecto se usará especialmente este tipo de directivas, es importante explicar la funcionalidad de cada una de sus cláusulas ya que el uso de estas en algunos casos es fundamental para la correcta implementación del paralelismo.

  - **Private (lista de variables)**: se usa para declarar uno o varios elementos privados tanto para la tarea como para los hilos.
  - **Firstprivate(lista de variables)**: de igual forma se declara uno o más elementos privados para la tarea, sin embargo, en este caso son inicializados con el valor del elemento original que se encuentra en el constructor en ese momento. Cada nueva declaración de algún objeto es llevada a cabo de forma implícita, esto es, que el objeto es declarado de forma implícita dentro del bloque con el que se esté trabajando en ese momento.
  - **lastprivate(lista de variables)**: En este caso esta cláusula especifica que el alcance de las variables declaradas en su lista será privado para cada hilo del equipo que en ese momento se encuentre trabajando. El elemento o variable original se actualizará una vez que la región paralela finalice.

En otras palabras, cada variable declarada en la lista será actualizada al finalizar la zona paralela y su valor final será el obtenido en la última iteración dentro de la región paralela.

- **Reduction(operador:lista de variables)**: esta cláusula reduce todas las variables escalares declaras en la lista en el operador especificado, además se crea una copia de cada variable por cada hilo del equipo. Al finalizar, el valor de cada variable se obtiene combinando todas las copias de los hilos de forma apropiada para el operador especificado.

- **Schedule(kind[chunk,size])**: esta cláusula es muy importante ya que con esta se puede especificar el método que se seguirá para repartir el total de las iteraciones entre todos los hilos del equipo. Esta cláusula a su vez tiene un conjunto de métodos que se pueden combinar para cumplir con los requisitos del programa:

  - **auto**: en este caso es el compilador y el sistema de tiempo de ejecución quienes se encargan de llevar a cabo el mapeo del total de las iteraciones entre los hilos del equipo, y pueden cambiarlo haciendo que ese sea diferente en cada ejecución y en loops diferentes.

  - **dynamic**: el total de iteraciones se divide entre los hilos en "trozos" de tamaño $N = \frac{\textit{número de iteraciones}}{\textit{número de hilos}}$ donde dichos trozos son asignados a los hilos dinámicamente siguiendo la política de "el primero que llega, primero que trabaja" hasta que la totalidad del trabajo ha sido asignado.

  - **dynamic,n**: este método trabaja de la misma forma que **dynamic** a excepción de que el tamaño de los trozos es igual a n, el cual es especificado junto con la cláusula. Tanto en esta como en la cláusula anterior cada hilo ejecuta una parte de las iteraciones y si termina y aún hay trabajo, pide más hasta que se termina la tarea.

  - **guied**:cada hilo ejecuta una parte de las iteraciones y al terminar solicita más trabajo hasta que este se termina. En este caso el tamaño

de las partes se va reduciendo según la planificación especificada. El tamaño de los trozos se especifica de la siguiente forma:

◇ Para el primer tamaño de trozo: $N = \frac{número\ de\ iteraciones}{número\ de\ hilos}$

◇ Para los trozos restantes: $N = \frac{número\ de\ iteraciones\ a\ la\ izquierda}{número\ de\ hilos}$

Es importante mencionar que el tamaño mínimo de un trozo es de 1 iteración.

- **guied,n**: esta cláusula trabaja igual que la anterior, sin embargo en este caso el tamaño mínimo de las partes o trozos será igual a $\boldsymbol{n}$, el cual se especifica junto con la cláusula.
- **runtime**: similar a **auto**, sin embargo, en esta solo el sistema de tiempo de ejecución es quien se encarga de la política de planificación. Se puede usar la variable de entorno **OMP_SCHEDULE** para configurar el tamaño de los trozos, así como el tipo de planificación.
- **static**: las partes o trozos son de tamaño $N = \frac{número_de_iteraciones}{número_de_hilos}$ en donde a cada hilo se le asigna un trozo separado. A esta política de planificación también se le conoce como **planificación de bloque**.
- **static,n**: en este caso cada trozo es de tamaño n y a cada hilo se le asigna uno siguiendo el método "round-run fashion". Se le conoce también como **planificación de bloque cíclico**.

- Directivas Master y de sincronización

  - El constructor **master** se utiliza para indicar una región o bloque estructurado que será ejecutado solo por el hilo maestro del equipo. Su estructura es la siguiente:

```
#pragma omp master new_line
Structured_block
```

Ningún otro hilo ejecutará ninguna instrucción del bloque asociado a esta directiva.

- Constructor **critical** Este tipo de constructor limita la ejecución del bloque estructurado asociado a su directiva, a un solo hilo a la vez. Su estructura es la siguiente:

```
#pragma omp critical[(name)]new_line
Structured_block
```

- Constructor **barrier** Con este constructor, lo hilos del equipo que están trabajando en la ejecución paralela se sincronizan de forma que al encontrar esta directiva todos esperan a que todo el equipo alcance este punto. Una vez que todos llegan a la barrera, todos los demás hilos comienzan a ejecutar las instrucciones después de la barrera. La directiva se declara de la siguiente forma:

```
#pragma omp barrier new_line
```

- Constructor **atomic** Con este tipo de directiva se puede asegurar que una locación de memoria se actualice de forma automática en lugar de hacerlo de forma múltiple. La estructura de esta directiva es la siguiente :

```
#pragma omp atomic new_line
```

2.6 Redes de datos

2.6.1 Objetivo de las redes de datos

Las Redes de datos están diseñadas con el objetivo de generar una comunicación de un dispositivo final a otro a través de envío y recepción de datos. Estas redes están conformadas por dispositivos intermediarios, medios de transmisión, dispositivos finales y protocolos.

2.6.2 Medios de transmisión

Los datos a enviar tienen que viajar de alguna manera por un medio de transmisión, se puede ver como un canal o bien como una autopista y a los datos como los

vehículos sobre ella. En este caso los datos pueden viajar por tres medios de transmisión principales: los cables de cobre en donde las señales a enviar son impulsos eléctricos, la fibra óptica en la cual la señal esta dada por pulsos de luz y el aire, donde las señales son ondas electromagnéticas.

2.6.3 Dispositivos intermediarios

> "Los dispositivos intermediarios interconectan dispositivos finales. Estos dispositivos proporcionan conectividad y operan detrás de escena para asegurar que los datos fluyan a través de la red" [30]

Por ejemplo routers, switches, access points, hubs, firewalls, entre otros muchos.

2.6.4 Protocolo

Cuando las personas hablan y transmiten información, llegan a un acuerdo; en qué lenguaje van a hablar, a qué velocidad, y sobre qué van a hablar. De nada sirve que una persona hable muy rápido si el receptor no puede captar nada del mensaje debido a que no hay un acuerdo previo de comunicación. Esto también pasa en la redes de datos, para que los dispositivos se puedan entender entre ellos, deben acordar a qué velocidad deben transmitir los datos, que codificación van a utilizar, cual es el tamaño del encapsulamiento, etc. Todo este conjunto de reglas las definen los protocolos los cuales muchos de estos pertenecen a una capa del modelo OSI y TCP/IP. Se puede pensar en un modelo de red como se piensa en un plano arquitectónico para la construcción de una casa. Se puede construir una casa sin el plano, pero el plano puede asegurar que la casa tiene la base y la estructura correcta para que ésta no caiga [31].

En un principio las redes no tenían un modelo establecido, trabajaban sobre los protocolos que cada compañía creaba; más tarde surgió un estándar llamado OSI creado por International Organization for Standardization (IOS) cuyas siglas cambian en cada lengua, por lo que los fundadores decidieron poner ISO derivado del significado "isos" en griego, lo que significa "igual" en español. El propósito de este modelo es crear un estándar sobre los protocolos de redes de datos para que

se pudieran comunicar todos los dispositivos en el mundo. Más tarde surgió otro modelo denominado TCP/IP derivado de investigadores y universidades con el fin de desarrollar un modelo basado en el trabajo del departamento de defensa de los Estados Unidos. Con el paso del tiempo el modelo TCP/IP fue dominando y el modelo OSI fue quedando solo como referencia, sin embargo, es importante conocer ambos modelos ya que la mayoría de las redes construidas hoy en día se basan en estos y en sus suites de protocolos. [31]

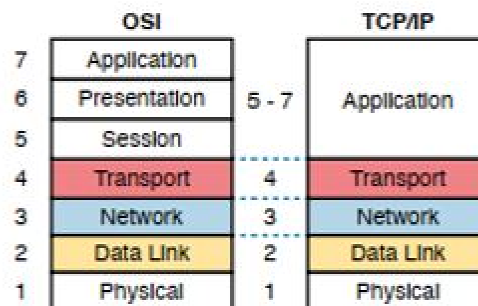


Figura 2.21: Modelo OSI comparado a el modelo TCP/IP [31]

En la figura 2.21 podemos observar que del lado izquierdo se tiene el modelo OSI con sus 7 capas que lo constituyen. De lado derecho se tiene el modelo TCP/IP el cual reduce el número de capas a 5, fusionando en su capa de aplicación, las capas de sesión, presentación y aplicación del modelo OSI. En cada una de las capas se hace uso de uno o más protocolos que interactúan con las capas cercanas y de esta forma, poder segmentar la construcción de la red.

Las redes de datos están clasificadas según algunas características que las componen, como lo son: el área geográfica que pueden abarcar, la cantidad de usuarios que pueden tener, entre otras cosas. Con base en esto, existen algunas clasificaciones de redes muy populares, algunas de estas pueden ser:

- LAN (Local Area Network)
- WAN (Wired Area Network)

- MAN (Metropolitan Area Netowrk)
- PAN (Personal area Network)
- VPN (Virtual Private Network)
- WLAN(Wireless Local Area Network)

Para el objetivo de la tesis, solo se hablará un poco sobre la Red de área local (LAN) ya que es en la cual está basada la arquitectura de red.

Una red de área local tiene como objetivo abarcar áreas pequeñas, como lo son escuelas, edificios, hogares, negocios, etc. Según referencias de Cisco, las redes LAN no tienen como tal límites de usuarios, pueden ir desde uno solo, hasta miles de ellos, el punto aquí es que está limitada a una sola área en especial. [32]

En la actualidad existen dos tipos de redes de área local; LAN con cable y LAN sin cable, denominada WLAN (Wireless Local Area Network). Por un lado las redes LAN pueden hacer uso de cable de cobre o bien fibra óptica como medio de transmisión, mientras que WLAN utiliza las ondas electromagnéticas para transmitir los datos. [31]

2.7 Internet de las cosas

Actualmente la tecnología digital se encuentra presente en un sinfín de lugares y dispositivos con el objetivo de ayudar y facilitar las tareas que se llevan a cabo diariamente ya sea en el hogar, en la oficina, viajando en automóvil o en el transporte público, en el gimnasio, en la escuela, etc. Las empresas han aprovechado toda esta tecnología, por ejemplo, para estar más cerca de los consumidores y de esta forma ofrecerles sus productos y servicios de formas más sencillas, cómodas y rápidas. [33]

Todas estas tecnologías, dispositivos y sensores digitales se basan en una serie de programas informáticos que funcionan según su lógica implementada y los datos proporcionados. Es importante mencionar que muchos de estos dispositivos y sensores, trabajan y procesan datos en tiempo real, ya que son utilizados en sistemas que requieren de respuestas y acciones inmediatas. Los datos son censados, procesados

y analizados con el fin de resolver ciertos problemas, desde el monitoreo del tráfico en una ciudad, medicina y cirugías a distancia, hasta la chapa de la puerta de una casa. La tecnología digital ha invadido y ha hecho posible que incluso las tareas más cotidianas puedan ser censadas para entregar información que puede ser utilizada en una infinidad de áreas como es seguridad, alimentación, salud, entre otras, volviendo las actividades que se llevan a cabo diariamente más sencillas.[33]

Esto es Internet de las Cosas, un mundo en donde millones de dispositivos y sensores se conectan a internet de forma que suben, bajan y procesan datos con el objetivo de llevar a cabo tareas específicas facilitándolas de muchas maneras. Las redes inalámbricas en conjunto con la inteligencia artificial trajeron consigo un gran avance en el cual los dispositivos no solo censan y procesan información, si no que son capaces de tomar decisiones basándose en los datos que recopilan.

> "Se estima que:
>
> - Más de 3 millones de nuevos dispositivos se conectan a internet todos los meses.
> - En los próximos 4 años habrá más de 30 millones de dispositivos conectados en todo el mundo:
>   - $\frac{1}{3}$: computadoras, smartphones, tables y televisores inteligentes.
>   - $\frac{2}{3}$: sensores, actuadores y dispositivos inteligentes que supervisen, controlen, analicen y optimicen el mundo."[33]

En general, la obtención de datos y de información a partir de dispositivos y sensores resulta ser una gran ventaja en todas las áreas que el internet de las cosas abarca. Las empresas, los gobiernos, hospitales, escuelas etc., están comenzando a usar todos estos datos para desarrollar una mejor capacidad a la hora de tomar decisiones, las cuales incluso pueden llegar a ser críticas. Sin embargo, todos los datos generados y recopilados, por sí solos, no pueden aportar valor ya que estos necesitan ser analizados para generar información. Las empresas, por ejemplo, pueden usar

todos estos datos para la toma de decisiones basándose en la información generada. De esta forma pueden conocer el impacto de sus productos y servicios y determinar mejoras que le den aun más valor a la empresa. En general este es el valor de los datos y aplica de la misma forma para muchas áreas, ya sea de ventas de productos o servicios, investigación, desarrollo, innovación, medicina, educación, etc.[33]

2.7.1 Procesamiento distribuido en internet de las cosas

Cuando los datos eran generados prácticamente por las personas, la cantidad generada era fácil de administrar, almacenar, procesar y depurar ya que realmente no eran cantidades enormes de datos. Sin embargo, con todo el avance antes mencionado, el incremento exponencial de dispositivos conectados a la red y el aumento de la automatización en las empresas y organizaciones ha ocasionado que todo este análisis se vuelva cada vez más difícil de procesar ya que se trata de cantidades enormes de información. "De hecho el 90 % de los datos que existen actualmente, se generó en los últimos dos años. Este aumento del volumen dentro de un período breve es una propiedad del crecimiento exponencial." [33]

Almacenar todos estos datos de forma centralizada en enormes arreglos de discos **(escalamiento vertical)** para ser procesados por una computadora muy potente ha dejado de ser una opción, ya que, por la cantidad de información, el análisis no se puede llevar a cabo en unidades de tiempo razonables, por muy potente que resulte ser una computadora.

Por lo tanto, el procesamiento distribuido dentro de internet de las cosas se ha vuelto una alternativa viable. El almacenamiento se lleva a cabo en diversos centros de datos, distribuyendo de forma física todo el conjunto de información. El procesamiento se lleva a cabo de la misma forma, ya que todo el conjunto de datos se divide entre un grupo de computadoras que los procesan y depuran **(escalamiento horizontal)**.

Todo esto resulta ser completamente transparente ya que a pesar de que los datos están distribuidos de forma física y su análisis se lleva a cabo de la misma manera, los programas clientes pueden acceder a los mismos como si se encontraran almacenados

de forma local, por lo que pueden acceder al conjunto completo. De igual forma, el procesamiento se lleva a cabo de forma distribuida, sin embargo, el cliente lo ve como si se llevara a cabo en su equipo, de esta forma cumpliendo con el objetivo principal del cómputo distribuido.

2.8 Clúster tipo Beowulf

Un clúster de tipo Beowulf tiene como propósito principal, el poder formar un cómputo paralelo con dispositivos de bajo costo o bien que sean dispositivos comerciales y de fácil adquisición. Este tipo de clúster ofrece una gran relación entre precio/beneficio, permitiendo el uso de software de bajo costo así como de software libre para el manejo de los nodos, su administración y la construcción completa del clúster mismo. [34]

> "Proporcionar a los usuarios sistemas consistentes, básicamente, en componentes que se tienen a disposición o que pueden conseguirse fácilmente en el mercado cibernético... abriendo la posibilidad de satisfacer las necesidades de cómputo paralelo" [35]

2.8.1 Historia

En 1994 Thomas Sterling y Don Becker del Center of Excellence in Space Data and Information Sciences (CESDIS) llevaron a cabo la construcción de un clúster en el cual conectaron 16 procesadores DX4 a través de canales Ethernet de 10 Mbps. La construción del clúster, al cual llamaron Beowulf, se realizó bajo el patrocinio del Proyecto Earth and Space Sciences (ESS). Uno de los principales problemas a los que Thomas y Don se enfrentaron, fue a las diferentes características de rendimiento y velocidad a las que trabajan los procesadores y tarjetas de red de su época.

El principal objetivo de la construcción del primer clúster Beowulf era el de explorar la posibilidad de crear un sistema de cómputo paralelo de alto rendimiento basado en nodos con características comerciales comunes. Este tipo de clúster fue

un éxito lográndose popularizar en la NASA, así como en los grupos académico científico. [36]

> "Gracias a estas configuraciones, las universidades con recursos limitados y sin acceso a una supercomputadora, encontraron una excelente opción de acceso al cómputo de alto rendimiento, para trabajar problemas científicos complejos que utilizan programas o códigos paralelos." [35].

Los clúster tipo Beowulf presentan una enorme ventaja en cuanto a recursos y el rendimiento ofrecido, ya que hoy en día el acceso a dichos recursos, como son tarjetas de red, procesadores, memoria RAM, discos duros, etc., se ha vuelto muy sencillo y relativamente barato. De igual forma la compatibilidad entre todos esos recursos ha ido a la mejora gracias la estandarización de protocolos de comunicación y de construcción, lo cual ha aumentado la capacidad de ensamblaje e interconexión entre dichos dispositivos haciendo aún más sencilla la construcción y el acceso a este tipo de clústers.

2.8.2 Características

Un clúster de tipo Beowulf consta de una serie de computadoras personales, en su mayoría con procesadores de la familia X86 de Intel. Sin embargo, puede ser implementado con procesadores distintos. Cada computadora se considera un sistema independiente, es decir, cuenta con su propia memoria primaria, secundaria y con su propio procesador. Su conexión está basada en redes que utilizan el protocolo Ethernet(802.3 IEEE). Con respecto al sistema operativo, la mayoría utiliza GNU/-LINUX. La última característica que involucra un clúster Beowulf es el tipo de API que se utilizará para la comunicación entre nodos; para el caso de esta tesis solo se involucrará a MPI y OMP. [36]

El clúster de tipo Beowulf implementado para el desarrollo de esta tesis se puede clasificar de la siguiente forma [37]:

- Es un clúster de cómputo de alto rendimiento (High Performance Computing(HPC)).

- Clúster de PC(CoPC) o pilas de PCs(Pops).
- Clúster con base en distribuciones Linux.
- Clúster de nodos heterogéneos: en este caso los nodos cuentan con procesadores de diferentes versiones y corren bajo diferentes sistemas operativos.
- Clúster o grupo clúster en donde se tienen 4 nodos conectados a través de una red LAN (Local Area Network).

2.8.3 Ventajas y desventajas

Los clúster de tipo Beowulf cuentan con varias ventajas y desventajas. Sin embargo, cada una de estas se les puede o no atribuir según el tipo de aplicación a la que están destinados dichos clústers. [37]

Ventajas

- Es económico, los nodos están basados en componentes comerciales, de fácil adquisición.
- La mayoría de los nodos que lo componen, tienen sistemas operativos de código libre.
- Cada nodo es un sistema independiente y se pueden hacer pruebas individualmente.
- Es escalable, se puede agregar un nodo extra sin mayor complicación.
- El ancho de banda de las redes LAN se ha incrementado muchísimo, lo que está relacionado a la velocidad de comunicación con cada nodo.
- Se pueden utilizar diferentes topologías de red para implementar el clúster.

Desventajas

- La configuración del clúster puede llegar a ser compleja, si los componentes del mismo tienen arquitecturas diferentes o bien, sistemas operativos totalmente distintos.

- La implementación del balanceo de cargas o la repartición de datos en cada nodo es complicada, ya que no en todos los casos los nodos tiene los mismos recursos (memoria primaria, ciclos de procesamiento, núcleos del procesador, etc.).

- La comunicación entre los procesadores, al tener cada uno su propio reloj, ocasiona que la ejecución se ralentice.

2.9 Requerimientos de software

> "El Instituto de Ingenieros en Electricidad y Electrónica, IEEE por sus siglas en inglés...define a los requerimientos de software como... condición o capacidad requerida por el usuario para resolver un problema o alcanzar un objetivo" [38]

2.9.1 Requerimientos funcionales

Los requerimientos funcionales enfocados en software, son aquellas características que debe cumplir un sistema, comportarse de la forma deseada, es decir, describen las acciones que dicho sistema debe realizar con base en ciertos objetivos. En ocasiones también los requerimientos funcionales establecen los puntos que no deben realizarse en el sistema [38]

2.9.2 Requerimientos no funcionales

Los requerimientos no funcionales son aquellos que no establecen una relación con los servicios que el sistema debe cumplir, así mismo pueden establecer limitantes en la implementación del sistema. Dentro de estos requerimientos podemos encontrar la seguridad, la disponibilidad, el rendimiento, entre otros criterios. [39]

Algunos de estos criterios pueden estar basados en el ISO 25010, que habla sobre la calidad de un producto de software dando como ejemplos los siguientes.

- Adecuación funcional

- Eficiencia de desempeño
- Compatibilidad
- Usabilidad
- Fiabilidad
- Seguridad
- Mantenibilidad
- Portabilidad

cuyas particularidades se describen con mayor detalle en la figura 2.22

Figura 2.22: ISO 25010 [40]

CAPÍTULO
3

Álgebra matricial

3.1 Definición de matriz

"Arthur Cayley, Matemático británico. En 1838 ingresó en el Trinity College de Cambridge, donde estudió matemáticas y fue nombrado profesor de esta disciplina; permaneció en Cambridge durante el resto de sus días. Uno de los matemáticos más prolíficos de la historia, publicó a lo largo de su vida más de novecientos artículos científicos. Es considerado como uno de los padres del álgebra lineal, introdujo el concepto de matriz y estudió sus diversas propiedades. Con posteridad empleó estos estudios para estudiar la geometría analítica de dimensión n. Arthur Cayley (1821-1895)." [41]

3.1.1 Definición

Una matriz es un arreglo rectangular de la forma:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} \tag{3.1}$$

Los números $a_{11}, a_{12}, a_{13}, ..., a_{ij}$ se denominan: **elementos de la matriz**. Una notación simplificada para expresar una matriz sería de la forma: $A = a_{ij}$

El primer subíndice i indica el renglón de la matriz, mientras que el segundo j indica la columna de la matriz.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1j} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2j} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3j} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & a_{i3} & \cdots & a_{ij} \end{bmatrix} \tag{3.2}$$

3.1.2 Orden de una matriz

El orden de una matriz es el tamaño de la misma, el cual se define como el número de filas m por el número de columnas $n: orden = m \cdot n$

El número de elementos total de una matriz es igual al producto de m o el total de filas, por n o el total de columnas.

3.1.3 Tipos de matrices

- **Matriz cuadrada**: es aquella en donde el número de filas es igual al número de columnas, $m = n$. Por lo tanto, el orden siempre será igual a $n \cdot n$.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{3.3}$$

- **Matriz renglón**: son todas aquellas matrices en donde el orden es igual a $1 \cdot n$. En otras palabras, son todas aquellas matrices que tienen un solo renglón.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \end{bmatrix} \tag{3.4}$$

- **Matriz columna**: En este caso, son todas las matrices cuyo orden sea igual a $m \cdot 1$, o que tengan una sola columna.

$$\mathbf{A} = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \end{bmatrix} \tag{3.5}$$

- **Matriz cero o nula**: es una matriz en la que todos sus elementos son iguales a 0.

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{3.6}$$

$$\mathbf{A} = \begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix} \tag{3.7}$$

- **Matriz identidad o unidad**: todas aquellas matrices cuadradas de orden n en las que los elementos de su diagonal principal son igual a 1.

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.8}$$

- **Matriz triangular superior**: se tratan de matrices cuadradas de orden n en las que todos los elementos debajo de su diagonal principal son iguales a 0. En

otras palabras, es una matriz donde $a_{ij} = 0$ para todos aquellos elementos en donde $i > j$.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{bmatrix} \tag{3.9}$$

- **Matriz triangular inferior**: se tratan de matrices cuadradas de orden n en las que todos los elementos por encima de su diagonal principal son iguales a 0. En otras palabras, es una matriz donde $a_{ij} = 0$ para todos aquellos elementos en donde $i < j$.

$$\mathbf{A} = \begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{3.10}$$

- **Matriz Hermitiana**: se denotan como A^* y son todas aquellas matrices cuadradas formadas por números complejos. Además una matriz hermitiana es igual a su propia transpuesta conjugada, esto es: $A = \bar{A}^T = A^*$ [42]

$$\mathbf{A} = \begin{bmatrix} 5 & 2-6i \\ 2+6i & 4 \end{bmatrix} \tag{3.11}$$

$$\bar{A} = \begin{bmatrix} 5 & 2+6i \\ 2-6i & 4 \end{bmatrix} \tag{3.12}$$

$$\bar{A}^T = A^* = \begin{bmatrix} 5 & 2-6i \\ 2+6i & 4 \end{bmatrix} = \mathbf{A} \tag{3.13}$$

- **Matriz simétrica**: son todas aquellas matrices cuadradas de orden n tales que sus elementos $a_{ij} = a_{ji}$ para todo i y para toda j [42]

  Sea A una matriz simétrica, entonces: $A = A^T$

  Es importante mencionar que toda matriz simétrica de números reales es her-

mitiana. Por lo tanto, las matrices utilizadas para el desarrollo de este proyecto, al ser simétricas, son hermitianas sin importar que no se esté trabajando con números complejos. [42]

3.2 Operaciones matriciales

Una operación matemática es un proceso en donde un operador realiza una acción sobre dos elementos de un conjunto, tomándolos y con base en ciertas reglas, los relaciona de forma que dicho proceso produce un resultado. Al tratarse de operaciones matriciales, los elementos operados o relacionados por un operador resultan ser matrices, por lo cual una operación matricial es el proceso en donde un operador realiza una acción sobre las dos matrices a operar. [43]

3.2.1 Suma de matrices

La suma de una matriz consta en sumar cada elemento de una matriz con el correspondiente de la otra, es decir la operación se hace de elemento con elemento.

Si las matrices $A = (a_{ij})$ y $B = (bij)$ tienen la misma dimensión, la matriz suma es:

$$A + B = (a_{ij} + b_{ij}) \tag{3.14}$$

La matriz resultante se obtiene sumando los elementos de las dos matrices que ocupan la misma posición. [44]

Ejemplo:

Sea

$$\mathbf{A} = \begin{bmatrix} 5 & -2 & 3 \\ -1 & 2 & 1 \\ 1 & 3 & 4 \end{bmatrix} \tag{3.15}$$

y

$$\mathbf{B} = \begin{bmatrix} 4 & 3 & -2 \\ 2 & 1 & 1 \\ 3 & -2 & 3 \end{bmatrix} \tag{3.16}$$

El resultado de la suma sería

$$\mathbf{A}+\mathbf{B} = \begin{bmatrix} 9 & 1 & 1 \\ 1 & 3 & 2 \\ 4 & 1 & 7 \end{bmatrix} \tag{3.17}$$

3.2.2 Resta de matrices

Para poder sumar o restar matrices deben tener el mismo número de filas y columnas. La operación se realiza de elemento a elemento, es decir, se restan los términos que ocupan la misma posición en las matrices. [44]

Ejemplo:

Sea

$$\mathbf{A} = \begin{bmatrix} 5 & -2 & 3 \\ -1 & 2 & 1 \\ 1 & 3 & 4 \end{bmatrix} \tag{3.18}$$

y

$$\mathbf{B} = \begin{bmatrix} 4 & 3 & -2 \\ 2 & 1 & 1 \\ 3 & -2 & 3 \end{bmatrix} \tag{3.19}$$

El resultado de la resta sería

$$\mathbf{A}\text{-}\mathbf{B} = \begin{bmatrix} 1 & -5 & 5 \\ -3 & 1 & 0 \\ -2 & 5 & 1 \end{bmatrix} \tag{3.20}$$

3.2.3 Multiplicación por un escalar

Se define la multiplicación de una matriz A por un escalar c como [45]

$$cA = (ca_{ij}) \tag{3.21}$$

Ejemplo

sea

$$c = 7 \tag{3.22}$$

$$\mathbf{A} = \begin{bmatrix} -3 & 2 & -2 \\ 1 & 3 & 2 \\ 2 & -2 & 1 \end{bmatrix} \tag{3.23}$$

entonces

$$\mathbf{cA} = \begin{bmatrix} -21 & 14 & -14 \\ 7 & 21 & 14 \\ 14 & -14 & 7 \end{bmatrix} \tag{3.24}$$

3.2.4 Multiplicación entre matrices

Sean dos matrices A y B en donde el número de columnas de la primera tiene que ser igual al número de filas de la segunda, es decir.

$$A \in R_{m\,x\,n}, B \in R_{n\,x\,l} \tag{3.25}$$

La matriz resultante será del tamaño de las filas de la primera matriz por las columnas de la segunda, es decir.

donde cada elemento de la matriz resultante viene dado por:

$$C_{ml} = \sum_{n=1}^{i} A_{mn} \cdot B_{nl} \tag{3.26}$$

Ejemplo:
Sea

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & -3 \\ -2 & 1 & 4 \\ 1 & 3 & -3 \end{bmatrix} \tag{3.27}$$

$$\mathbf{B} = \begin{bmatrix} 1 & 4 & 2 \\ 0 & 2 & -3 \\ -4 & 1 & 3 \end{bmatrix} \tag{3.28}$$

$$\mathbf{A} \cdot \mathbf{B} = \begin{bmatrix} 13 & 5 & -13 \\ -18 & -2 & 5 \\ 13 & 7 & -16 \end{bmatrix} \tag{3.29}$$

3.2.5 Potencia de una matriz

Consiste en multiplicar la matriz por ella misma n veces donde n es el exponente.
ejemplo:
sea

$$\mathbf{A} = \begin{bmatrix} 1 & -1 & 4 \\ -2 & 3 & 2 \\ 3 & 1 & -1 \end{bmatrix} \tag{3.30}$$

entonces para $n = 3$

$$A^3 = \begin{bmatrix} 1 & -1 & 4 \\ -2 & 3 & 2 \\ 3 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 4 \\ -2 & 3 & 2 \\ 3 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 4 \\ -2 & 3 & 2 \\ 3 & 1 & -1 \end{bmatrix} \tag{3.31}$$

$$A^3 = \begin{bmatrix} 15 & 0 & -2 \\ -2 & 13 & -4 \\ -2 & -1 & 15 \end{bmatrix} \begin{bmatrix} 1 & -1 & 4 \\ -2 & 3 & 2 \\ 3 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 9 & -17 & 62 \\ -40 & 37 & 22 \\ 45 & 14 & -25 \end{bmatrix} \tag{3.32}$$

3.2.6 Matriz de cofactores

Sea A una matriz cuadrada de orden n. La matriz de cofactores de A, denotada por *cof(A)*, se define como la matriz cuyos elementos son los cofactores correspondientes a los elementos de A. [46]

3.2.7 Matriz adjunta

Sea A una matriz cuadrada; la matriz adjunta de A, denotada por *adj(A)*, se define como la matriz transpuesta de la matriz de cofactores de A. [46]

3.2.8 Determinante

Para una matriz A de $n \cdot n$ (cuadrada), se considera el producto $a_1j_1........a_nj_n$, donde $\pi_j = (j_1, ..., j_n)$ es una de las permutaciones $n!$ de los enteros de *1* a n. Se define una permutación como par o impar según el número de veces que un elemento más pequeño sigue a uno más grande en la permutación. Por ejemplo, dada una tupla *(1, 2, 3)*, entonces *(1, 3, 2)* es una permutación impar, y *(3, 1, 2)* y *(1 , 2, 3)* son permutaciones pares.[45]

Es decir

$$\delta(\pi) = X_{ij} = \begin{cases} 1 \ si \ \pi \ es \ una \ permutación \ par \\ -1 \ en \ otro \ caso \end{cases} \tag{3.33}$$

Entonces el determinante de A, se denota por *det(A)*, y es definido por:

$$det(A) = \sum_{todas\ las\ permutaciones\ en\ \pi_j} \delta(\pi_j) a_{ij}......a_{nj} \tag{3.34}$$

Ejemplo

Sea

$$\mathbf{A} = \begin{bmatrix} 1 & -1 & 4 \\ -2 & 3 & 2 \\ 3 & 1 & -1 \end{bmatrix} \tag{3.35}$$

Entonces

$$\begin{aligned} det(A) = 1 \cdot 3 \cdot (-1) + (-1) \cdot 2 \cdot 3 + 4 \cdot (-2) \cdot 1 - 3 \cdot 3 \cdot 4 - 1 \cdot 2 \cdot 1 \\ -(-1) \cdot (-2) \cdot (-1) = -53 \end{aligned} \tag{3.36}$$

3.2.9 Inversa

Sea la matriz A y A^{-1} entonces $AA^{-1} = A^{-1}A = I$ donde I es la matriz identidad se dice que A^{-1} es la inversa de A

Ejemplo:

Sea:

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & 3 \\ 2 & -2 & 1 \\ -2 & 1 & -1 \end{bmatrix} \tag{3.37}$$

y

$$A^{-1} = \begin{bmatrix} \frac{-1}{8} & \frac{-1}{2} & \frac{-7}{8} \\ 0 & -1 & -1 \\ \frac{1}{4} & 0 & \frac{-1}{4} \end{bmatrix} \tag{3.38}$$

Existen diferentes algoritmos para encontrar la inversa de una matriz, algunos ejemplos son los siguientes:

- Algoritmo de Bareiss
- Eliminación de Gauss-Jordan
- Método de la matriz adjunta.

CAPÍTULO

4

Implementación

4.1 Diseño de la implementación del caso de estudio

En este capítulo se mostrará la forma en cómo fueron implementados los 4 sistemas necesarios para hacer las pruebas en capítulos posteriores, es decir, el sistema en secuencial, paralelo implementado con OpenMP, el sistema distribuido haciendo uso de MPI y el sistema distribuido-paralelo haciendo uso de MPI con OpenMP. Cada uno de estos con cada tipo de dato: entero, flotante, doble y corto (int, float, double y short, respectivamente por sus nombres en inglés).

4.1.1 Diagrama de casos de uso

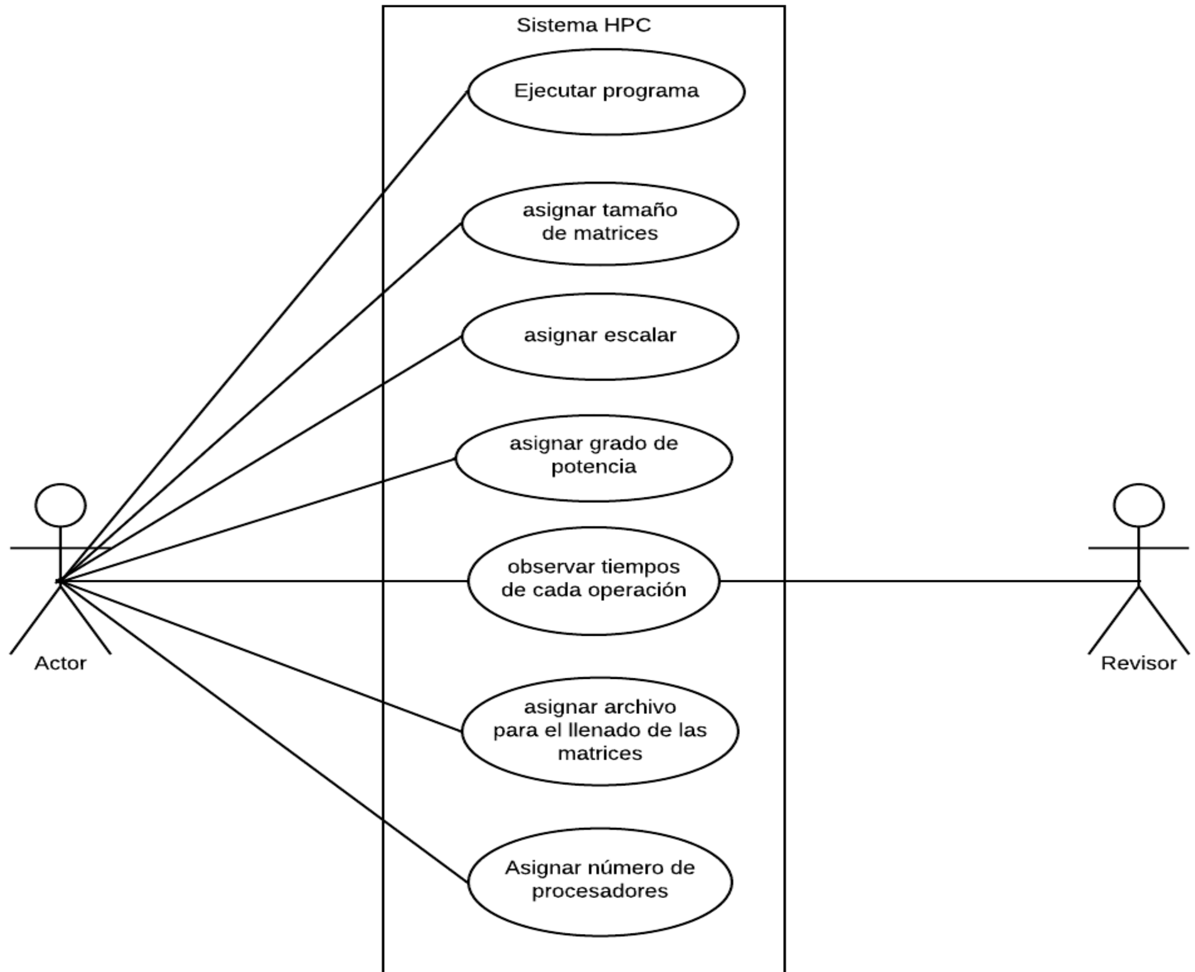


Figura 4.1: Diagrama de casos de uso del sistema HPC

En el diagrama 4.1 se pueden observar las actividades que cada usuario desempeña con el sistema implementado. El actor es el encargado de llevar a cabo la ejecución del programa así como la asignación de parámetros necesarios para la correcta ejecución. El Revisor solo verifica el tiempo de ejecución de cada operación.

4.1.2 Diagramas UML

En el diagrama 4.2 se puede observar el proceso de paso de mensajes para realizar las operaciones suma, resta, igualdad, multiplicación por un escalar e inversa. Se utilizaron las funciones *MPI_ Send y MPI_ Recv*.

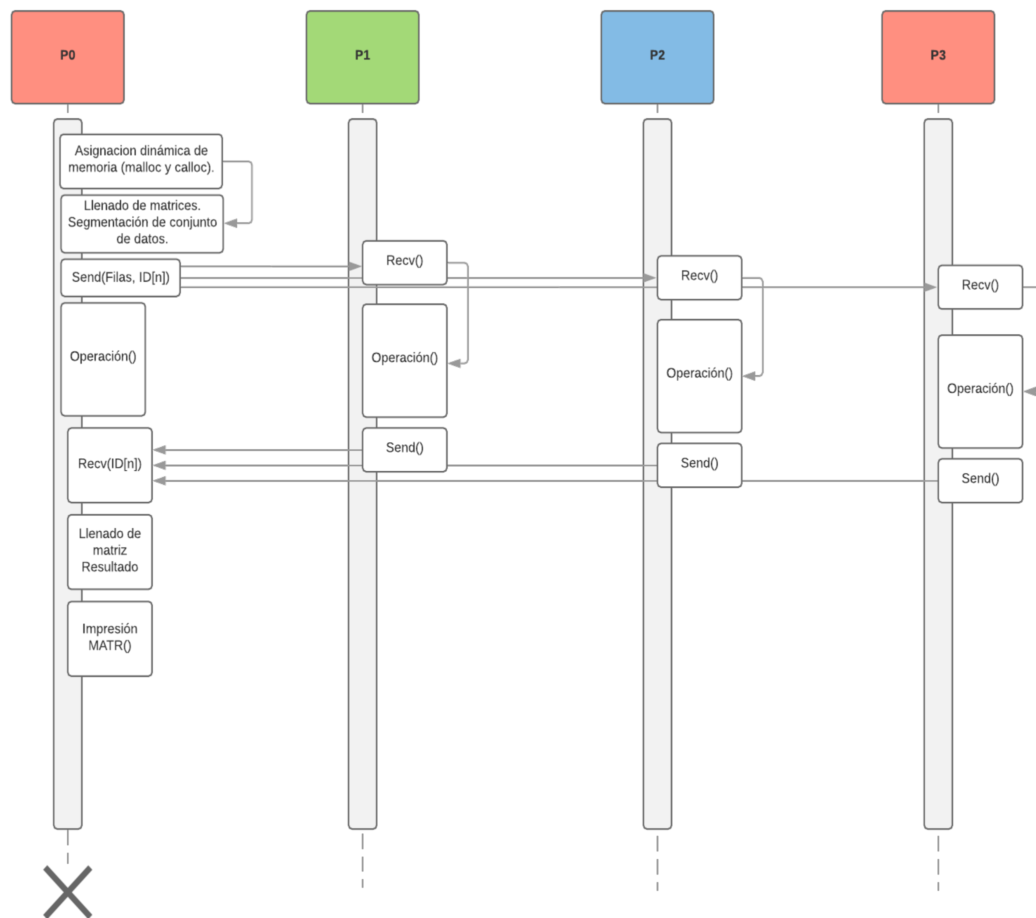


Figura 4.2: Paso de Mensajes: suma, resta, multiplicación por un escalar, igualdad e inversa.

Para el caso de la multiplicación y la potencia la forma en que se llevó a cabo

el paso de mensajes fue diferente al método utilizado para las operaciones anteriores ya que por el algoritmo a través del cual se resuelven, fue necesario enviar, completa y a cada procesador, una de las matrices. Se utilizó la función *MPI_ Bcast*, como se puede observar en el diagrama 4.3.

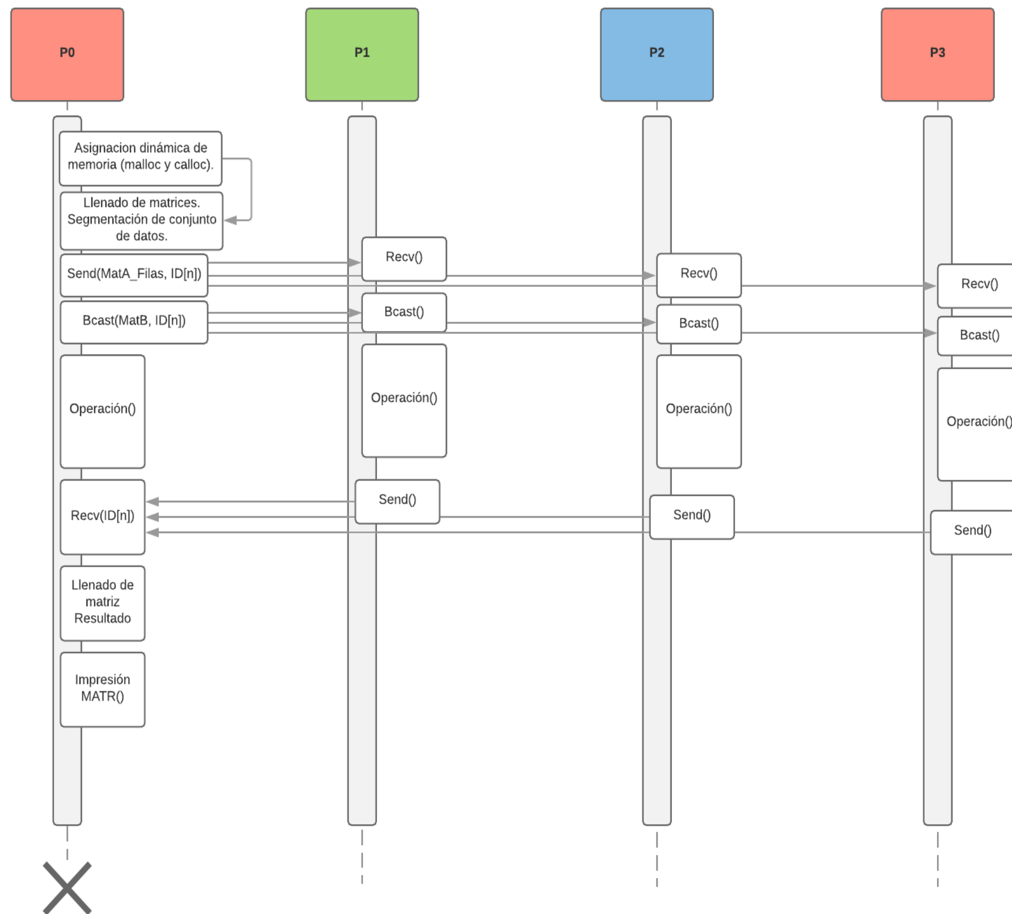


Figura 4.3: Paso de Mensajes: multiplicación y potencia.

Los métodos se describirán de forma detallada en la sección "Programación y/o Técnicas utilizadas".

4.1.3 Diagrama de Flujo

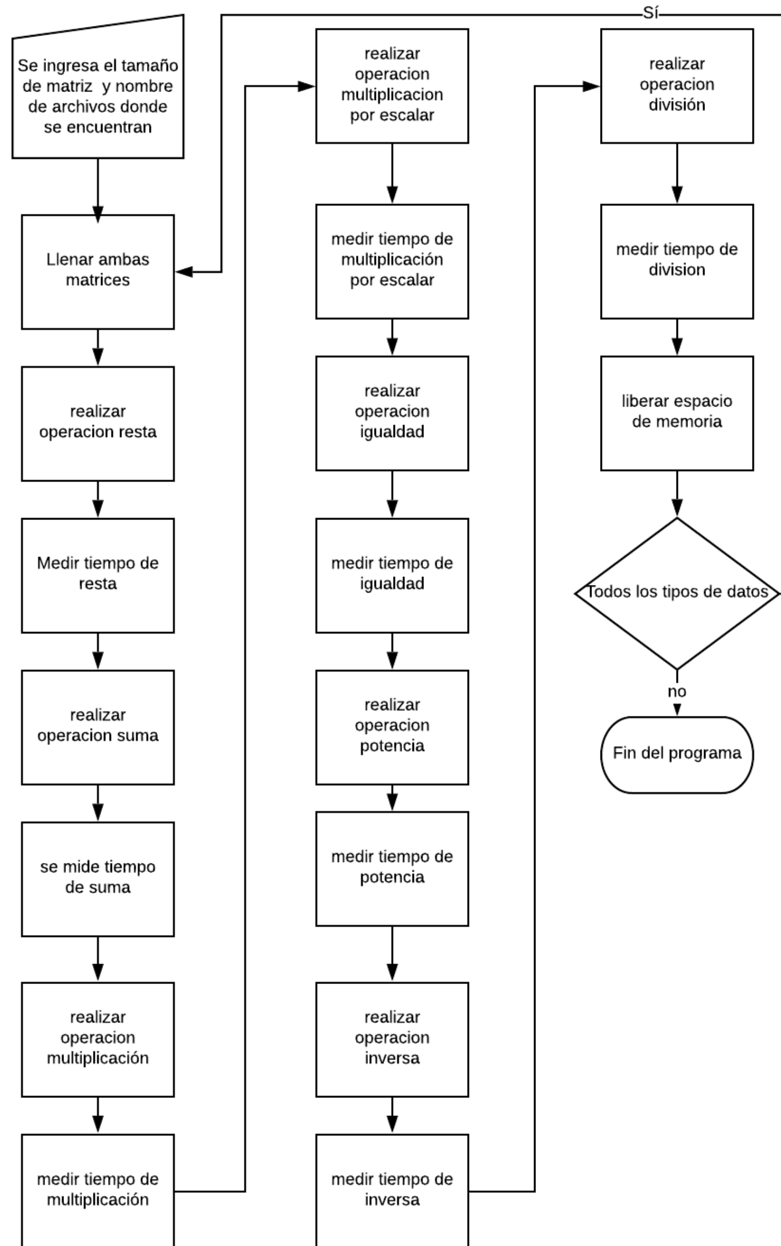


Figura 4.4: Diagrama de flujo del programa

El diagrama de flujo 4.4 muestra el proceso que lleva a cabo el programa de manera general. Se puede apreciar que el flujo comienza reservando espacio de memoria para las matrices con las que se va a trabajar, posterior a esto cada una de ellas es llenada con los archivos que se otorgaron como entrada al momento de ejecutar el programa. En seguida se comienzan a realizar las operaciones matriciales de resta, suma, multiplicación, multiplicación por escalar, igualdad, potencia, inversa y divisón. Al término de cada una, se hace una toma de tiempo con las cuales se sacarán las métricas de rendimiento. Al final de todo el proceso, se libera la memoria y se repite el ciclo para los cuatro tipos de datos con los que se está trabajando.

4.2 Levantamiento de ambiente

4.2.1 Instalación GNU/Linux

Para la instalación de GNU/Linux nativo, se descargó el SO [47]
El proceso de instalación es el siguiente:

- En caso de tener Windows instalado, generar y configurar las particiones necesarias para Linux.

- Generar la USB booteable: en este caso la ISO se booteo en una USB con ayuda del programa Rufus.

- Posteriormente, se verificó la configuración del BIOS y UEFI del equipo para poder iniciar el sistema desde la USB con el sistema operativo ya booteado.

- No se requieren configuraciones especiales para la instalación de GNU/Linux y específicamente de MINT para el desarrollo de este proyecto, por lo que basta con seguir el proceso de instalación marcado por el sistema operativo.

- Las distribuciones de GNU/Linux utilizadas para este proyecto ya cuentan con el compilador *gcc* por defecto y de igual forma ya cuentan con las librerías y el ligador de OpenMP que es el encargado de agregar las extensiones de la API.

4.2.2 Instalación y configuración de SSH

El protocolo SSH es ampliamente usado como un software de terminal remoto seguro. SSH nos permite registrarnos a una computadora remota a través de una red insegura, ejecutar un comando y transferir archivos entre una computadora remota y la computadora local [48]. Este protocolo fue el remplazo del protocolo telnet el cual permite de igual forma realizar conexión remota a otro dispositivo, con la gran diferencia de que la información que viaja de punto a punto, en telnet no está cifrada.

Para hacer uso de SSH se realizaron los siguientes pasos:

- Se configuraron las direcciones IP de cada dispositivo. Para realizar esto se configuró el archivo *interfaces* cuya ruta es la siguiente

  ```
  /etc/network/
  ```

  el comando queda de la siguiente manera

  ```
  $ sudo nano /etc/network/interfaces
  ```

  en el archivo se introducen las siguientes líneas para crear un IP estática en el equipo

  ```
  auto eth0
  iface eth0 inet static
  address 192.168.1.1
  netmask 255.255.255.255.0
  network 192.168.1.0
  gateway 192.168.1.254
  ```

  Se modifica el archivo *hosts* con el fin de asociar una IP con un nombre con mayor facilidad para recordar. Ubicado en la siguiente ruta

  ```
  /etc/hosts
  ```

  Esto se hace en cada uno de los nodos participantes del clúster

  ```
  192.168.1.1 master
  ```

```
192.168.1.2 slave1
192.168.1.3 slave2
192.168.1.4 slave3
```

- Se instala el servidor de ssh en el nodo master:

```
sudo apt-get install openssh-server
```

- Se realiza SSH a cualquier nodo mediante el nodo master:

```
ssh master
```

- Se pedirá contraseña del nodo remoto. Se ingresa la contraseña.

  Es muy importante darse cuenta que en esta parte solo estamos indicando la IP sin el usuario, esto se debe a que en los demás nodos también se debe crear el mismo usuario. De esta forma el ssh agrega por defecto el usuario con el que se está intentando realizar el ssh.

- Se genera la llave RSA en el nodo maestro

```
ssh-keygen -t rsa
```

  Se pedirán algunos datos como la ruta donde guardar la llave, y alguna contraseña para mantenerla segura. Se puede solo dar *enter* en esos pasos y generar la llave.

- Se envía la llave pública a todos los nodos

```
cd ~/.ssh
scp id_rsa.pub slave1:~/.ssh
```

- En cada nodo esclavo se agrega la llave pública del nodo master en el archivo de llaves autorizadas

```
cat id_rsa.pub >> authorized_keys
```

- Se corrobora que el SSH en todos los nodos, por medio del nodo maestro, se pueda realizar sin pedir contraseña

```
ssh slave1
```

4.2.3 Instalación de Sistema de Archivos de Red

NFS es un protocolo de distribución de archivos sobre la red, es decir, con él podemos compartir archivos de una forma casi como si estuviéramos accediendo a ellos localmente, mientras la realidad es que están almacenados en un dispositivo remoto.

Con este protocolo vamos a compartir todos los archivos que necesiten conocer todos los nodos, en este caso el archivo compilado y el archivo que contiene las matrices.

Para hacer la configuración de este protocolo se realizaron los siguientes pasos:

- Intalar el servidor NFS en el nodo que será Master con el siguiente comando:

```
$ sudo apt-get install nfs-kernel-server samba
```

- Se crea el directorio que será compartido en el nodo Master y en cada uno de de los nodos esclavos

```
$ mkdir RecursosMP
```

- Se cambian los permisos de dicho directorio previamente creado, del tal forma que todos tengan permiso de escritura, lectura, y ejecución de archivos.

```
$ chmod -R 777 RecursosMP
```

- Se especifica quién y cómo se van a conectar los usuarios al servidor NFS; se ingresa a la siguiente ruta:

```
$ sudo nano /etc/exports
```

Se ingresa el siguiente comando en la última línea del archivo

```
1 $/home/clustermpi/RecursosMPI *(rw,async, no_subtree_check,
    no_root_squash)
2
```

- Se reinicia el servidor NFS

```
$ sudo /etc/init.d/nfs-kernel-server restart
```

- Se crea un montaje automático en cada uno de los nodos. Se ingresa en la siguiente ruta

```
$ sudo nano /etc/fstab
```

y se agrega la siguiente línea

```
master:/home/clustermpi/RecursosMPI /home/clustermpi/
    RecursosMPI nfs rsize=8192, wsize = 8192,rw ,user, owner
    , exec, auto 0 0
```

Se reinician los nodos esclavos y en cuanto enciendan, al estar prendido el nodo maestro, en todos los nodos esclavos se va a montar la carpeta compartida en automático.

4.2.4 Instalación MPI

Para la instalación de la interfaz de paso de mensajes MPI se llevaron a cabo una serie de configuraciones y modificaciones de variables de entorno para el correcto funcionamiento de los programas. Es importante mencionar que para el desarrollo de este proyecto se utilizó la implementación de la interfaz de paso de mensajes llamada MPICH versión 3.1.

Para llevar acabo la instalación en los sistemas operativos Ubuntu y Mint, se realizaron los siguientes pasos:

- 1: Abrir una terminal y autenticarse como **root o usuario administrador** y posicionarse en la ubicación en la que se descargará y configurará el software. Para esto usar los siguientes comandos:

```
sudo su: para autenticarse como administrador.
cd <Ubicacion> para posicionarse en la carpeta deseada
```

- 2: Descargar el software, para lo cual se proporcionará una ruta de descarga. Usar el siguiente comando:

```
$ wget http://www.mpich.org/static/downloads/3.1/mpich-3.1.
   tar.gz
```

- 3: Una vez descargado el software, descomprimir el paquete:

```
$ tar -zxvf mpich-3.1.tar.gz
```

- 4: Se debe verificar que el sistema cuenta con una serie de bibliotecas y configuraciones necesarias para el correcto funcionamiento de MPICH. Ejecutar el siguiente comando y en caso de ser necesario, confirmar todas las solicitudes de instalación.

```
$ sudo apt-get install mc build-essential fort77 gfortran
   libstdc++6 libltdl7-dev
```

- 5: Una vez instalados todos los prerrequisitos necesarios, posicionarse en la carpeta descomprimida usando el comando:

```
$ cd mpich-3.1
```

- 6: Ejecutar el siguiente archivo con las configuraciones indicadas:

```
$ ./configure -prefix=/home/clustermpi/MPICH CC=gcc CXX=g++
    F77=gfortran FC=gfortran
```

- 7: Cuando el sistema termine todas las configuraciones necesarias, continuar con los siguientes comandos:

  - Primero ejecutar:

    ```
    $ make all
    ```

    Tardará aproximadamente de 15 a 20 minutos dependiendo de los recursos de la computadora.

  - Una vez terminado el proceso, ejecutar:

    ```
    $ make install
    ```

- 8: Finalmente se debe configurar el archivo **bash**:

  - Abrir el archivo con el siguente comando:

```
$ nano ~/.bashrc
```

- Sin borrar ninguna línea y hasta el final del archivo, agregar las siguientes líneas:

```
PATH:ruta/carpeta/bin/configurada
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:ruta/carpeta/lib/
    configurada
```

- Guardar los cambios en el archivo y salir del editor.

- 9: Finalmente para verificar la correcta instalación y configuración de MPICH ejecutar el siguiente comando:

```
$ mpiexcec -version
```

Se debería visualizar la versión y nombre de MPICH.

4.3 Software

4.3.1 Métodos computacionales para la implementación de las operaciones

Suma

Para la implementación de la suma se hizo un recorrido de elemento por elemento de cada matriz y cada elemento fue sumado con su respectiva parte. Tal como la ecuación 3.14 lo indica.

```
for(int i=0 ; i<n ; i++){
  for( int j=0 ; j<n ; j++ ){
    ResS[i*n+j]= matrizA[i*n+j] + matrizB[i*n+j];
  }
```

```
}
```

Listing 4.1: Ejemplo de suma

Se puede apreciar en el código 4.1 que el recorrido de las matrices es elemento por elemento

Resta

En el caso de la operación resta, el recorrido se hizo de la misma forma, elemento por elemento y se restan.

```
for(int i=0 ; i<n ; i++){
  for(int j=0 ; j<n ; j++ ){
    ResR[i*n+j]= matrizA[i*n+j] - matrizB[i*n+j];
  }
}
```

Listing 4.2: Ejemplo de resta

Como se puede apreciar en el código 4.2, de igual forma se recorren las matrices elemento por elemento y el resultado de la resta es guardado en su correspondiente posición en la matriz resultante.

Multiplicación por un escalar

En el caso de la multiplicación por un escalar, se recorrió la matriz y cada elemento de ésta se multiplicó por el mismo escalar.

```
for (int i=0; i<n; i++){
  for(int j=0; j<n; j++ ){
    ResME[i*n+j]=matriz[i*n+j]*esc;
  }
}
```

Listing 4.3: Ejemplo de multiplicación por escalar

Como se puede apreciar en el código 4.3, la matriz es recorrida por un ciclo y a su vez cada elemento es multiplicado por una escalar, en este caso dentro de la variable *esc*

Multiplicación

Para la implementación de la operación multiplicación, se llevó a cabo un recorrido de la fila de la primera matriz mientras que al mismo tiempo se recorría la columna de la segunda matriz, con el fin de multiplicar cada elemento y sumarlos al final.

```
for(int i=0 ; i<n ; i++){
  for(int j=0 ; j<n ; j++ ){
    for (int k=0; k<n ; k++){
      ResM[i*n+j]+=matrizA[i*n+k]*matrizB[k*n+j];
    }
  }
}
```

Listing 4.4: Ejemplo de multiplicación

Como se puede apreciar en el código 4.4 es necesario realizar la operación dentro de 3 ciclos, esto con el fin de recorrer columnas de la matriz A y filas de la matriz B en la misma iteración.

Igualdad de matrices

Para realizar la igualdad de matrices, se tuvo que recorrer cada elemento de la matriz con el fin de comparar una diferencia entre los elementos, al encontrar una

diferencia, se notifica lo sucedido.

```
1 for(int i=0; i<n; i++){
2   for(int j=0; j<n; j++){
3     if(matrizA[i*n+j] != matrizB[i*n+j]){
4       printf("las matrices no son iguales\n");
5     }
6   }
7 }
8 printf("Se han revisado las matrices\n");
```

Listing 4.5: Ejemplo de igualdad entre matrices

Se puede observar en el código 4.5 que se hace una impresión en la pantalla y no se detiene el ciclo, el objetivo de esto es porque se quiere observar el rendimiento de la operación en el peor de los casos donde el elemento diferente esté en la última posición de las matrices.

Inversa

Para el caso de la operación inversa se realizó mediante el método de la matriz adjunta multiplicada por el inverso del determinante, en este caso al trabajar con matrices hermitianas la transpuesta de la matriz de cofactores es ella misma, por lo que no se tuvo que realizar la transpuesta de la matriz de cofactores para obtener la adjunta. Para la obtención de la matriz de cofactores primero se implementó la obtención del determinante de cualquier matriz.

```
1 for(int j=0;j<=N-1;j++){
2   for(int i=0; i<=(N-1); i++){
3     if(i>j){
4       //Divir los elementos de la matriz
5       if(matriz[j*N+j]){
6         division=matriz[i*N+j]/matriz[j*N+j];
7         for(int k=0;k<=N-1;k++){
8         //Obterner el nuevo valor para los elementos en la diagonal
    inferior
9           matriz[i*N+k]-=division*matriz[j*N+k];
```

```
        }
      }
    }
  }
}
```

Listing 4.6: Ejemplo de determinante

En el código 4.6 se implementó la descomposición de la matriz en LU. El determinante de una descomposición LU de una matriz es solo el producto de los elementos de la diagonal. [49] Se obtiene entonces solo la matriz triangular superior.

```
for (int i = 0; i < N; ++i){
  det*=matriz[i*N+i];
}
```

Listing 4.7: Ejemplo de multiplicación diagonal principal

En el código 4.7 se multiplica la diagonal principal y se obtiene el determinante.
Posterior a ello se implementa una función con el fin de obtener cada cofactor de la matriz

```
int n = orden - 1;
int  * submatriz = malloc(orden*orden*sizeof(int));
int i, j;
int coeficiente =-1;

int x = 0;
int y = 0;
for (i = 0; i < orden; i++) {
  for (j = 0; j < orden; j++) {
     if (i != fila && j != columna) {
        submatriz[x*n+y] = matriz[i*orden+j];
        y++;
        if (y >= n) {
           x++;
           y = 0;
        }
     }
```

```
18   }
19 }
20 if((fila + columna)%2 == 0) coeficiente = 1;
21 return coeficiente * determinante_i(submatriz, n);
```

Listing 4.8: Ejemplo de cofactor

En el código 4.8 se puede observar cómo se crea una submatriz donde se descarta la fila y la columna del elemento que queremos obtener el cofactor, esto se sabe con los parámetros que se le pasa a la función nombrados ***columna*** y ***fila***; posterior a ello se multiplica el determinante de la submatriz creada previamente con el coeficiente del elemento al que se le quiere obtener el cofactor. Donde ***determinante_ i*** es la función del código 4.6

Para la obtención de la matriz de cofactores entonces solo basta con recorrer cada elemento de la matriz y sacará su respectivo cofactor como se indica en el código 4.9

```
1 for (int i = 0; i < n; ++i){
2   for (int j = 0; j < n; j++) {
3      matrizR[i*n+j]= cofactor_i(matriz, n, i, j);
4   }
5 }
```

Listing 4.9: Ejemplo de matriz de cofactor

Una vez obtenida la matriz de cofactores se procede solo a multiplicar por el inverso del determinante de la matriz a cada elemento, lo que indica el uso de la función previamente creada para la multiplicación por un escalar.

```
matrizDeCofactores_i(BI, TAMMATRIZ);
det = determinante_i(BI,TAMMATRIZ);
llenarMatriz_i("cofactI.txt", BI, TAMMATRIZ);
if(det)
  det = 1/det;
mul_escalar_i(BI, det, TAMMATRIZ);
```

Listing 4.10: Ejemplo completo inversa

En el código 4.10 primero se obtiene la matriz de cofactores, luego el determinante, después se llena la matriz con el valor del resultado de la matriz de cofactores, se valida que el determinante sea distinto de cero, y se procede a realizar la multiplicación por un escalar, en este caso el inverso del determinante. De esta forma se obtiene la matriz inversa. Donde ***BI*** es un apuntador a la matriz que se quiere obtener la inversa, ***TAMMATRIZ*** es el tamaño de la matriz

División

Para la divisón se obtuvo la inversa de la matriz que estará como divisor y se multiplica el dividendo por el divisor.

4.3.2 Técnicas de programación utilizadas

Además de los métodos utilizados que ya se mencionaron en la sección anterior, para la programación de cada una de las técnicas fue necesario llevar a cabo una serie de ajustes ya que al pasar de secuencial a paralelo, de paralelo a distribuido y de distribuido a paralelo-distribuido, el programa o técnica utilizada requería de ciertos cambios en la forma en que se implementaron las operaciones tanto de llenado y creación de las matrices, como de las operaciones matriciales.

En general, para todos los programas implementados, se utilizaron las funciones *malloc* y *calloc* para llevar a cabo la asignación de memoria de forma dinámica y poder almacenar las matrices creadas de esta forma. Por lo tanto el uso de apuntadores fue crucial durante todo el desarrollo, ya que de esta forma el manejo de las matrices se facilita de forma considerable.

Con respecto al manejo de las matrices, en general tuvieron que tratarse como vectores o matrices desdobladas, ya que para el manejo a la hora de trabajar de forma distribuida o con MPI, si éstas se trabajaban como un vector de vectores, la segmentación y distribución de los datos se complicaba de forma considerable generando errores a la hora de ejecutar y enviar los datos del proceso maestro a los procesos hijos y viceversa.

```
1     enviarMatriz_d(matrizA, filas, size);
2       for(int i=0; i<*filas; i++){
3         for (int j = 0; j < N; j++) {
4           matrizR[i*N+j]=matrizA[i*N+j]*E;
5         }
6       }
```

Listing 4.11: Ejemplo de uso de matrices desdobladas

En este caso, el codigo 4.11 corresponde a la operación multiplicación por escalar, que lleva a cabo el procesador *maestro* en el programa que se ejecuta de forma distribuida. Como se puede observar para la lectura y escritura de las matrices se utiliza el siguiente algoritmo $matrizA[i * N + j]$ en donde para especificar el índice en el que se posicionará la matriz se usan las variables i y j al mismo tiempo.

Para el caso en donde se utiliza la técnica de programación paralela con OpenMP, es importante mencionar que las directivas utilizadas, y con las que se obtuvieron los mejores resultados, son las siguientes:

- omp parallel: para los casos en donde se necesitaba iniciar una zona paralela.

- omp parallel for: en este caso se especificaba la versión paralela y además la ejecución de un loop de forma paralela.

- collapse: esta se utilizó junto con las directivas *for* para indicar la unión de las iteraciones de dos o mas loops for consecutivos.

- share: para especificar variables compartidas.

- schedule: en este caso esta directiva permitió especificar el tipo de política para la segmentación y repartición de las iteraciones de un loop o de un grupo de loop entre los subprocesos involucrados en la ejecución

En general estas directivas fueron las únicas que se utilizaron y las secciones que se lograron paralelizar de forma exitosa son todas aquellas en las que se utilizan las sentencias de iteración *for*. Las directivas se implementaron llevando a cabo combinaciones y verificando cuál de estas era la que entregaba los mejores resultados.

```
1      void multiplica_d(double *matrizA, double *matrizB, int n, char
     * nombre){
2    double *ResM = crearMatriz_d(n, 0);
3    #pragma omp parallel shared(ResM)
4    {
5      #pragma omp for collapse(2) schedule(auto)
6      for(int i=0 ; i<n ; i++){
7        for(int j=0 ; j<n ; j++ ){
8          for (int k=0; k<n ; k++){
9            ResM[i*n+j]+=matrizA[i*n+k]*matrizB[k*n+j];
10         }
11       }
12     }
13   }
14   deleteMatriz_d(ResM);
15 }
```

Listing 4.12: Ejemplo de uso de directivas de OpenMP

El código 4.13 corresponde a la función de multiplicación implementada con la técnica de programación paralela, como se puede observar las directivas utilizadas en este caso son *omp parallel, shared(), omp for, collapse() schedule(auto)*. Para este caso, se indica que la región paralela comienza a partir de la directiva *pragma omp parallel* y que dentro de esta la variable *ResM* será compartida para todos los hilos generados. Adicional a esto, con la directiva *pragma omp for* se indica la ejecución paralela de un loop y con las cláusulas *collapse y schedule* se indica que las iteraciones de los dos *for* consecutivos se sumarán y repartirán entre los hilos

involucrados siguiendo la política *auto*.

Es importante mencionar que dentro de los cambios que se llevaron a cabo al pasar de una técnica a otra, la función de la igualdad se implementó de dos formas. En la implementación secuencial y paralela, para determinar si dos matrices son iguales, el recorrido de las mismas se lleva a cabo de forma completa. En cambio, para las implementaciones distribuida y paralela-distribuida, al trabajar con diversos procesadores y dividiendo o segmentado la totalidad de los datos, los recorridos para determinar si dos matrices son iguales o no, no se llevaban a cabo de forma completa ya que en cuanto uno de los procesadores encuentra una diferencia, la búsqueda termina. Esto se logró gracias a las técnicas utilizadas y la forma en que se procesa la información, lo cual aumenta en gran medida la eficiencia a la hora de resolver esta operación.

Con respecto a la implementación del programa distribuido con las funciones de paso de mensajes de MPI, podemos destacar dos diferentes algoritmos.

- Para el caso de las operaciones suma, resta, multiplicación por un escalar, igualdad e inversa, el paso de mensajes se llevó a cabo con las funciones *MPI_Send y MPI_Recv* de la siguiente forma: primero la matriz o el conjunto de datos a enviar es segmentado en partes iguales, de tal forma que a cada procesador esclavo y al maestro les corresponda el mismo número de datos. Después con ayuda de la función *MPI_Send*, se envía el conjunto de datos, como se puede observar en el código 4.13, fila por fila solo a los procesadores esclavos correspondientes quienes, con la función *MPI_Recv* los reciben, como se ilustra en el código 4.14. Es importante mencionar, que el procesador maestro mantiene y opera su propio conjunto de datos. Inmediatamente después de recibir los datos, cada procesador los opera de acuerdo a la operación a resolver y finalmente se envían los resultados al procesador maestro siguiendo la misma dinámica. El procesador *maestro* se encarga de recolectar los datos y armar la matriz resultante y para hacerlo, cada que recibe un conjunto de datos, los ordena en la matriz resultante final de acuerdo al *ID y al tag* del procesador esclavo que envió el conjunto, justo como se puede observar en el código 4.15.

- Para las operaciones multiplicación y potencia, el paso de mensajes se llevó a cabo de forma diferente. Para éstas, se usaron las funciones *MPI_ Bcast, MPI_ Send y MPI_ Recv*, ya que fue necesario enviar completa una de las matrices a cada procesador esclavo. Para esto, la matriz se envió fila por fila con la función *MPI_ Bcast* (como se observa en el código 4.16), y cada procesador esclavo recibe los datos almacenándolos en una matriz del mismo tamaño, lo cual se puede ver en el código 4.17. El envío de la segunda matriz se hizo de la misma forma que la usada en las operaciones anteriores. Finalmente, y después de llevar acabo la operación correspondiente, los resultados se enviaron al procesador maestro, quien se encarga de armar la matriz resultado siguiendo la misma dinámica de las operaciones anteriores.

```
int offset=0;

for (int i = 1; i < *size; ++i){
  offset+=*filas;
  for (int j = 0; j < *filas; ++j){
    int * temp = matriz + (N*(j+offset));
    MPI_Send(temp, N, MPI_INT, i, 0, MPI_COMM_WORLD);
  }
}
```

Listing 4.13: Ejemplo de envío de matriz

```
for(int i=0; i<*filas; i++){
  iniRecv = MPI_Wtime();
  MPI_Recv(columnas, N, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  for(int j=0; j<N; j++){
    //se realiza operacion
  }

  MPI_Send(columnasR, N, MPI_INT, 0, i, MPI_COMM_WORLD);
}
```

Listing 4.14: Ejemplo de procesamiento en un nodo esclavo

```
MPI_Status status;
for(int i=0; i<(*filas *( *size-1)); i++){
  MPI_Recv(columnasCI, N, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
   MPI_COMM_WORLD, &status);

  //Posicionamos el arreglo de acuerdo al proceso y el orden en el
   que va

  int pos = ( ( *filas * status.MPI_SOURCE ) + status.MPI_TAG ) * N;
  for (int j = 0; j < N; ++j){
    matrizR[pos+j]= columnasCI[j];
   }
}
```

Listing 4.15: Ejemplo de recibimiento de datos procesador maestro

```
for(int i=0; i<N; i++){
  int * temp =  matriz + (i*N);
  MPI_Bcast(temp, N, MPI_INT, 0, MPI_COMM_WORLD);
}
```

Listing 4.16: Ejemplo de envío de una matriz completa con MPI_Bcast

```
for(int i=0; i<*filas; i++){
  MPI_Recv(columnasA, N, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  for(int j=0; j<N; j++){
    bloque[i*N+j]=columnasA[j];
  }
}
for(int i=0; i<N; i++){
  MPI_Bcast(temp, N, MPI_INT, 0, MPI_COMM_WORLD);
  for(int j=0; j<N; j++){
    matrizBI[i*N+j] =temp[j];
  }
}
```

Listing 4.17: Ejemplo de recibimiento de datos en un esclavo para la operación multiplicación

4.4 Método de sincronización utilizado

Ya que dentro de los objetivos de esta tesis fue llevar a cabo la implementación de un clúster en donde se hará uso de la interfaz de paso de mensajes (MPI) y al utilizar las funciones MPI_Send y MPI_Recv, el método de sincronización utilizado es el que está de forma implícita en dichas funciones. La comunicación y sincronización entre el grupo de procesadores involucrados se lleva a cabo de forma síncrona. Por lo tanto, la sincronización se realiza tal y como se describió en la sección 2.4 en el método de sincronización: Operaciones de comunicación sincrónica.

4.5 Hardware

Computadora	Procesador	Memoria primaria	Memoria secundaria	Tarjeta de red
Computadora ASUS VivoBook S	Intel Corei5 8th 4 núcleos físicos, 4 lógicos hasta 3.4 GHz.	Memoria RAM DDR4 de 8GB.	HDD 1TB.	Gigaware 71520
Computadora de escritorio Dell Inpiron	Intel Core 2 Quad, hasta 2.4 GHz.	Memoria RAM 2GB.	HDD 500GB	
Computadora Samsung	Intel Core i3 hasta 1.5 GHz.	Memoria RAM 4 GB DDR3	HDD 500 GB.	100 Mbps.
Computadora Acer Notebook aspire one	Intel Atom con 4 núcleos lógicos. Trabaja hasta 1.6GHz	Memoria RAM DDRII-533 de 1024MB	HDD 500GB	Fast Ethernet 10/100 Mbps.

Tabla 4.1: Características de las computadoras

En la tabla 4.1 se observan las características de cada una de las computadoras utilizadas para armar el clúster tipo Beowulf usado para la ejecución de los programas distribuidos.

CAPÍTULO 5

Métricas de desempeño

5.1 Definición

Las métricas de desempeño se pueden definir como un conjunto de mediciones(no exactas) a partir de las cuales se puede determinar el rendimiento de un sistema basándose en diferentes enfoques. Con base en dichas métricas se pueden determinar puntos clave en los cuales un sistema trabaja de forma más eficiente de acuerdo a sus características y los recursos con los que cuente. [50]

A la hora de probar la corrección de un software [51] es necesario realizar pruebas de software que nos permitan observar si se están cumpliendo los requisitos mínimos de implementación. Dichas pruebas comúnmente se dividen en dos grupos:

- Pruebas funcionales: Se prueba la funcionalidad que se requiere implementar en el producto de software. Se valida y verifica que el software cumple con lo especificado y hace lo que debe hacer y cómo debe hacerlo.

- Pruebas no funcionales: Se prueban los aspectos relacionados con el comportamiento del producto de software, no con el uso final. Las pruebas más conocidas y que se utilizan comúnmente son [52], [53]:

    - Compatibilidad: Mide la capacidad de convivir y trabajar en conjunto, ya sea entre sus propios componentes o con las aplicaciones vecinas.

- Adaptabilidad: Mide la tolerancia a los cambios del entorno actual.

- Seguridad: Mide e identifica los riesgos de infiltración y vulnerabilidades presentes en el software.

- Protección crítica: Mide el grado de corrección de los resultados del software y el grado de peligro al que se enfrentan los usuarios si algo sale mal. Es decir, permite prever escenarios desastrosos como daños al medio ambiente, lesiones graves a personas e incluso, la muerte.

- Estabilidad: Mide la eficiencia y la capacidad del software para funcionar continuamente durante un periodo determinado de software. Se mide si en ese periodo de uso normal falla o se presentan errores.

- Rendimiento: Mide el desempeño que puede alcanzar un producto de software con distintas configuraciones o parámetros de uso, involucra tanto tiempos de respuesta como nivel de ocupación de los recursos de hardware.

5.1.1 Tiempo de ejecución

Esta métrica se basa en el tiempo que le toma al programa llevar a cabo la tarea o tareas para el que fue diseñado. Este tiempo se mide desde que inicia la ejecución del programa, hasta que finalizan todos los procesos.

Se representa mediante una gráfica en la que se mapea el número de procesadores contra el tiempo de ejecución, tal y como se muestra en la figura 5.1:

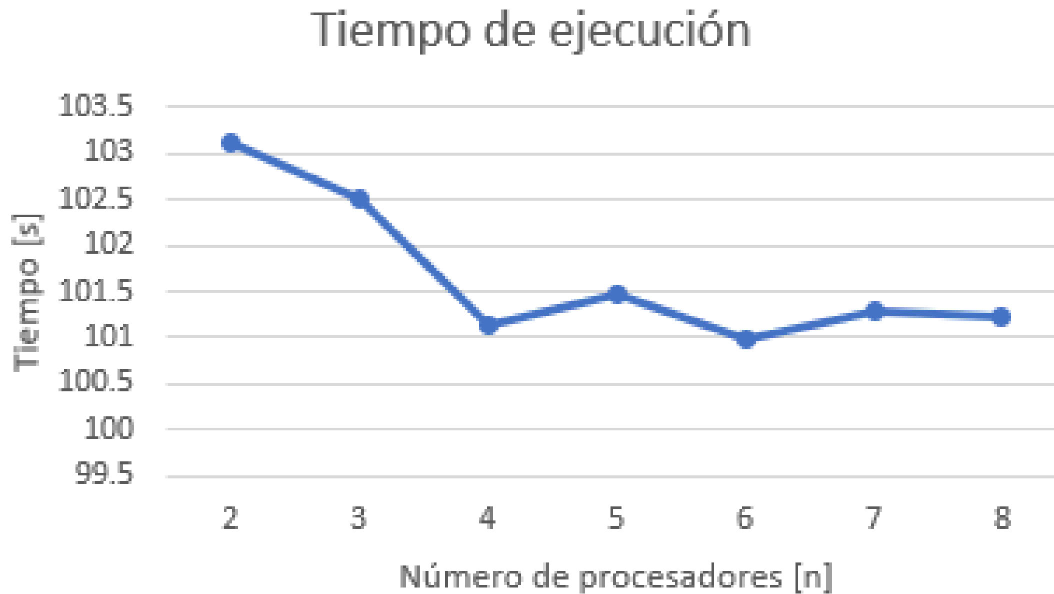


Figura 5.1: Ejemplo: Métrica tiempo de ejecución

5.1.2 Factor de costo

Representa el trabajo realizado por un programa. Aunque es bien sabido que en la mayoría de los casos un programa paralelo disminuye el tiempo de ejecución, al estar trabajando más de un procesador al mismo tiempo, se multiplica el tiempo del programa por el número de procesadores [26]. De acuerdo a lo anterior la fórmula para calcular el Factor de Costo es:

$$C(p) = número_procesadores(n) * tiempo_paralelo(t_p) \tag{5.1}$$

Para representar esta métrica se mapea el número de procesadores contra el factor de costo obtenido, como se muestra en la figura 5.2

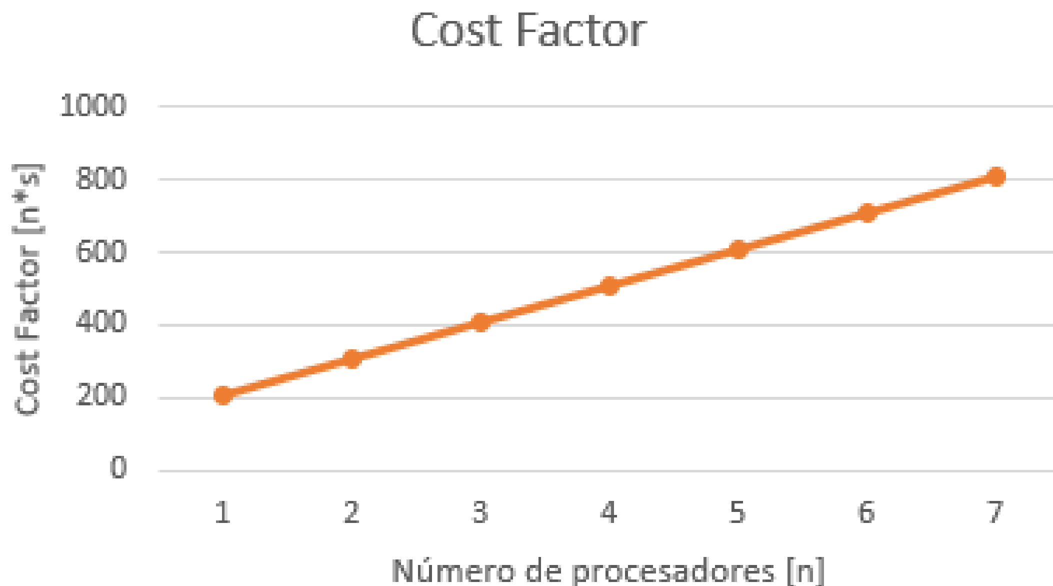


Figura 5.2: Ejemplo: Métrica factor de costo

5.1.3 Factor de aceleración

Es el valor de comparación del tiempo que toma un programa en secuencial con el tiempo que toma el mismo programa en paralelo; para que la comparación sea justa ambos programas se deben de ejecutar bajo las mismas condiciones, es decir, con los mismo recursos disponibles y los mismos datos a procesar [26]. La fórmula para calcular el factor de aceleración es la siguiente:

$$S(p) = \frac{tiempo\ del\ programa\ en\ secuencial}{tiempo\ del\ programa\ en\ paralelo} \tag{5.2}$$

Esta métrica se grafica comparando el número de procesadores contra el factor de aceleración obtenido, como se muestra en la siguiente imagen 5.3

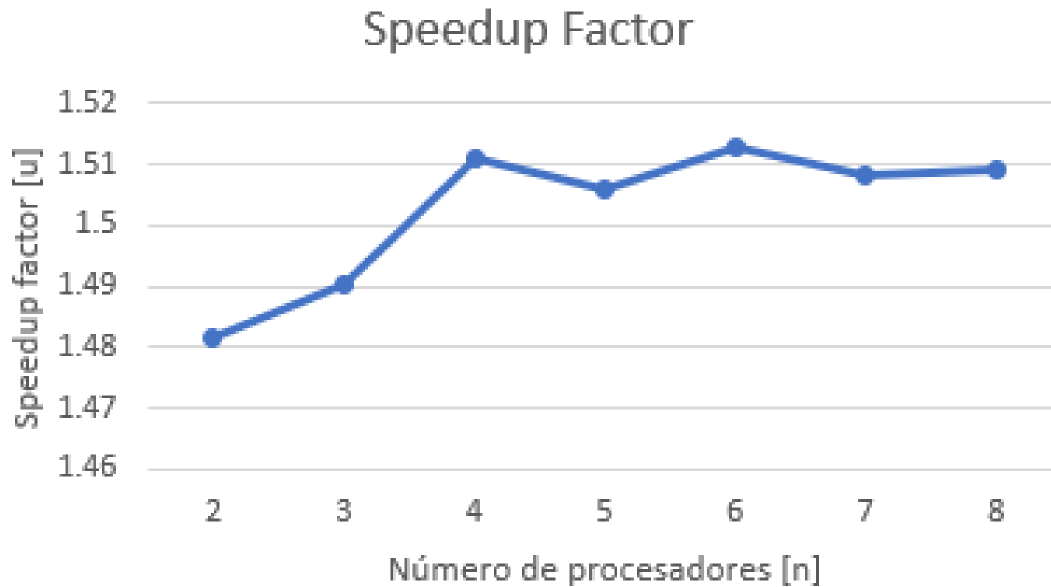


Figura 5.3: Ejemplo: Métrica factor de aceleración

5.1.4 Eficiencia

La eficiencia representa el tiempo que le toma a cada procesador llevar a cabo su tarea. En este caso, se verifica cuánta capacidad del procesador se está utilizando para resolver cierta tarea, por ejemplo si se obtiene un 25 % de eficiencia significa que cada procesador está utilizando solo un cuarto de su capacidad para resolver un problema de lo que le tomaría de forma secuencial. Para calcular la eficiencia se divide el factor de aceleración, entre el numero de procesadores y se multiplica por 100 % [26], [27] En la figura 5.6 se muestra una curva con la tendencia de la eficiencia

$$E = \frac{S(P)}{p} * 100 \tag{5.3}$$

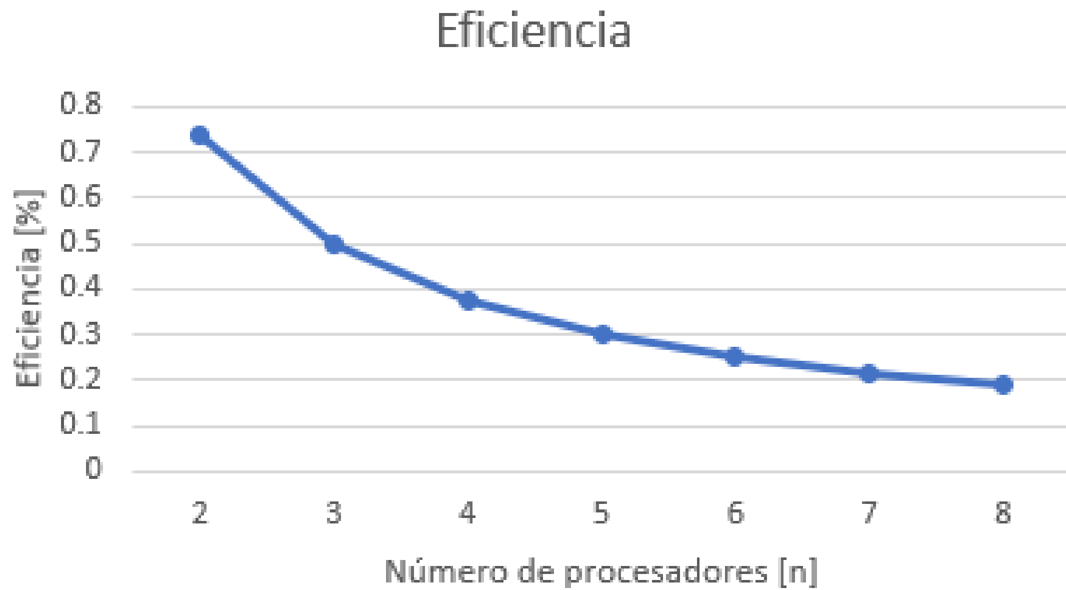


Figura 5.4: Ejemplo: Métrica eficiencia

5.2 Métricas de desempeño propuestas

Por la forma en que se tomaron los datos de los resultados, existen casos en los que para llevar a cabo un análisis más profundo, se volvió necesario diseñar y proponer ciertas métricas especiales para este proyecto enfocadas al tamaño de las matrices utilizadas y por ende a la cantidad de memoria utilizada durante el tiempo de ejecución de los programas.

5.2.1 Tiempo de uso de la memoria

Con esta métrica verificamos el tiempo en que la memoria está ocupada o se está utilizando para llevar a cabo la solución del problema o de las tareas implementadas en el programa.

$$TM = \frac{tiempo\ de\ ejecución}{número\ de\ procesadores} \tag{5.4}$$

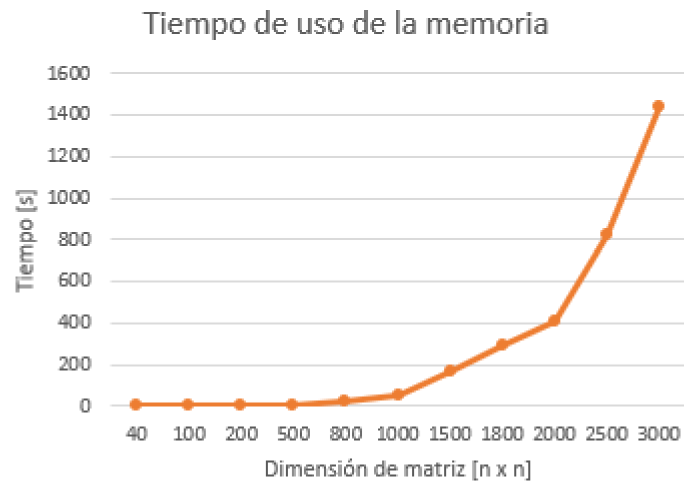


Figura 5.5: Ejemplo: Métrica tiempo uso de memoria

5.2.2 Tamaño de memoria utilizado

En este caso se mide la cantidad de memoria utilizada por cada tamaño de matriz, basándonos en los tipos de datos utilizados.

$$TMU = \sum_{i=tipo\ dato} \left(\frac{tamanio\ de\ matriz \cdot espacio\ tipo\ de\ dato}{1024}\right) \cdot \frac{1}{número\ de\ procesadores} \tag{5.5}$$

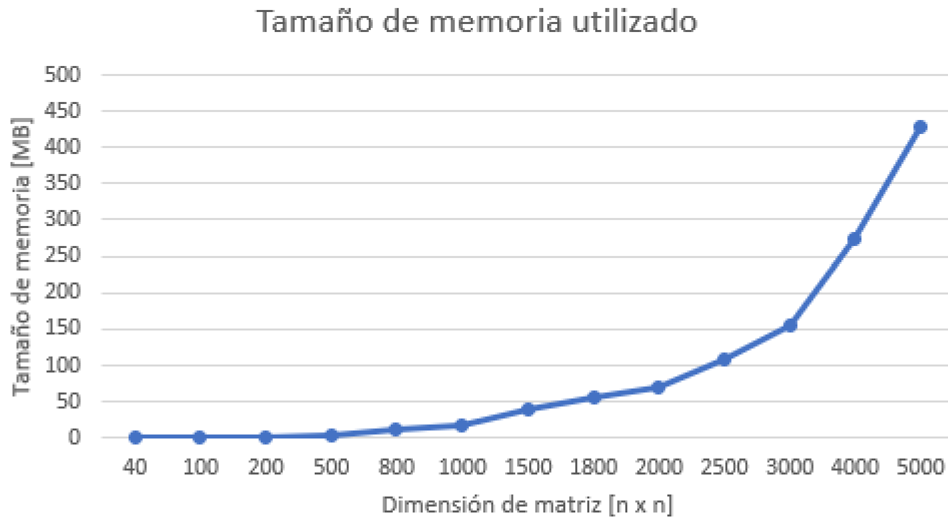


Figura 5.6: Ejemplo: Métrica tamaño de memoria utilizada

5.2.3 Planteamiento de las pruebas: Enfoque

El desarrollo de las pruebas se llevó a cabo con un enfoque en la eficiencia en cuanto al tiempo de procesamiento durante el momento en que los programas se ejecutan. Tomando este enfoque como base, las pruebas se dividieron y se llevaron a cabo obedeciendo a tres principales variables: el tiempo de ejecución del programa, el tamaño de las matrices o la cantidad de datos a procesar y el tipo de datos con los que se llevarán a cabo las operaciones.

De esta forma la obtención de resultados sera más favorable ya que se tendrá una visión más amplia para determinar si es factible o no el uso de HPC y en qué casos es mejor utilizar ciertas técnicas.

Es importante considerar que las pruebas se llevaron a cabo con los cuatro códigos implementados, los cuales se programaron utilizando diferentes técnicas de HPC. Los programas generados se implementaron con las siguientes técnicas:

- Programa secuencial: en este caso se llevó a cabo una programación estructurada en C y la cual es completamente secuencial, por lo que a la hora de ejecutarlo solo participa un procesador.

- Programa paralelo: Para esta implementación se hizo uso de la técnica de programación paralela con memoria compartida. En este caso se utilizaron directivas y declaraciones de la API OpenMP. Por lo tanto, para estas ejecuciones participan múltiples subprocesos utilizando y compartiendo la misma memoria.

- Programación distribuida: Se hizo uso del paradigma de paso de mensajes utilizando la API de MPI, y en este caso cada procesador es un nodo el cual cuenta con su propia memoria.

- Programación paralela-distribuida: se diseñó un programa híbrido, es decir, un programa paralelo con uno distribuido de forma que no solo participan múltiples procesadores en la ejecución, si no que, además en cada procesador se despliegan múltiples subprocesos para fragmentar aún más el trabajo.

Tomando en cuenta todas estas consideraciones las pruebas que se llevaron a cabo se dividieron y clasificaron de la siguiente forma:

- Pruebas del programa secuencial (1 solo procesador):
    - 4 tipos de datos: int, float, double, short
    - Diferentes tamaños de matriz(orden de la matriz): 40, 100, 500, 800, 1000, 1500, 2000, 3000

- Pruebas de programa paralelo:
    - 4 tipos de datos (int, float, double, short), 4 subprocesos y variando el tamaño de la matriz (40, 100, 500, 800, 1000, 1500, 2000, 3000).
    - 4 tipos de datos (int, float, double, short), tamaño de la matriz 1000x1000 y variando el número de subprocesos (2,3,4,5,6,7,8)

- Pruebas de programa distribuido (4 procesadores físicos con su propia memoria):
    - 4 tipos de datos (int, float, double, short)

    - variando el tamaño de la matriz (40, 100, 500, 800, 1000, 1500, 2000, 3000, 4000)

- Pruebas de programa paralelo-distribuido: 4 tipos de datos (int, float, double, short)

    - 4 procesadores con 4 subprocesos y variando el tamaño de la matriz (40, 100, 500, 800, 1000, 1500, 2000, 3000, 4000)
    - tamaño de la matriz 1000x1000 y variando el número de subprocesos (2,3,4,5,6,7,8)

CAPÍTULO

6

Resultados

Una vez que se llevó a cabo la implementación del clúster se procedió a la toma de tiempos llevando a cabo diferentes pruebas. Cada una de estas pruebas, como ya se mencionó, se enfocó a una variable diferente ya que de esta forma se pudo abarcar un mayor campo para poder determinar en qué casos es conveniente utilizar programación distribuida o paralela. Al hacer uso de diferentes variables, se pueden obtener conclusiones más completas sobre el uso, así como sobre las ventajas y desventajas de la programación paralela y distribuida o en general del cómputo de alto rendimiento.

Las diferentes variables que se consideraron para llevar a cabo las métricas de rendimiento son la siguientes:

- Cantidad de procesadores.
- Tamaño de Matriz
- Tipo de dato

A continuación se explicarán cada una de las métricas generadas.

6.1 Secuencial y Paralelo

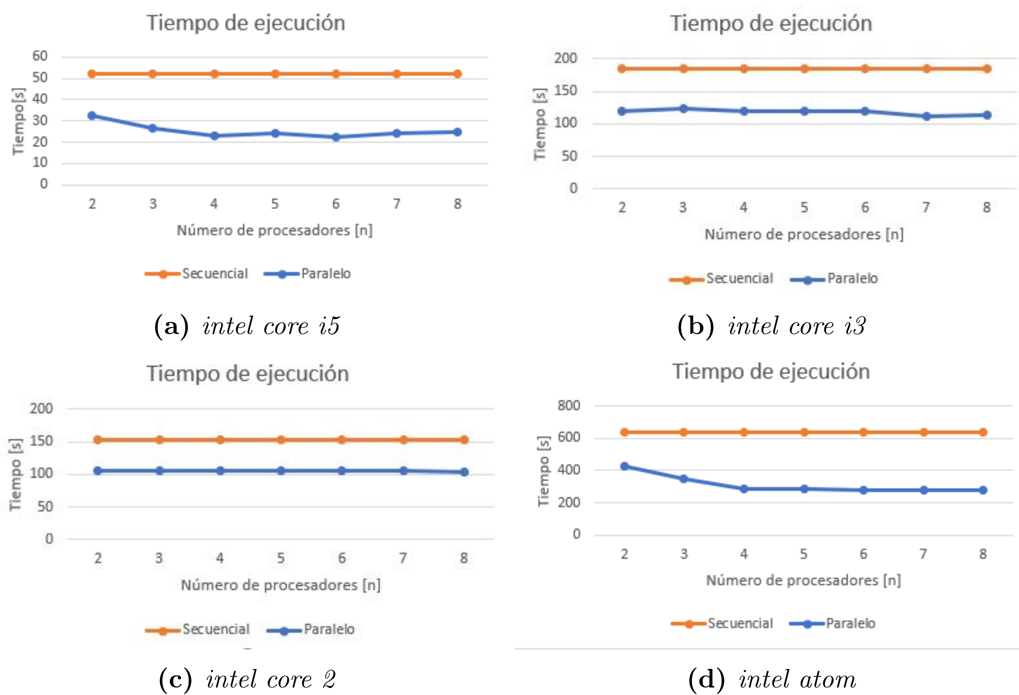
6.1.1 Variando el número de procesadores

En esta sección se observarán los resultados obtenidos al variar el número de procesos. Se llevará a cabo la comparación entre el programa secuencial y el paralelo

en donde se utiliza OpenMP. Por otro lado, se hará la comparación entre el programa distribuido haciendo uso de las rutinas de MPI y el programa paralelo-distribuido utilizando MPI con directivas de openMP.

Tiempo de ejecución

Como se puede observar en la figura 6.1 el tiempo obtenido para cada una de las computadoras y el punto en donde cada una trabaja de forma más eficiente, depende completamente de los recursos que poseen.



(a) *intel core i5* **(b)** *intel core i3*

(c) *intel core 2* **(d)** *intel atom*

Figura 6.1: Tiempo de ejecución secuencial vs paralelo

Al ejecutar el programa secuencial, solo un procesador interviene en la solución, por lo que el tiempo entre una ejecución y otra no suele variar considerablemente y es por eso que se puede apreciar una linea horizontal en los cuatro casos.

Sin embargo, y a la hora de trabajar con el programa en paralelo, el tiempo no solo disminuye considerablemente, si no que la tendencia de las gráficas se comporta de tal forma que al aumentar el número de subprocesos que intervienen en la ejecución,

el tiempo disminuye aun más hasta un punto en el que se mantiene casi constante y en el cual se vuelve a apreciar una tendencia horizontal.

La computadora con procesador *intel core i5*, por ejemplo, tiene un mejor desempeño al trabajar de forma paralela, ya sea con 4 o 6 procesadores, como se ve en la figura 6.1a. En el caso del procesador *intel core i3*, su desempeño mejora considerablemente al ejecutar el programa en paralelo con 7 procesos, como se aprecia en la gráfica 6.1b. Para las computadoras *intel core 2* e *intel atom*, al trabajar de forma paralela, se observa una tendencia más horizontal. En la gráfica 6.1c se puede ver claramente esta tendencia sin un aumento en la eficiencia, al igual que en la figura 6.1d, con la diferencia de que en este caso, si se aprecia una disminución en el tiempo de ejecución hasta llegar a los 4 procesadores, a partir de los cuales la curva vuelve a ser horizontal.

Factor de costo

El factor de costo en cada computadora que forma parte del cluster, tiene la misma tendencia como se vio en la figura 5.2. Sin embargo, como se puede apreciar en la figura 6.2a, la computadora con el procesador *intel core i5*, es la computadora con el menor factor de costo y esto se debe a que se tarda menos tiempo ejecutando un programa y a su vez los procesadores se utilizan menos tiempo. En contraste con la computadora con procesador *intel atom*, en la figura 6.2d se aprecia un factor de costo de al rededor de $2500[n \cdot s]$ es decir los procesadores tardan más en resolver una operación matricial y por lo mismo están más tiempo activos a lo que lleva a un mayor consumo energético. En conclusión, la computadora perteneciente a la gráfica 6.2d es la computadora con mayor factor de costo que se tiene dentro del cluster.

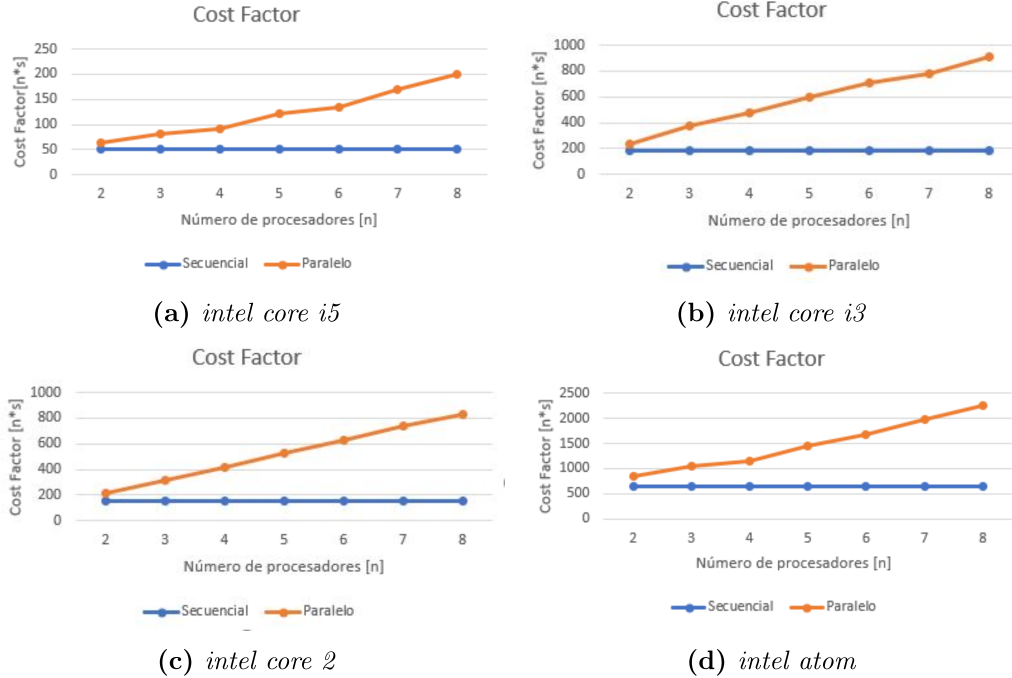
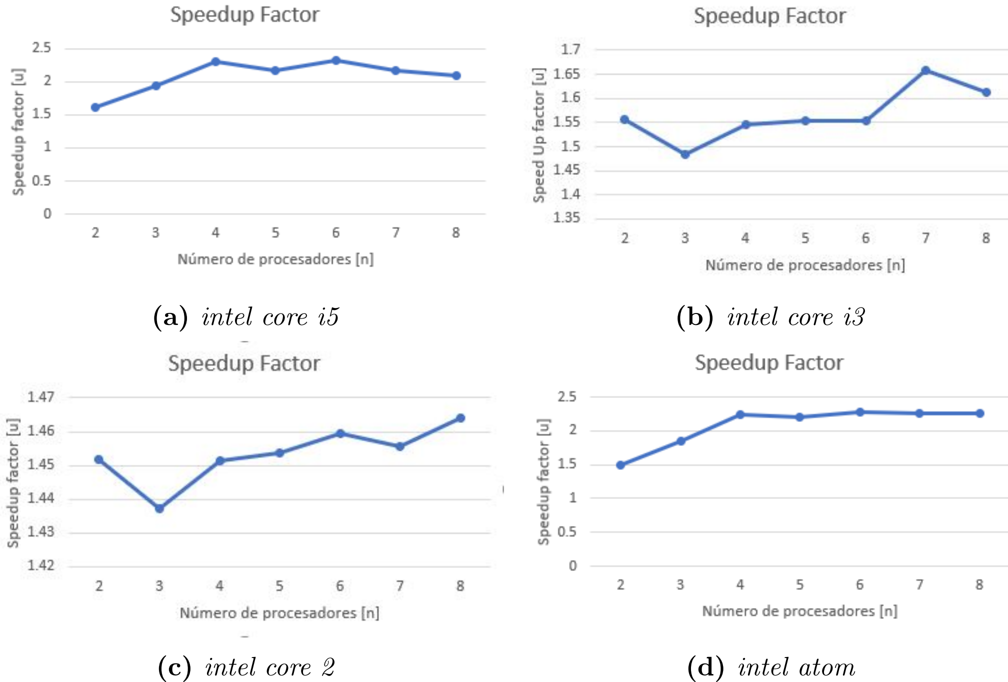


Figura 6.2: Factor de costo secuencial vs paralelo

Factor de aceleración

El factor de aceleración se refiere a qué tan rápida se volvió la ejecución de un programa que pasó de ser ejecutado de forma secuencial, a ejecutarse de forma paralela. Como se puede apreciar en la figura 6.3 éste depende del número de subprocesos que intervienen, y en general las gráficas siguen una tendencia parabólica en done se alcanza un punto máximo tal cual se pudo apreciar en la figura 5.3.

En las computadoras donde se puede ver la aceleración más notable es en la *intel core i5* y en la *intel atom* ya que como se puede ver en las gráficas 6.3a y 6.3d, respectivamente, la ejecución se aceleró casi 2.5 veces más al trabajar con 4 procesadores en cada una, siendo la primera la que logró un mejor desempeño.



(a) *intel core i5* **(b)** *intel core i3*

(c) *intel core 2* **(d)** *intel atom*

Figura 6.3: Factor de aceleración secuencial vs paralelo

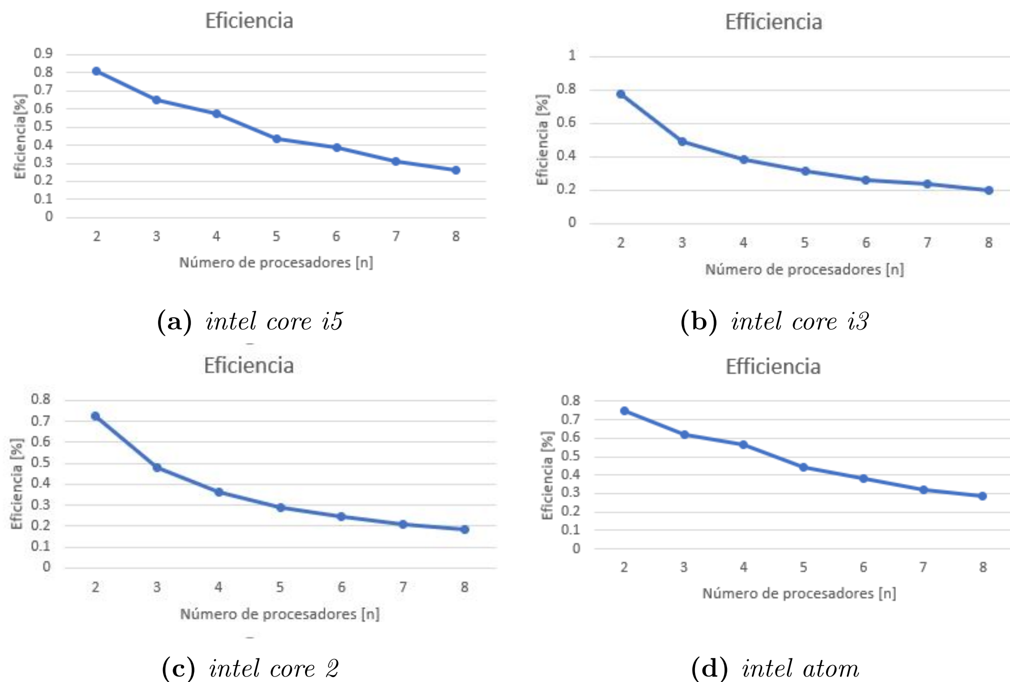
Para el caso de la computadora *intel core i3* el mayor factor de aceleración obtenido fue 1.66 veces más rápida que la ejecución secuencial, como se puede observar en la figura 6.3b. Por lo tanto, la computadora en donde se obtuvo la menor aceleración es en la *intel core 2* ya que como se puede apreciar en la gráfica 6.3c, el mayor factor de aceleración obtenido fue de 1.46.

Eficiencia

La eficiencia está sujeta al uso que se le está dando a cada procesador, por lo que las cuatro gráficas de las computadoras del cluster cumplen la tendencia que se vio en la figura 5.6. Aquí se puede apreciar que entre más procesos activos tengan las computadoras, éstas tienden a ser menos eficientes, esto se debe a que va reduciendo el porcentaje de uso de cada procesador para las operaciones asignadas. Sin embargo, la computadora con el procesador *intel atom* y la computadora con el procesador *intel core i5* son las más eficientes a la hora de paralelizar utilizando OpenMP, si se observa

cada gráfica con 3 procesadores activos, se puede ver que 6.4a y 6.4d están por arriba del 60 % de uso del procesador, mientras que 6.4b y 6.4c están al rededor del 50 % de uso de cada procesador y si se sigue aumentando el número de procesadores la tendencia continua. Por ejemplo, con 4 procesadores la computadora con procesador *intel core i5* y la computadora con procesador *intel atom* están cerca del 60 % de uso de sus procesadores, mientras que las dos computadoras restantes están por debajo del 40 % del uso de sus procesadores.

Se puede concluir que la computadora con procesador *intel core i5* y la computadora con procesador *intel atom*, son las más eficientes a la hora de paralelizar con OpenMP, esto se debe a que la primera tiene 4 núcleos en su socket, la segunda tiene 2 en el suyo pero con la capacidad de crear 2 núcleos virtuales y las menos eficientes tienen tan sólo 2 núcleos en los suyos.



(a) *intel core i5* **(b)** *intel core i3*

(c) *intel core 2* **(d)** *intel atom*

Figura 6.4: Eficiencia secuencial vs paralelo

6.1.2 Variando el tamaño de la matriz

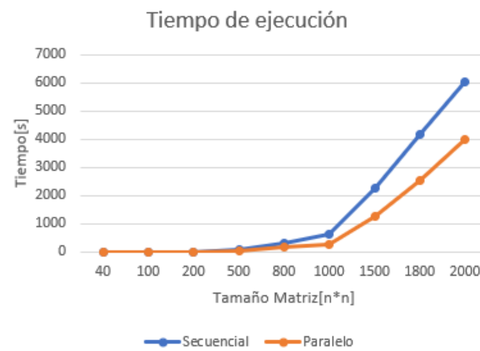
En esta sección se mostrarán los resultados obtenidos con las pruebas que se llevaron a cabo variando el tamaño de las matrices. En este caso se estableció un conjunto de tamaños específicos con los cuales se trabajó a lo largo de todas las pruebas, desde secuencial hasta distribuido-paralelo. Es importante mencionar que para este caso se ocuparon las métricas de tiempo de ejecución y tiempo de uso de la memoria.

Tiempo de ejecución

(a) *intel core i5*

(b) *intel core i3*

(c) *intel core 2*



(d) *intel atom*

Figura 6.5: Tiempo de ejecución Secuencial - Paralelo

Para este caso el tiempo de ejecución depende completamente del aumento en la cantidad de los datos con los que se tiene que operar. Es por esta razón que, a diferencia de la sección anterior, no se presentan gráficas con tendencia horizontal.Las gráficas de tiempo de ejecución para esta sección presentan más bien una tendencia exponencial ya que al crecer el tamaño de una matriz la cantidad de los datos crece de igual forma.

Como se puede notar en las gráficas de la figura 6.5, para el caso de las cuatro computadoras, el tiempo de ejecución no crece de forma considerable para tamaños de matrices de entre 40x40 a 1000x1000 y es precisamente a partir de este último tamaño en que los tiempos comienzan a crecer.

En este caso se cae de nuevo en el hecho de que todo depende completamente de los recursos. Para el caso de la gráfica 6.5a, que corresponde al procesador *intel core i5* se puede apreciar que el tiempo máximo en secuencial fue de aproximadamente 2500 segundos para matrices de 3000x3000. Éste comparado con el tiempo obtenido por el procesador *intel core i3*, es considerablemente más pequeño ya que como se observa en la gráfica 6.5b el tiempo máximo obtenido en secuencial para matrices de 3000x3000 fue de 7000 segundos.

Para el caso de la computadora *intel atom* se optó por detener las pruebas en matrices con tamaños de 2000x2000 elementos ya que el tiempo en esta computadora se incrementa demasiado rápido por los pocos recursos que posee. Como se observa en la gráfica 6.5d para ese tamaño de matriz el tiempo es casi tres veces mayor comparado con el tiempo de 2500 segundos logrado por la computadora con procesador *intel core i5* con matrices de 3000x3000.

En general la tendencia es exactamente la misma y como se puede observar en la gráfica 6.5c es recomendable trabajar de forma paralela con tamaños de matrices de entre 1000x1000 y 3000x3000 por el ahorro de tiempo que esta técnica presenta.

Tiempo de uso de la memoria

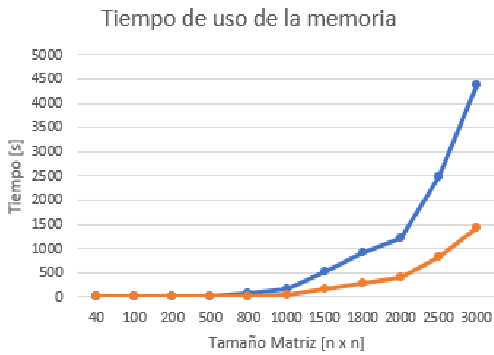
El tiempo de uso de la memoria está dado por el tiempo que se tomó para realizar el programa (en secuencial, paralelo y con cada tipo de dato) entre el número de procesadores que están involucrados , es decir:

$$TUM = \frac{tiempo\ programa}{número\ de\ procesadores} \tag{6.1}$$

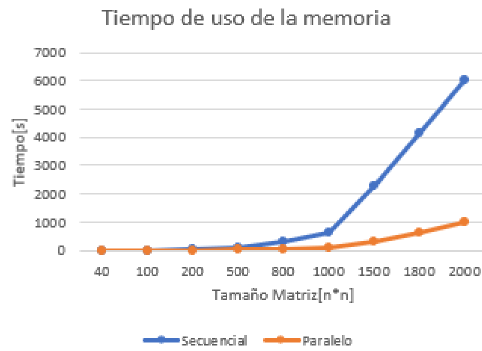
donde TUM es Tiempo de uso de la memoria.

(a) *intel core i5*

(b) *intel core i3*



(c) *intel core i3*



(d) *intel atom*

Figura 6.6: Factor de costo Secuencial - Paralelo

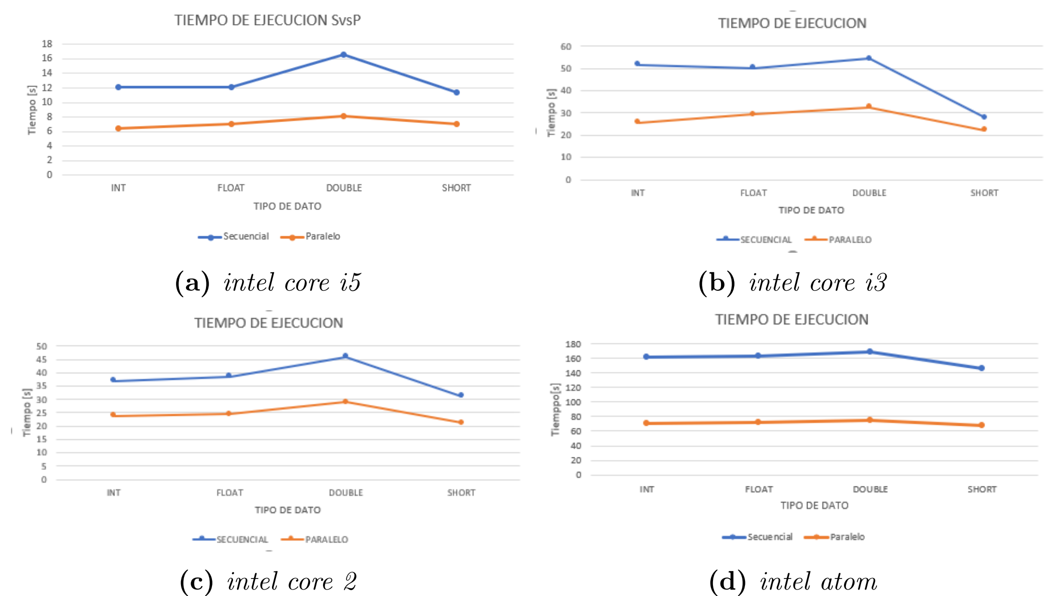
Como se puede observar en las figuras 6.6 el tiempo de uso de la memoria va aumentando conforme se incrementa el tamaño de la matriz, sin embargo, también va en crecimiento el ahorro de tiempo cuando se paraleliza el programa en secuencial con OpenMP, permitiendo un uso de la memoria principal menor que en el caso secuencial.

6.1.3 **Variando el tipo de dato**

Al variar el tipo de dato en un programa, el tiempo que le toma a una computadora resolverlo se ve afectado. En esta sección se mostrarán los resultados de las pruebas donde se lleva a cabo una variación de los tipos de datos utilizados.

Tiempo de ejecución

El tiempo de ejecución en este caso al depender completamente de los tipos de datos con los que se trabaja, presenta una tendencia en la cual se observan picos para ciertos tipos de datos, de esta forma se puede determinar cuál de estos tiene un mejor desempeño en el tiempo, tal cual se pude observar en la figura 6.7.



(a) *intel core i5* **(b)** *intel core i3*

(c) *intel core 2* **(d)** *intel atom*

Figura 6.7: Tiempo de ejecución Secuencial - Paralelo

Para la computadora con procesador *intel core i5* durante la ejecución secuencial fue el tipo de dato double con el que más tiempo le tomó resolver el problema y por ende a la hora de ejecutar en paralelo, fue el tipo de dato que obtuvo una mejora considerable de tiempo de ejecución, justo como se observa en la gráfica 6.7a. Esta misma tendencia se repite para las computadoras con procesador *intel core i3* e

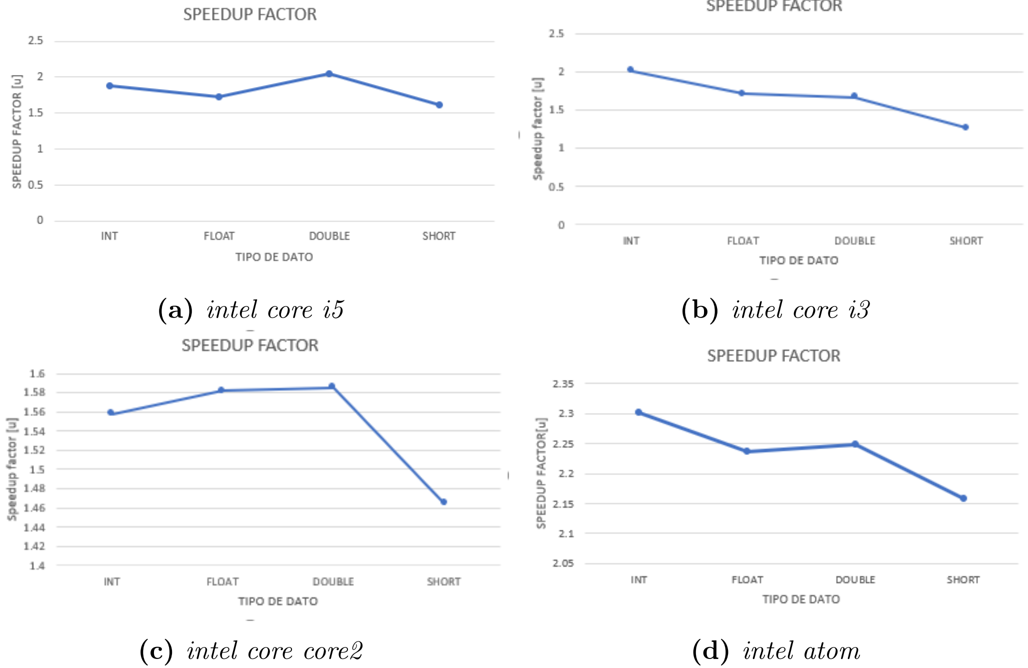
intel core 2, como se puede apreciar en las gráficas 6.7b y 6.7c respectivamente. Sin embargo, en este caso al ejecutar el programa en paralelo, la mejora de tiempo es menor.

Para el caso de la computadora con procesador *intel atom*, no se presenta un salto significativo en el tipo de dato double, sin embargo la tendencia se mantiene. A pesar de esto, y como se observa en la gráfica 6.7d, la mejora de tiempo al ejecutar en paralelo es bastante grande para los 4 tipos de datos.

En general con el tipo de dato short es con el que las computadoras tardan menos tiempo en llevar a cabo la solución del programa y en donde se presenta una menor diferencia al pasar de una ejecución en secuencial a una en paralelo. Con respecto a los tipos de dato int y float, el tiempo que se toma para trabajar con estos, es muy similar y las mejoras de tiempo al pasar a paralelo tampoco cambian considerablemente entre un tipo de dato y otro.

Factor de aceleración

Según las pruebas que se realizaron se puede observar que en las gráficas 6.8a y 6.8c se ve una aceleración en el programa paralelo de casi 2 veces en promedio por ambas gráficas para el tipo de dato double (que es la mayor aceleración dentro de los cuatro tipos de datos). Por otro lado en la gráfica 6.8d y 6.8b se ve una tendencia diferente, siendo el tipo de dato int con mayor aceleración. Lo único que se puede concluir en las gráficas de 6.8 es que el tipo de dato short tiene una diferencia muy pequeña de tiempo en su versión paralelo con OpenMP respecto a su versión secuencial.



(a) *intel core i5*

(b) *intel core i3*

(c) *intel core core2*

(d) *intel atom*

Figura 6.8: Factor de aceleración Secuencial - Paralelo

6.2 Distribuido y Paralelo-Distribuido

En esta sección se mostrarán los resultados obtenidos a partir de las pruebas llevadas a cabo con el clúster. Es importante resaltar que en este caso las computadoras se encuentran trabajando en conjunto y que los tiempos obtenidos son un promedio del tiempo que tardaba cada computadora en procesar su parte del trabajo.

6.2.1 Variando el número de procesadores

Tiempo de ejecución

En la figura 6.9 se puede apreciar un tiempo de ejecución constante para el sistema distribuido(curva de color azul), esto se debe a que los nodos siempre son fijos (en este caso cuatro nodos dentro del clúster). Sin embargo para el sistema Paralelo-Distribuido (curva de color naranja), se puede apreciar una caída de tiempo cuando se varían los procesadores de 2 a 3 y de 3 a 4. Esto se podría interpretar de forma que

entre más se vayan incrementando el número de procesadores, menor tiempo tardará el sistema en resolver las operaciones matriciales. Sin embargo, esto no es verdad, el sistema Paralelo-Distribuido es más eficiente con 4 procesadores activos en OpenMP. Adicional a esto se ve claramente que el sistema Paralelo-Distribuido es más rápido que el sistema distribuido por casi 10 segundos en promedio, con una matriz de 1000 x 1000.

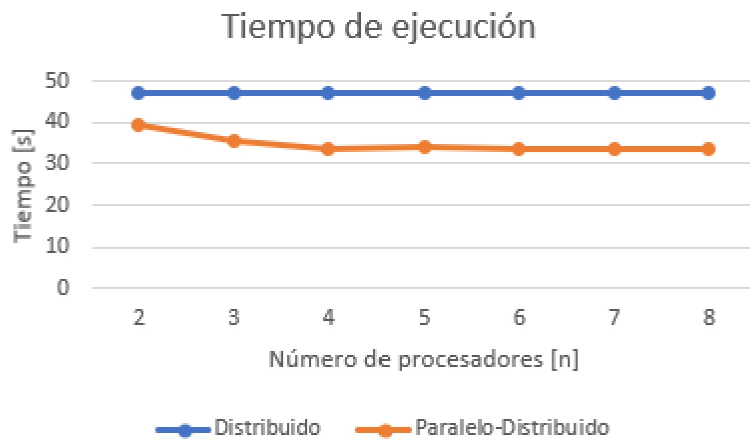


Figura 6.9: Tiempo de ejecución Distribuido vs Paralelo-Distribuido

Factor de costo

En la figura 6.10 se puede apreciar cómo el factor de costo del sistema distribuido-paralelo va aumentando según se vayan variando los subprocesos de éste. En la curva de color naranja (sistema distribuido) el factor de costo va en incremento ya que se multiplica el número de procesadores activos en OpenMP por el tiempo total del programa tal como se mencionó en la ecuación 5.1. Por otro lado en el sistema distribuido (curva color azul) se ve constante, y esto se debe a que los nodos en el cluster solo tienen asignado un proceso a la tarea, es decir, siempre es constante, por lo que el factor de costo no incrementa.

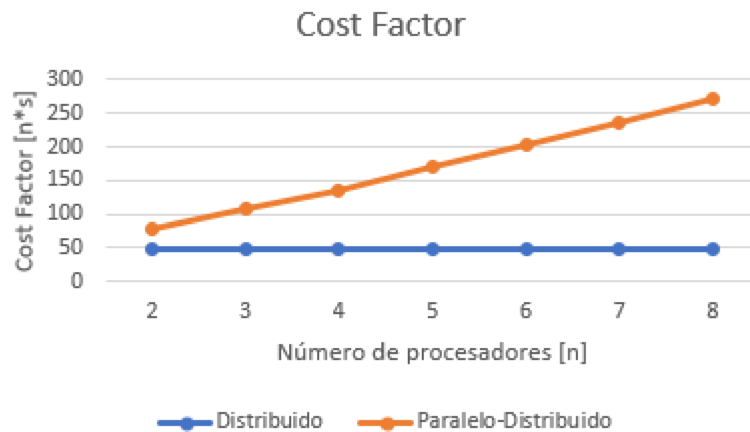


Figura 6.10: Factor de costo Distribuido vs Paralelo-Distribuido

Factor de aceleración

Como se puede ver en la gráfica 6.11 la tendencia del factor de aceleración, al pasar de un sistema distribuido a una sistema paralelo-distribuido, se comporta de la misma forma que en los casos de la sección anterior, por ejemplo, como se puede observar en la gráfica 6.3.

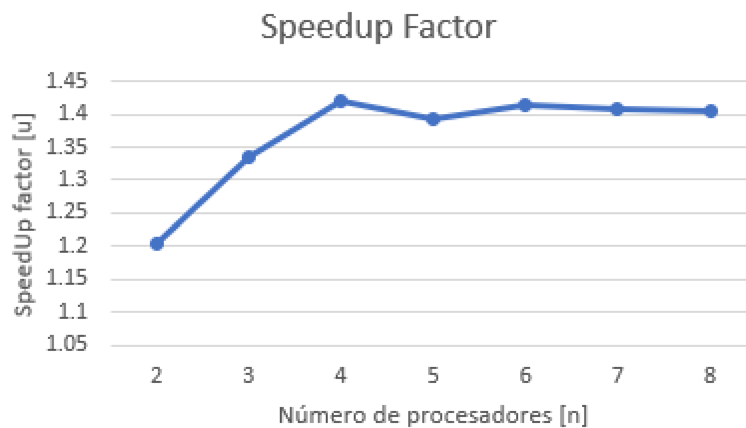


Figura 6.11: Factor de aceleración Distribuido vs Paralelo-Distribuido

Esto quiere decir que en efecto, trabajar de forma paralela con un programa que ya se está ejecutando de forma distribuida sigue mejorando la eficiencia del tiempo

de ejecución.

Es importante mencionar que al trabajar con cuatro procesadores es cuando se obtiene la máxima aceleración, ya que como se puede observar en la gráfica 6.11, la ejecución paralela-distribuida se aceleró casi 1.45 veces más que la ejecución distribuida.

Eficiencia

La tendencia obtenida es casi idéntica a la observada en la sección anterior. Como se puede ver en la figura 6.12, la eficiencia va disminuyendo conforme se aumenta el número de procesadores involucrados en la ejecución. Esto sucede porque al repartir el trabajo, los procesadores involucrados van desperdiciando una parte de su capacidad para procesar información, en otras palabras, entre más procesadores estén involucrados en la ejecución, cada procesador desperdicia parte del potencial total que utilizaría si ejecutara por sí solo el programa.

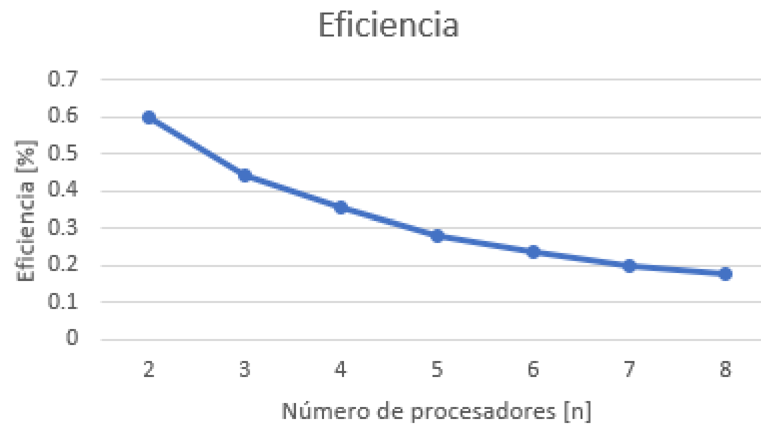


Figura 6.12: Eficiencia Distribuido vs Paralelo-Distribuido

6.2.2 Variando el tamaño de la matriz

Tiempo de ejecución

Variando el tamaño de la matriz como se vió en la figura 6.5, de igual forma en la figura 6.13 va en incremento según la dimensión de las matrices con las que se realizan

las operaciones. Lo que se puede resaltar en los resultados es que efectivamente, y como se esperaba, la implementación en el sistema Paralelo-Distribuido toma menor tiempo para resolver las operaciones matriciales con los cuatro diferentes tipos de datos que se utilizaron (int, float, double, short).

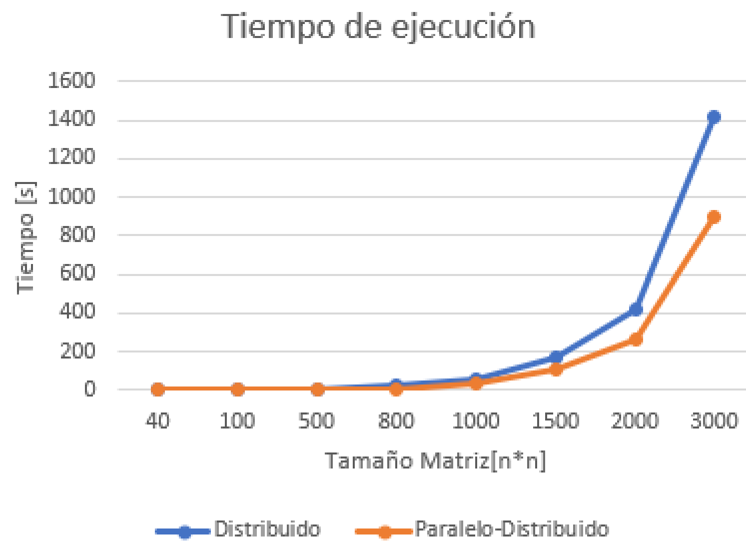


Figura 6.13: Tiempo de ejecución

6.2.3 Variando el tipo de dato

Tiempo de ejecución

Al trabajar de forma distribuida y de forma híbrida, esto es, paralela-distribuida, de igual forma se observa una mejora considerable con los cuatro tipos de datos con lo que se trabajó, sin embargo la tendencia en ambos casos se repite. Como se puede observar en la gráfica 6.14 el tiempo que se tomó para ejecutar el programa con los tipos de dato int y float es muy similar para ambas ejecuciones tomando, por ejemplo, 38 y 40 segundos respectivamente para la ejecución distribuida así como 20 y 21 segundos respectivamente, para la ejecución paralela-distribuida. En este caso el tiempo máximo obtenido en ambas ejecuciones se dio con el tipo de dato double, y el mínimo con short.

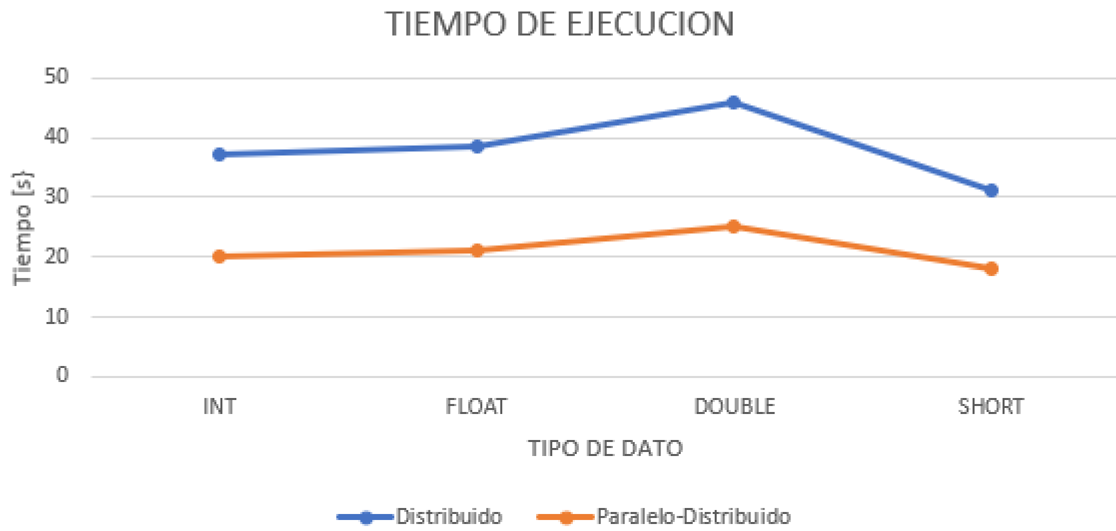


Figura 6.14: Tiempo Distribuido vs Paralelo-Distribuido

Factor de aceleración

En la figura 6.15 se puede notar que el tipo de dato entero es el que más se acelera a la hora de paralelizar el sistema distribuido, tal como los dos nodos menos eficientes que se tenían en el clúster y como lo muestran sus respectivas gráficas 6.8b y 6.8d, se puede decir con base en los resultados que el clúster se va a comportar por lo menos variando el tipo de datos igual al nodo menos eficiente dentro de él.

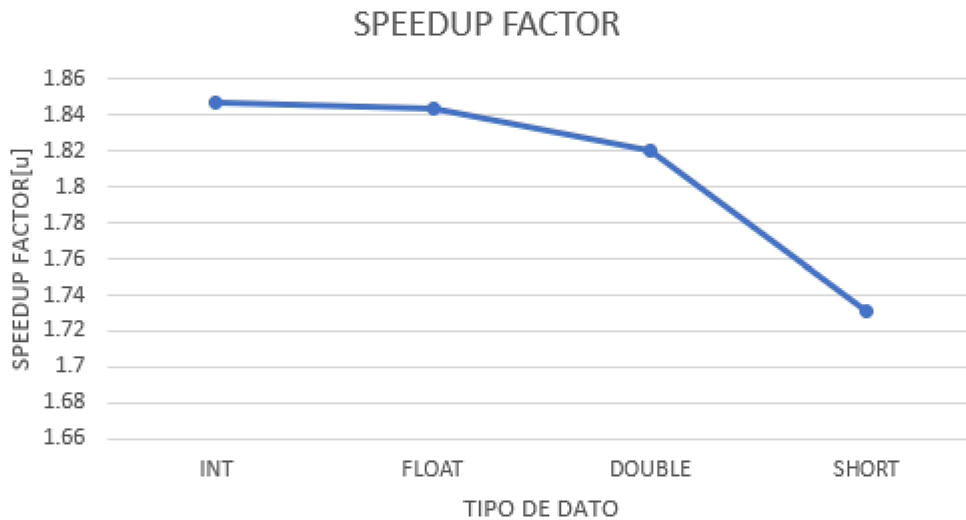


Figura 6.15: Factor de costo Distribuido vs Paralelo-Distribuido

6.3 Pruebas operación inversa y división

En esta sección se tomará el tiempo de ejecución que llevó realizar la operación inversa y división el cual se expondrá en la gráfica 6.16. Se hicieron las pruebas independientes debido a que el método que se utilizó en la inversa, involucra una matriz de cofactores que toma mucho tiempo resolver, computacionalmente hablando, lo cual no permite llevar una solución más allá de matrices con dimensiones mayores a 200 x 200. Estas dos operaciones en particular, para poder llegar a su resultado, involucran a la multiplicación y multiplicación por escalar. Por lo anterior, para la medición de tiempos se tuvo que medir el tiempo que toma escribir los datos en un archivo y leerlos para continuar con la solución. Leer los datos de un archivo y escribirlos no se puede paralelizar, por lo que lo realiza un solo nodo dentro del clúster, en este caso el proceso *maestro*.

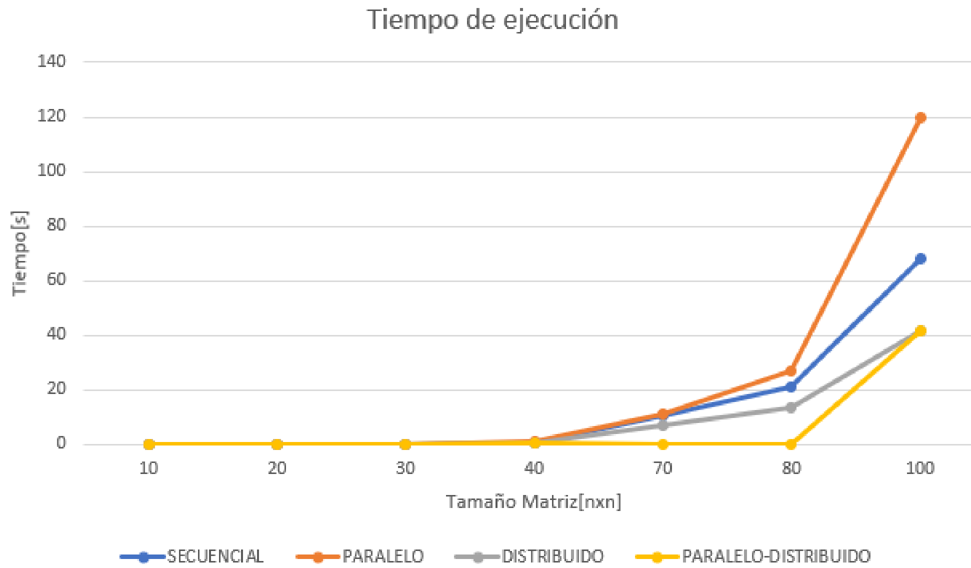


Figura 6.16: Tiempo de ejecución inversa.

Como se puede observar, al variar el tamaño de las matrices la tendencia de la gráfica 6.16 para cualquiera de las técnicas de programación utilizadas es exponencial, ya que, a partir de matrices con tamaños de 40x40 el tiempo que toma resolverlas crece muy rápido, esto por el método utilizado para calcular la inversa.

CAPÍTULO
7

Conclusiones y trabajo futuro

Con el desarrollo del proyecto y la investigación llevada a cabo en la presente tesis, logramos aplicar gran parte de los conocimientos adquiridos a lo largo de nuestra carrera. De igual forma, el desarrollo de esta tesis nos permitió ampliar nuestro conocimiento, adentrándonos al mundo del cómputo de alto rendimiento ya que logramos conocer y comprender algunas de sus técnicas y herramientas para su implementación.

Como se mencionó, el objetivo principal de esta tesis fue determinar bajo qué condiciones es mejor utilizar un sistema secuencial, paralelo o distribuido según la cantidad y el tipo de datos con los que se estén trabajando. Con base en esto y en los resultados obtenidos, podemos afirmar lo siguiente:

En primera instancia, para la implementación de programas que hagan uso de las técnicas de cómputo de alto rendimiento, se requiere de sistemas capaces de soportar las necesidades que este tipo de programación demanda, como son múltiples procesadores o almacenamiento distribuido. Un sistema o clúster de tipo Beowulf es perfecto para este tipo de implementaciones, ya que además de cumplir con los requerimientos necesarios, resulta ser un sistema relativamente barato, sencillo y rápido de armar.

Por otro lado, el trabajar con métodos numéricos en donde se utilizan matrices y se aplican técnicas de cómputo de alto rendimiento, implica una serie de variables que se estudiaron y que, además se llevaron a cabo las pruebas necesarias para determinar

en qué momentos es mejor usar una técnica y en qué momentos otra, o qué tipo de dato es mejor utilizar de acuerdo a las características de la operación o el sistema, por lo que con base en los resultados obtenidos podemos concluir que de acuerdo al número de procesadores, tamaño de las matrices y tipos de datos, se tienen ciertos intervalos en los cuales se obtiene una mejor eficiencia en cuanto tiempo de ejecución, esto es:

- Pasar de una ejecución secuencial a paralela, implica una serie de análisis y cambios en el código, que al ser probados nos permite obtener resultados satisfactorios (en cuanto a tiempo de ejecución) a partir de conjuntos de datos considerables. Esto es, y como se puede observar en la gráfica 7.2, trabajar de forma secuencial o paralela no implica una diferencia considerable de tiempo de ejecución y de tiempo de uso de la memoria cuando se opera con matrices de tamaños de entre 40x40 a 800x800, por lo que en este intervalo se recomienda mantener una implementación y/o ejecución secuencial. Al pasar a un intervalo en donde el tamaño de la matriz varia de 800x800 a 1500x1500 es recomendable hacer uso de la técnica de programación paralela, en este caso con OpenMP y llevar a cabo la ejecución de esta forma. Es en este punto en donde se pueden observar mejoras considerables en cuanto a tiempo de ejecución y tiempo de uso de la memoria al trabajar con una sola computadora y más de un núcleo.

  Por otro lado procesar matrices en donde los tamaños se encuentran en un intervalo de entre 1500x1500 a 2000x2000, se vuelve demasiado pesado y tardado para una sola computadora, aun con múltiples procesadores. Por lo anterior, para este intervalo, se recomienda hacer uso de la técnica de cómputo distribuido y del clúster Beowulf implementado; de esta forma los tiempos de operación vuelven a presentar mejoras. Finalmente para matrices cuyos tamaños superen los 2000x2000, se sugiere implementar un sistema híbrido en donde se utilicen tanto la técnica distribuida como la paralela, con ayuda de las directivas de OpenMP y MPI en conjunto, ya que esto ayuda a mejorar los tiempos de ejecución como se puede observar en la gráfica 7.2 en la curva denominada

PAR-DIS.

- Al trabajar de forma paralela, con múltiples procesadores y en una sola computadora, es de suma importancia determinar el número de procesadores con el que el sistema se comporta de mejor forma o arroja los mejores resultados, ya que tener más procesadores, no siempre garantiza la mayor eficiencia o la mejor forma en que un sistema trabaja. De acuerdo a los resultados obtenidos, podemos recomendar que al trabajar de forma paralela y con OpenMP, el uso de 4 procesadores es lo mejor para arquitecturas similares a las de los procesadores usados en este proyecto.

  Es importante mencionar que, a pesar de que en las gráficas de resultados mostradas en el capítulo 6 en la imagen 6.1 se observe cierta estabilidad de tiempo de ejecución a partir de 4 procesadores en adelante en las cuatro computadoras, no quiere decir que al trabajar con 4, 5 o 6 procesadores, se obtendrá la misma eficiencia o una mejor, ya que se deben tomar en cuenta otros factores.

  Uno de estos es el factor de costo. Como se puede observar en las gráficas de la figura 6.2, este se incrementa conforme el número de procesadores aumenta, por lo que buscar un equilibrio entre el factor de costo y tiempo de ejecución es lo mejor para determinar con cuántos procesadores es mejor trabajar. En este caso sabemos que al trabajar con 4 procesadores el tiempo disminuye considerablemente y a partir de aquí se mantiene con pequeñas variaciones. Sin embargo, aumentar el número de procesadores significaría aumentar el factor de costo de nuestra ejecución.

  Una vez analizado el caso anterior es importante incluir en dicho análisis otros dos factores muy importantes: el factor de aceleración y la eficiencia. En la mayoría de los casos, como se puede observar en las gráficas de la figura 6.3, la mayor aceleración obtenida es al trabajar con 4 procesadores, a excepción de la computadora con procesador *intel core i3* e *intel core 2* los cuales cuentan con

dos núcleos. Finalmente la eficiencia, como se ve en las gráficas de la imagen 6.4, no se ve tan disminuida para 4 procesadores y a pesar de que no es la misma que con 1 o 2 procesadores, al final y al comparar los cuatro factores o métricas mencionadas, la conclusión y recomendación final es, trabajar con el número de procesadores que encuentre el equilibrio con base en estas cuatro métricas de rendimiento.

Adicional a esto, al trabajar con una programación híbrida en donde se ocupen las técnicas paralela y distribuida, podemos concluir que el comportamiento o la tendencia es la misma y resulta ser mucho más eficiente trabajar con el número de procesadores con los que se trabajó de forma paralela.

- Finalmente al trabajar con técnicas de cómputo de alto rendimiento en donde se espera encontrar una forma de eficientizar nuestros programas, es importante tener en cuenta otra variable que aunque a simple vista parece que no afecta mucho el tiempo de ejecución de un programa, a largo plazo sí lo hace; esta es el tipo de dato a emplear.

  Es importante hacer hincapié que en este proyecto se usaron dos tipos de arquitecturas, 32 y 64 bits, la cual también resultar ser una variante importante ya que al trabajar con diferentes tipos de datos el comportamiento cambia en el sentido que a una arquitectura de 32bits le costará el doble de trabajo operar con un tipo de dato double, mientras que a una computadora con arquitectura de 64 bits le sera más sencillo operar, ya que puede incluso realizar una operación en un ciclo de reloj.

  Trabajar con tipos de datos como int y float no genera una gran diferencia en cuanto a tiempo de ejecución entre ambos tipos de datos y por lo tanto al pasar de secuencial a paralelo y de paralelo a distribuido, la aceleración para ambos es casi la misma. Sin embargo al comparar estos dos tipos de datos con double, resultan ser mucho más eficientes, pero al ser double un tipo de dato que es más tardado procesar y al trabajar con este de forma paralela, su aceleración es

mucho mayor. Por el contrario al hacerlo en un sistema distribuido la caída del factor de aceleración se nota mucho, y esto se debe a la tendencia que tienen los nodos menos eficientes del clúster, en este caso con procesadores *atom* y *core i3*. Finalmente short en todos los casos resulta ser el que menos trabajo cuesta procesar pero que ofrece una menor aceleración a la hora de pasar a ejecuciones paralelas y distribuidas.

Para decidir el tipo de dato a emplear en un programa primero se deben considerar las características y requerimientos. Después y tomando en cuenta los resultados, lo más conveniente es utilizar tipos de dato int y float a la hora de trabajar de forma paralela o distribuida, porque de esta forma se obtendrán mejores resultados en las métricas de rendimiento. Si se usa double es importante tener en cuenta que a pesar de que double ofrezca aceleraciones más grandes en el caso paralelo, éstas son proporcionales a la dificultad para procesarse, por lo que aun en paralelo, este seguirá tardando más que int y float. Short al final es lo más sencillo de procesar sin embargo tiene menor representación decimal comparada con int.

Como se puede observar en las gráficas 7.1, 7.2, 7.3 se ve claramente cómo la computadora con el procesador CORE i5 es más eficiente incluso que el clúster mismo al resolver operaciones matriciales. Esto se debe a que la comunicación que se lleva entre los nodos es una comunicación síncrona, es decir, los nodos no van a empezar otra operación hasta que todos los nodos restantes terminen con sus tareas. Esto implica que, y como ya se mencionó, la eficiencia total del clúster se basa en la capacidad y recursos del nodo menos eficiente en el caso de un clúster heterogéneo. Por lo tanto, y de igual forma recomendamos que el clúster que vaya a ser implementado sea de nodos homogéneos para evitar el problema mencionado anteriormente o de lo contrario y si esto no es posible, llevar a cabo el diseño de una política de balanceo de cargas.

Como recomendación adicional si se trabaja con matrices enteras se recomienda que la implementación sea con tipo de dato int ya que es el más eficiente entre los

cuatro tipos de datos y operaciones como suma, resta, multiplicación por escalar, multiplicación de matrices, potencia e igualdad entrega resultados correctos. Sin embargo, si una operación involucra divisiones, el tipo de dato int ya no resulta ser el mejor ya que se pierde precisión y es casi seguro que el resultado esté completamente erróneo; para este caso es mejor utilizar un tipo de dato float ya que es el segundo más eficiente y que tiene mayor precisión.

En resumen si se trabaja con matrices pequeñas que no involucran conjuntos de datos mayores a 800 x 800 , es recomendable trabajar con un programa secuencial, de 800 x 800 a 1500 x 1500 se recomienda trabajar con programas paralelos con OpenMP en una sola computadora o equipo dedicado a esa tarea. Si el trabajo requiere de mayor cantidad de datos, hablando de matrices mayores a 1500 x 1500 pero menores y cercanas a 2000 x 2000 es recomendable trabajar con programas distribuidos.

Por otro lado, si el tamaño de la matriz es mayor a 2000 x 2000 es ampliamente recomendable implementar un sistema híbrido, es decir, con rutinas de MPI en un cluster y con OpenMP en cada nodo que lo conforma. Para saber cuál es el número de procesos más eficiente, es importante considerar por lo menos las primeras 4 métricas de rendimiento antes mencionadas, las cuales son: el tiempo de ejecución, el factor de costo, la aceleración y la eficiencia, que un programa paralelo tiene con respecto a uno secuencial.

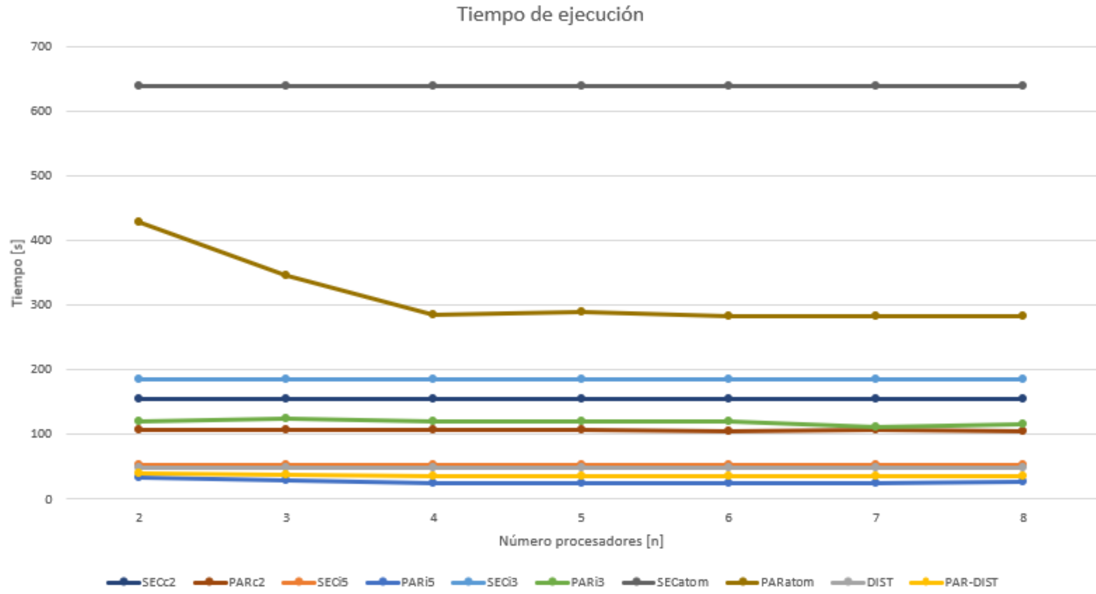


Figura 7.1: Tiempo de ejecución variando número de procesadores.

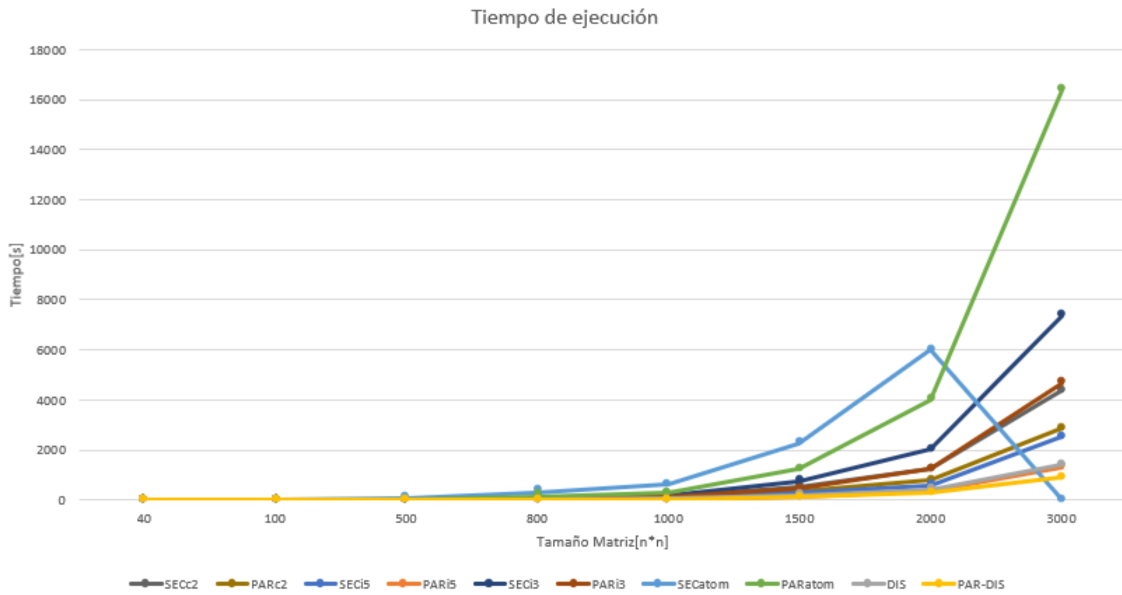


Figura 7.2: Tiempo de ejecución variando tamaño de la matriz

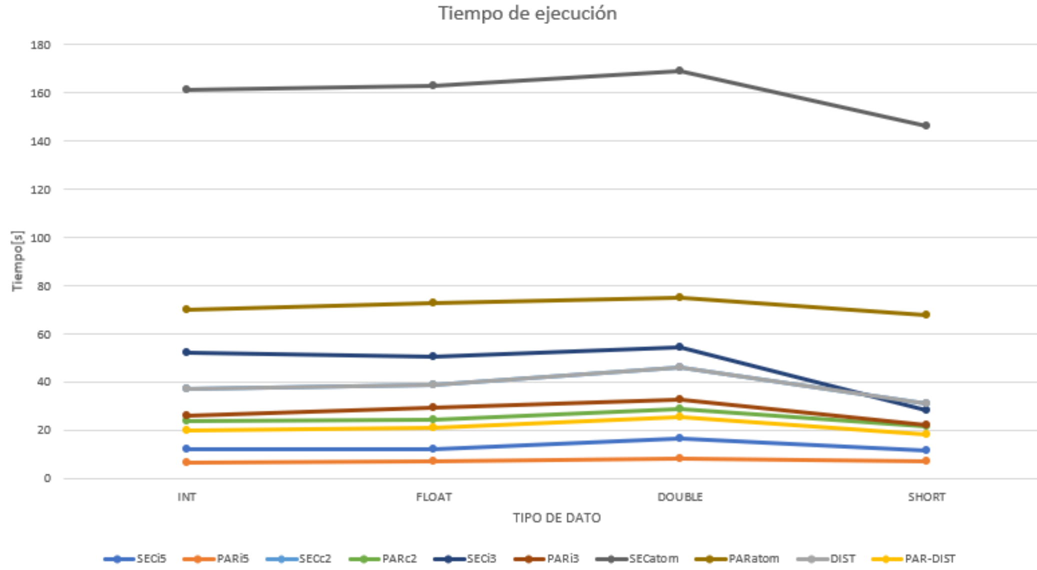


Figura 7.3: Tiempo de ejecución variando el tipo de dato

Cabe resaltar que en la gráfica 7.1 en la curva que pertenece a la computadora *intel atom* se ve una caída ya que no se lograron ejecutar los programas con tamaños de matrices mayores a 2000x2000, siendo este el tamaño máximo ejecutado por esa computadora.

7.0.1 Trabajo Futuro

Como trabajos futuros se propone implementar un algoritmo paralelo para la operación inversa que utilice la eliminación por el método de Gauss, ya que este método es más eficiente computacionalmente. Desafortunadamente no se ha encontrado literatura que ayude a implementar este algoritmo de forma paralela. Otra área de oportunidad, es generar pruebas por cada operación que se realiza en la implementación de esta tesis, de esta forma se podría saber qué algoritmo o qué método es mejor para la resolución de cada operación.

En futuras implementaciones del clúster quitar la interfaz gráfica de cada nodo esclavo así como los puertos o periféricos con los que cuenta cada uno. Esto ayudaría a reducir los procesos que se están ejecutando para dicha tarea y a su vez aumentar

los recursos destinados a la ejecución del proceso.

Por otro lado es importante tener en cuenta que el clúster estará siempre condicionado a la computadora que tenga menos recursos por lo que para mejorar la implementación de las técnicas de procesamiento paralela y distribuida, resultaría conveniente llevar acabo el diseño e implementación de una política de repartición de cargas en donde la cantidad de datos que se le asignen a cada esclavo se base en los recursos de cada uno.

Finalmente se recomienda aplicar más pruebas no funcionales a cada sistema implementado y que todos estos cumplan con las características básicas de desarrollo de software, las cuales se enlistan a continuación:

- Escalabilidad
- Compatibilidad
- Adaptabilidad
- Seguridad
- Estabilidad

Es importante mencionar que estas características se basan o son parte de la ISO/IEC 25010 la cual es un estándar que especifica los requisitos de la evaluación de la calidad del software. [54]

Referencias

[1] S. Mueller, *Upgrading and Repairing PCs*. Que, 2006, vol. 17th Edition.

[2] R. Martínez y A. García, “Breve historia de la informática”, *Universidad Politécnica de Madrid C/ José Gutiérrez Abascal*, pág. 20, 2000.

[3] M. Andrés y G. Isabel, *Introducción a la programación con python*. Universitat Jaume I, 2003.

[4] D. Allen, *How to Think Like a Computer Scientist*. Green Tea Press, 2002, vol. 1.

[5] B. Richard, *Professional Assembly Language*. Wiley, 2005.

[6] C. Vanessa, “Lenguajes de programación”, Tesis de mtría., Universidad Nacional Autónoma de México, Facultad de Ingeniería, México DF, 2007.

[7] S. Angel, *Curso de lenguaje 'C'*. Secretariado de publicaciones de la Universidad de Zaragoza, 1991.

[8] T. Allen y N. Robert, *Programming Languages Principles and Paradigms*, Second edition. MCGraw-Hill, 2007.

[9] AMD, *High-performance computing explained*, [Web; accedido el 04/11/2020]. dirección: `https://www.amd.com/system/files/documents/hpc-explained.pdf`.

[10] NetApp, *What is High-Performance Computing?*, [Web; accedido el 04/11/2020]. dirección: `https://www.netapp.com/data-storage/high-performance-computing/what-is-hpc/`.

[11] Intel. (2014). Introduction to OpenMP, dirección: https://www.youtube.com/watch?v=nE-xN4Bf8XI&ab_channel=OpenMP (visitado).

[12] J. Ayala, *Introducción al Software de HPC*. 2019.

[13] ——, *Introducción al Hardware de HPC*. 2019.

[14] I. general, *Zócalo del Microprocesador*, [Web; accedido el 09/05/2021]. dirección: http://www.informaticageneral.com.uy/software-y-hardware/hardware/zocalo-de-cpu.

[15] E. P. P. López, *Sistemas Operativos II, Memoria Compartida Distribuida*, [Web; accedido el 19/05/2021]. dirección: https://sites.google.com/site/sistemasoperativospaty/unidad-4/unidad-4-memoria-compartida-distribuida.

[16] E. Veliz. (2015). Taxonomías de Flynn. [Web; accedido el 19/09/2020], dirección: https://www.academia.edu/18404647/Taxonom%5C%C3%5C%ADas_de_Flynn?auto=download (visitado).

[17] R. Salado. (2011). Teoría Programación Avanzada UCO 2011. Comunicación entre procesos (IPC). [Web; accedido el 08/11/2020], dirección: http://teoriapa1112.blogspot.com/2011/10/comunicacion-entre-procesos-ipc.html (visitado).

[18] A. Herrero. (2008). Comunicación de procesos a través del paso de mensajes en sistemas distribuidos. [Web; accedido el 08/11/2020], dirección: https://sites.google.com/site/mrtripus/home/sistemas-operativos-2/2-4-comunicacion-de-procesos-a-traves-del-paso-de-mensajes-en-sistemas-distribuidos (visitado).

[19] R. Butler, *MPI*, [Web; accedido el 04/11/2020]. dirección: https://www.cs.mtsu.edu/~rbutler/courses/pp6430/www.llnl.gov/computing/tutorials/workshops/workshop/mpi/MAIN.html.

[20] F. Restrepo. (). Programación concurrente. [Web; accedido el 08/10/2020], dirección: `http://ferestrepoca.github.io/paradigmas-de-programacion/progconcurrente/concurrente_teoria/index.html` (visitado).

[21] E. libro de Python, *Acoplamiento de programación*, [Web; accedido el 01/05/2021]. dirección: `https://ellibrodepython.com/acoplamiento-poo#:~:text=El%20acoplamiento%20puede%20ser%20de,tiene%20dependencias%20internas%20con%20otros.`.

[22] G. William, L. Ewing y S. Anthony, *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1999.

[23] M. P. I. Forum, *MPI: A Message-Passing Interface Standard Version 3.1*, [Web; accedido el 01/11/2020]. dirección: `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

[24] B. Barney. (año no publicado). Message Passing Interface (MPI), dirección: `https://computing.llnl.gov/tutorials/mpi/` (visitado).

[25] F. Bernal, C. Albarracín y et.al., *Programación Paralela*, [Web; accedido el 04/11/2020]. dirección: `http://ferestrepoca.github.io/paradigmas-de-programacion/paralela/paralela_teoria/index.html`.

[26] B. Wilkinson y M. Allen, *Parallel Programing. Techniques and Applications using networked Workstations and Parallel Computers*. PEARSON, Prentice Hall, 2005, vol. Segunda Edición.

[27] G. Hager y G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2019.

[28] R. Baeza-Yates y C. Oviedo. (). Arquitectura de Computadores-Comparación de UMA y NUMA. [Web; accedido el 08/10/2020], dirección: `https://users.dcc.uchile.cl/~rbaeza/cursos/proyarq/choviedo/comp_uma_numa.html` (visitado).

[29] OpenMPCon2020. (). OpenMP. [Web; accedido el 08/10/2020], dirección: https://openmpcon.org/about/iwompopenmp/ (visitado).

[30] I. Roque, *Instituto Tecnológico de Roque*, [Web; accedido el 04/11/2020]. dirección: http://itroque.edu.mx/cisco/cisco1/course/module1/1.2.1.3/1.2.1.3.html.

[31] O. Wendell, *CCNA 200-301*. Cisco Press, 2020, vol. 1.

[32] Cisco. (2020). What Is a LAN? [Web; accedido el 09/19/2020], dirección: https://www.cisco.com/c/en/us/products/switches/what-is-a-lan-local-area-network.html (visitado).

[33] NetworkingAcademy. (2020). Introducciòn a IoT, dirección: https://static-course-assets.s3.amazonaws.com/I2IoT20/es/index.html (visitado).

[34] T. Sterling, *Beowulf Cluster Computing With Linux (Scientific And Engineering Computation Series)*. MIT Press, 2002.

[35] L. Hernández, A. Santillán y R. Caballero, "Maestros y esclavos. Una aproximación a los cúmulos de computadoras", *Revista Digital Universitaria (RDU)*, vol. 4, n.º 2, págs. 3-9, 2013, [Web; accedido el 07/10/2020], ISSN: 1607-6079. dirección: http://www.revista.unam.mx/vol.4/num2/art3/art3.htm#mixbaal.

[36] D. Manrique, "Construcción y evaluación de desempeño de un cluster de tipo Beowulf para cómputo de alto rendimiento.", Tesis de mtría., Universidad Nacional Autónoma de México, Facultad de Ingeniería, México DF, oct. de 2001.

[37] C. Flores, L. Flores y E. Ramírez, *Elaboración de practicas de programación distribuida en el cluster tipo Beowulf de la Facultad de Ingeniería*, México CDMX, 2004.

[38] S. Mariño, M. Godoy, P. Alfonzo, J. Acevedo, L. Gómez y A. Fernández, "Accesibilidad en la definición de requerimientos no funcionales. Revisión de herramientas", *Multiciencias*, vol. 12, n.º 3, págs. 305-312, 2012, [Web; accedido el 03/12/2020], ISSN: 1317-2255. dirección: `https://www.redalyc.org/articulo.oa?id=90426810009`.

[39] I. Sommerville, *Ingeniería de software*, Novena edición. Pearson, 2011.

[40] ISO, *ISO 25010*, [Web; accedido el 01/12/2020]. dirección: `https://iso25000.com/index.php/normas-iso-25000/iso-25010`.

[41] M. Aguilar, C. Arturo, V. Miguel y et. al., *Matemáticas Simplificadas*. CONAMAT. PEARSON EDUCACIÓN. México, 2009, vol. 1640 pp.

[42] B. José, *Matemáticas Básicas Matrices UNAM*, [Web; accedido el 27/10/2020]. dirección: `http://132.248.164.227/publicaciones/docs/apuntes_matematicas/33.%20Matrices.pdf`.

[43] EcuRed, *Operación matemática*, [Web; accedido el 03/12/2020]. dirección: `https://www.ecured.cu/Operaci%C3%B3n_matem%C3%A1tica`.

[44] L. Nieto, R. Cuevas y A. Feliciano, "Herramienta de ingeniería para el estudiante: calculador de matrices.", *Revista Vínculos*, vol. 13, n.º 1, págs. 56-64, 2016, [Web; accedido el 07/10/2020], ISSN: 1794211X. dirección: `http://pbidi.unam.mx:8080/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=asn&AN=136109405&lang=es&site=eds-live`.

[45] J. Gentle, *Matrix Algebra*. Springer International Publishing, Cham, 2017, vol. 1640 pp.

[46] C. Zurita y D. Elia, *Matemáticas con aplicaciones : cálculo integral de una variable, cálculo diferencial de varias variables y álgebra matricial*. Cengage Learning, 2014, vol. 561 pp.

[47] C. Drifter. (2020). Linux Mint 20 Úlyana'- Cinnamon (64-bit) - Linux Mint, dirección: `https://www.linuxmint.com/edition.php?id=281` (visitado).

[48] T. Saito, T. Kito, K. Umesawa y F. Mizoguchi, “Architectural defects of the secure shell”, en *Proceedings. 13th International Workshop on Database and Expert Systems Applications*, 2002, págs. 22-28.

[49] W. Press, S. Teukolsky y B. Flannery, *Numerical Recipes in C. The art of scientific computing*. Cambridge University Press, 1996, vol. Second Edition.

[50] J. Ayala, *Evaluación de Desempeño*. México CDMX, 2019.

[51] A. C. Govantes S., *Criterios para la selección de aseguramientos de calidad en torno a tecnologías de información*. México CDMX, 2005.

[52] G. Everett, *Software testing : testing across the entire software development life cycle*. Piscataway, NJ Hoboken, 2007.

[53] S. Guru, *Pruebas de Software*, [Web; accedido el 01/05/2021]. dirección: `https:%20//sg.com.mx/buzz/evento-sg/sg-virtual-4-abril-2013`.

[54] P. I. 25000, *La familia de normas ISO/IEC 25000*, [Web; accedido el 03/12/2020]. dirección: `https://iso25000.com/index.php/normas-iso-25000`.