Taller - Creación de Cluster con MPI

Daniel Ramirez
Guillermo Aponte
Samuel Pico
Ana Sofía Arboleda
Paula Gabriela Losada
Luis Enrique Santos
Santiago Hernández Rendón
Pontificia Universidad Javeriana
Bogotá, Colombia

I. Objetivos

I-A. Objetivo General

Evaluar el desempeño y la escalabilidad de un algoritmo de multiplicación de matrices paralelizado con MPI, analizando su comportamiento en términos de tiempo de ejecución, speedup, eficiencia y estabilidad al variar el tamaño de la matriz y el número de cores.

I-B. Objetivos Específicos

- Analizar el comportamiento del tiempo de ejecución del algoritmo bajo diferentes configuraciones de paralelización, evaluando su estabilidad mediante métricas estadísticas como promedio, desviación estándar e intervalos de confianza.
- Determinar el grado de aceleración (speedup) alcanzado al incrementar el número de núcleos de procesamiento, comparando los resultados obtenidos con el modelo ideal de escalabilidad lineal.
- Evaluar la eficiencia de paralelización del algoritmo, cuantificando el aprovechamiento de los recursos computacionales disponibles e identificando posibles pérdidas de rendimiento asociadas a sobrecarga o limitaciones inherentes al modelo paralelo.

II. Marco Teórico

En la sección actual se exploran las principales definiciones sobre los sistemas distribuidos, así como las tecnologías utilizadas y las métricas de los resultados.

II-A. Sistemas Distribuidos y Desafios

Dentro de las numerosas definiciones de lo que es un sistema distribuido, la mas simple es la que da [1] en su documento .*A brief introduction to distributed systems"*, en donde dice que un sistema distribuido es üna colección de equipos de cómputo autónomos, que utilizan un único sistema coherente". Esta definición trae a flote las dos principales características de un sistema distribuido, como es una colección de más de una computadora, y el hecho de que las computadoras sigan una misma lógica. Esto es muy de la mano con la definición que da [2], ya que se refiere a la utilización de múltiples dispositivos informáticos, en diferentes ubicaciones, para un único propósito; refiriéndose a la coherencia del sistema.

La red de equipos informático es visto como un solo sistema, y la comunicación entre los equipos es realizada mediante una red LAN para equipos cercanos entre sí, o una red WAN para componentes de la red que están separados de manera geográfica [2].

En la actualidad, un sistema distribuido busca, no solo ser escalable y aumentar la capacidad de operaciones informáticas, sino de tener un nivel de tolerancia a los fallos, de tal forma de que si un componente falla, el sistema pueda seguir funcionando como uno solo. Un sistema distribuido puede mantener un mayor grado de trazabilidad [3] y tolerancia a fallos. Este nivel de versatilidad permite la aplicación de los sistemas distribuidos a diferentes ámbitos, como lo son el avance del aprendizaje automático [4]. La importancia del análisis a los sistemas distribuidos no radica únicamente en el rendimiento, sino en la tolerancia a fallos, concepto crítico para su aplicación en áreas diferentes a la de la computación [5].

II-B. Computación Paralela

Tradicionalmente, el software ha sido escrito de una forma serial; de manera que solo sea ejecutado en un solo computador, en una única CPU, un problema es construido mediante una serie de instrucciones seriales, y únicamente se ejecuta una instrucción al tiempo. La computación paralela, definida como .el uso simultáneo de múltiples recursos de cómputo para resolver un problema computacional"[6], permite que las instrucciones se ejecuten de manera simultánea en múltiples CPUs. El uso del paralelismo ha sido motivado en parte por la evolución de los microprocesadores, y la evolución de arquitecturas de memoria, entre las que se destacan las arquitecturas de memoria compartida y distribuida [7].

Unos de los principales conceptos de la computación paralela es el paralelismo y la concurrencia, que si bien parecen ser iguales, son conceptos que tienen sus diferencias clave. En primera instancia, el paralelismo es un término que se refiere al momento en el que dos o más procesos estan sucediendo de forma simultánea, y por cada proceso existe una CPU que lo contiene, mientras que la concurrencia es una extensión al

paralelismo; se refiere a la capacidad de gestionar y procesar múltiples tareas en el mismo periodo de tiempo, pero no necesariamente de forma simultánea [8].

El paralelismo maneja diferentes tipos de arquitecturas, las dos mas relevantes son la arquitectura de memoria compartida y la arquitectura de memoria distribuida, esta última de suma importancia para el manejo de MPI, herramienta utilizada en el informe presente. La arquitectura de memoria compartida es en el caso en donde, en un sistema paralelo pueden haber varias CPUs realizando sus tareas y procesos, sin embargo, los recursos de memoria son compartidos entre ellos; los cambios en la memoria realizados por un procesador son visibles para los demás procesadores. Por otra parte, una arquitectura de memoria distribuida ocurre cuando los procesadores tienen su propia memoria local, por lo que si un procesador quiere acceder a los datos de la memoria asociada a otro procesador, este debe hacer uso de los medios de comunicación y los medios de red dados y definidos por el programador, para que de esta manera pueda ocurrir un proceso de sincronización [6]. Esta última arquitectura siempre debe de tener una red de comunicación entre las diferentes CPUs, y es el modo utilizado por MPI [9].

II-C. Clústeres de Computadores

La computación por clústers es definida como una serie de computadores independientes, combinados en un único sistema por software y redes [10]. Un cluster es un sistema de computación distribuida compuesto por un conjunto de computadoras independientes y conectadas entre sí, mediante una red, para funcionar como un único recurso integrado. Se caracteriza por una arquitectura de acoplamiento fuerte, donde los equipos comparten el mismo hardware, software y red, permitiendo distribuir las tareas entre ellos para maximizar el rendimiento y acelerar el procesamiento de datos de manera eficiente [11].

Un tipo especial de clústers, común en los entornos de Linux, es el cluster Beowulf. Este es un sistema de cómputo científico y de ingeniería construido a partir de hardware y software de fácil acceso, específicamente utilizando procesadores x86 [12]. Parte del éxito de los clústers Beowulf está dado por la creación de un modelo de código abierto; su impacto radica en haber capitalizado décadas de experimentación de la comunidad de procesamiento paralelo, impulsado a su vez, por entornos de ejecución linux de código abierto, estudiado en el libro "Beowulf Cluster Computing with Linux", por Thomas Sterling [13].

II-D. MPI (Message Passing Interface)

MPI (Message-Passing Interface), según el MPI Forum, es un estándar de comunicación por mensajes, dando la interfaz de biblioteca para el modelo de programación paralela; especificando su implementación, y diseñada para mover datos entre los espacios de direcciones de distintos procesos de forma cooperativa. No es un lenguaje de programación ni una implementación específica, sino un conjunto de normas definidas por consenso, creando un protocolo de comunicación que permite crear programas portables, eficientes y escalables. El objetivo principal de MPI es ofrecer un marco de trabajo flexible que incluya desde operaciones de comunicación para el desarrollo del software paralelo [14].

El protocolo se maneja de tres maneras diferentes. La más común de estas, SPMD (Same program, different data), es cuándo un proceso está guiado por un programa igual, pero cada proceso tiene datos diferentes, ya sea en memoria distribuida o variable [15]. El modelo es: varios procesos que ejecutan el mismo programa en paralelo, cada uno con un pedazo de datos distinto, comunicándose mediante MPI.

Al tratarse de un protocolo de comunicación, estándar utiliza diferentes tipos de mensajes, definidos por el MPI Forum [14], estos son [15]:

- **MPI_Init:** Inicializa el entorno MPI, debe ser llamada antes de cualquier otra rutina MPI.
- **MPI_Finalize:** Libera recursos del entorno MPI; después de llamarla no se deben invocar más funciones MPI.
- **MPI_Send / MPI_Recv:** Rutinas básicas de comunicación punto a punto, bloqueantes, que envían/reciben mensajes etiquetados entre procesos.
- **MPI_Bcast:** Operación colectiva de broadcast, donde un proceso raíz envía el mismo mensaje a todos los demás procesos en un communicator.
- **MPI_Reduce:** Operación colectiva de reducción, que combina valores de todos los procesos y entrega el resultado a un proceso raíz.

La implementación utilizada en el laboratorio de MPI es Open MPI. El proyecto Open MPI es una implementación de código abierto de la especificación MPI, desarrollada y mantenida por un consorcio de socios académicos, industriales y de investigación. Al integrar la experiencia y recursos de toda la comunidad de computación de alto rendimiento, ofrece una solución robusta y colaborativa para el cómputo paralelo . Sus lenguajes principales de implementación son C++ y Fortran [16].

II-E. Métricas de Rendimiento en Sistemas Paralelos

Existen diferentes métricas utilizadas para poder medir el desempeño de un sistema paralelo. A continuación se explicarán las utilizadas, para que caso y su unidad.

- **Tiempo de ejecución:** Tiempo total que tarda un programa (secuencial o paralelo) en resolver un problema dado, en una plataforma concreta. Normalmente es medido por tiempo de reloj, en segundos, milisegundos, microsegundos, horas o dias para operaciones complejas. [17].
- **Speedup (aceleración):** Mide cuánto se acelera un programa al pasar de 1 procesador a p procesadores, su fórmula es:

$$S(p) = \frac{T_s}{T_p} \tag{1}$$

  En donde T_s es el tiempo secuencial y T_p el tiempo paralelo, con p procesadores [18].

- **Eficiencia:** Mide qué fracción de la capacidad teórica de los procesadores se está aprovechando:

$$E(p) = \frac{S(p)}{p} = \frac{T_s}{pT_p} \tag{2}$$

  Si $E(p) = 1$ significa que se está aprovechando el 100 % de la capacidad [17].
- **Escalabilidad:** Capacidad de un sistema/algoritmo paralelo para mantener o mejorar su rendimiento al aumentar bien sea el número de procesadores, o el tamaño del problema. Se mide mediante aceleración, eficiencia y tiempo de ejecución [19].
- **Overhead:** Tiempo extra dedicado a comunicación, sincronización y coordinación entre procesos, que no contribuye directamente al cómputo útil. Esto incluye la latencia del mensaje, el tiempo de espera por sincronización, y los tiempos de carga por procesos previos adicionales [20]. El overhead de comunicación puede ser modelado como: [21]

$$T_{comm} = \alpha + \beta \cdot m \tag{3}$$

  En donde α y β son latencia y tiempo por byte, y m es el tamaño del mensaje
-

III. Especificaciones de los Equipos de Cómputo

Para esta practica se utilizaron 5 computadores portátiles del laboratorio Alan Turing, en la Pontificia Universidad Javeriana, los cuales contaban las mismas especificaciones, ya que eran el mismo modelo, lo cual nos permitía contar con un entorno mas homogéneo, evitando diferencias en el rendimiento de cada computador

III-A. Características Generales de los Equipos

Cada uno de los equipos cuenta con las siguientes especificaciones:

- Procesador Intel Core i7-6500U con frecuencia base de 2.50 GHz y hasta 3.10 GHz.
- 2 núcleos físicos y 4 hilos de procesamiento.
- 8 GB de memoria RAM DDR4.
- Sistema operativo linux, corriendo de manera local, el cual nos permitió la creación del cluster y la ejecución de procesos distribuidos.

IV. Procedimiento Para Crear el Cluster

IV-A. Requisitos

Para la creación del cluster evaluado en el presente taller, se necesita una serie de requisitos tanto en hardware como en software, explicados a continuación.

- Varios equipos de cómputo, que se interconectarán para trabajar en conjunto en las tareas que se les sean asignadas. En este caso, fueron usados 5 computadores portátiles del laboratorio Alan Turing en la Pontificia Universidad Javeriana.
- Una misma red de área local (LAN) a la que se deben conectar todos los équipos que conformarán el clúster, esto es necesario ya que es indispensable la conexión mediante una red estable y rápida, que permita a los equipos coordinarse, repartir el trabajo y compartir resultados.
- Un mismo sistema operativo, preferiblemente una misma distribución de linux en todos los equipos. Si bien, este requisito no es estrictamente necesario, pues en la teoría un clúster se podría realizar con diferentes sistemas operativos, en la práctica, resulta mucho mas factible la realización de este usando un solo SO en todas las máquinas, así se evitan problemas de incomparibilidad de librerías, diferencias de versiones o incluso la inexistencia de algunas herramientas en sistemas diferentes a Linux. Siendo así, el presente experimento fue realizado utilizando Ubuntu en todos los équipos.
- El estándar de librerías MPI, que permite a los computadores enviar y recibir datos por la red, de forma sincronizada, optimizada y estándar, permitiendo a los computadores trabajar juntos como un solo sistema.

IV-B. Creación de la red

1. Asignar Hostnames: Al asignar un hostname en cada máquina, lo que se hace es cambiar sus nombres en la red, permitiendo que MPI conozca estos nombres y los relacione con sus IP´s, facilitando el proceso de comunicación. Para el presente experimento, se le asignó un nombre nodeN, a cada máquina, comenzando desde 0 hasta 4, según su ubicación en la sala. Este proceso se hace con el comando ***"sudo hostnamectl set-hostname (nombre a colocar)"***. Ejemplo:

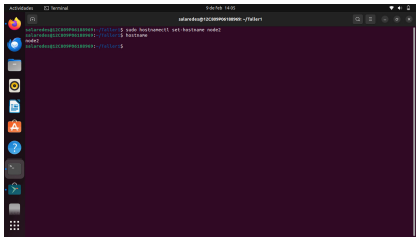

Figura 1. Ejemplo de como fijar el hostname en una máquina.

2. Optener las IP: Para que todos los nodos se puedan comunicar, se necesita saber la ip de estos, que posteriormente utilizará MPI para hacer que el cluster funcione, el comando para obtener esta ip es: ***ïp a"***, este comando muestra las interfaces de red, las direcciones ip, las direcciones MAC y el estado de red, de toda información, la que se usará es las direcciones ip. Ejemplo:
3. Editar el archivo de hosts dentro de la carpeta etc: Para que MPI pueda interactuar con todos los nodos, lo debe hacer con un nombre asignado, no directamente con su IP, esto facilita el entendimiento de los comandos realizados. Esto se logra modificando el archivo de hosts, en la carpeta etc, que relaciona direcciones ip con nombres en el sistema Linux MPI accede a este archivo para realizar las acciones correspondientes. Para moficarlo se usó el siguiente comando: ***"sudo nano /etc/hosts"***,

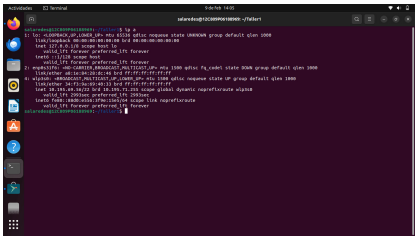

Figura 2. Ejemplo de como ver la ip del equipo.

en este caso, se debe usar sudo, pues este archivo se modifica unicamente con permisos de administrador, se usa el editor de texto nano y finalmente se indica la ruta del archivo a modificar. Posteriormente se coloca en el archivo la ip de cada nodo, seguido de su nombre, modificado anteriormente Ejemplo:

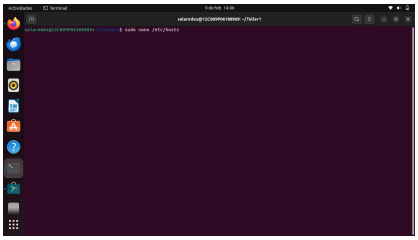

Figura 3. Ejemplo de como editar /etc/hosts.

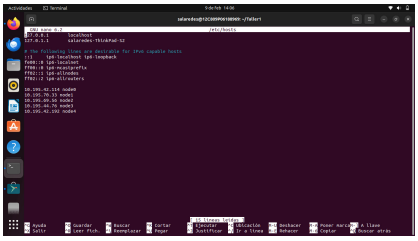

Figura 4. Ejemplo de como debe quedar el archivo hosts.

4. SSH sin contraseña: Para que MPI pueda funcionar, es necesario que este pueda realizar las comunicaciones e instrucciones de todas las máquinas sin que estas le soliciten contraseña, pues, de este ser el caso, el proceso se bloquearía. Las comunicaciones se realizan usando SSH, que significa Secure Shell, siendo este un protocolo que permite controlar otro computador de forma segura, remota y encriptada. Para evitar la solicitud de contraseña se debe crear una ssh key, que representa un par de claves criptográficas, una privada que se queda en la máquina que la genera y una pública que se copia en los otros nodos, cada vez que MPI lanza procesos remotamente mediante ssh, el servidor verifica las claves, y si coinciden, permite manipular el computador sin necesidad de su contraseña. Para generar la llave se usa el comando ***"ssh-keygen"*** y para enviarla a los demás nodos se usa ***"ssh-copy-id nodeN"***. Finalmente, se verifica que haya funcionado usando el comando ***"ssh nodeN"*** Ejemplo:

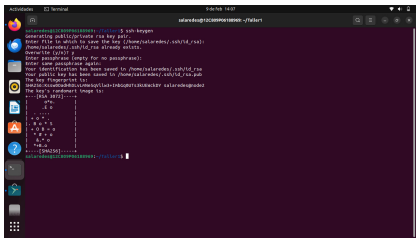

Figura 5. Ejemplo de crear la ssh key.

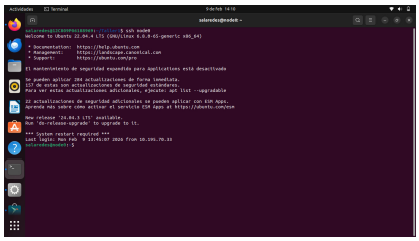

Figura 6. Ejemplo de como verificar la conexión ssh.

IV-C. Uso de MPI

Una vez se configura la red en la que operará el clúster, es necesario configurar MPI en todos los equipos, de esta forma el clúster podrá operar correctamente.

1. Instalar MPI: Para instalar las librerías de MPI, en todos los equipos se debe ejecutar el siguiente comando: ***"sudo apt install -y openmpi-bin openmpi-common libopenmpi-dev"***. El comando se compone de las siguientes partes.
   - sudo: Indica que el comando se ejecutará con permisos de administrador.
   - apt install: Indica que lo que se quiere es instalar nuevos paquetes en el dispositivo.
   - -y: Indica que se debe responder que si a todas las preguntas.
   - openmpi-bin: Instala el paquete que contiene los ejecutables de OpenMPI sea mpirun, mpicc, mpiexec, etc.
   - openmpi-common: Instala el paquete que permite crear archivos compartidos y algunas configuraciones básicas de OpenMPI.

   Finalmente se usa el comando ***"mpirun –version"*** para ver que se haya instalado correctamente. Ejemplo:
2. Indicarle a MPI cuales nodos hay: Para que MPI pueda funcionar correctamente, se le debe indicar de cuantos nodos se dispone para el clúster y cuantos procesos puede ejecutar cada nodo. Para esto se crea un archivo "hostfile", en donde se indica el nombre de cada nodo al que MPI puede acceder y que previamente han sido guardados con su ip en /etc/hosts/, seguido de cuantos procesos MPI puede ejecutar cada nodo al mismo

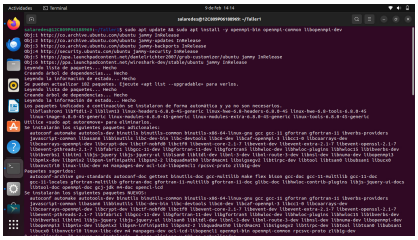

Figura 7. Ejemplo de como instalar MPI.

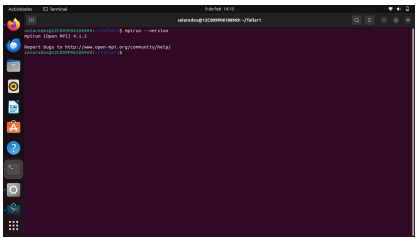

Figura 8. Ejemplo de como verificar la instalación de MPI.

tiempo, estos procesos, generalmente dependen de la cantidad de núcleos que contiene el procesador de la máquina que está fungiendo como nodo del clúster. De esta forma, al ejecutar el comando de ejecución, se le indica el archivo, para que MPI lo lea y ejecute el programa con los respectivos nodos. El archivo se crea con el comando ***"nano hostfile"***. Ejemplo:

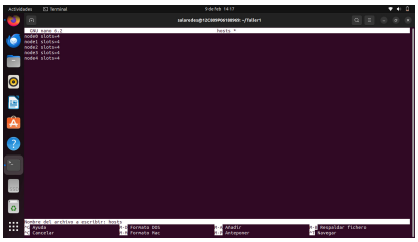

Figura 9. Ejemplo del archivo hostfile.

IV-D. Funciones MPI en código

Para el uso de MPI durante la distribución para códigos en C, es necesario emplear una serie de funciones de la librería, que permite la comunicación entre los diferentes nodos del cluster.

1. MPI_Init(&argc, &argv): Arranca el entorno MPI, recibe se le envía como parámetro los argumentos ingresados al ejecutar el código, correspondiente a la configuración del cluster, al indicarle la cantidad de subprocesos a implementar.
2. int MPI_Comm_rank(MPI_Comm comm, int *rank): Función que le asigna un identificador único a cada proceso dentro del comunicador. Recibe un parámetro MPI_comm comm, que define los grupos de procesos que se están mirando entre sí y int *rank, que es donde MPI escribe el número del proceso.
3. int MPI_Comm_size(MPI_Comm comm, int *size): Obtiene el número total de procesos activos dentro del communicador. Recibe el grupo de procesos y int *size, que es donde MPI escribe cuantos procesos existen en el comunicador.
4. int MPI_Abort(MPI_Comm comm, int errorcode): Termina inmediatamente la ejecución de todos los procesos MPI dentro del comunicador sin liberar memoria ni sincronizar. Recibe el grupo de procesos y int errorcode, que es el código de error que se devuelve al sistema operativo.
5. int MPI_Barrier(MPI_Comm comm): Funciona como una barrera de sincronización, obligando a que todos los procesos del comunicador lleguen a esa sección de código para que puedan continuar. Recibe el grupo de procesos.
6. int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm): Divide un arreglo grande en pedazoa iguales y los reparte entre todos los procesos. Recibe sendbuf, que es el arreglo completo en el proceso root, sendcount que es la cantidad de elementos que le envia a cada parámetro, sendtype, que es el tipo de dato, recvbuf, que es un arreglo local donde cada proceso guarda su parte, recvcount, que es cuantos arreglos recibe cada proceso, recvtype que es el tipo de dato recibido, root, que es el proceso que envia información a los demás y finalmente comm, que es el número de procesos.
7. int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm): Envía un solo bloque de datos, desde un mismo proceso, el proceso raíz, a los demás procesos. Recibe buffer, el arreglo que será enviado desde la raíz, count, que es la cantidad de datos a transmitir, datatype que es el tipo de dato, root, que es el proceso raíz y comm, que es el grupo de procesos.
8. int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm): Toma los fragmentos calculados por cada proceso y lo une en un solo arreglo final en el proceso raíz. Recibe sendbuf, que son los datos locales de cada proceso, sendcount que es la cantidad de datos que envia cada proceso, sendtype que es el tipo de dato que será enviado, recvbuf que es el buffer donde el root recibirá todo, recvcount que representa cuanto recibirá el buffer de cada proceso, recvtype que es el tipo de dato que recibirá, root, que es el proceso que juntará todo y comm, que es el grupo de procesos.
9. int MPI_Finalize(void): Cierra el entorno MPI de forma correcta, liberando recursos, cerrando canales de comunicación y sincronizando todos los procesos antes de que terminen.

IV-E. *Compilación y ejecución del código*

Cuando MPI está configurado para operar el cluster, solo resta ejecutar el código que se desee distribuir con ciertos parámetros que le indican al cluster como hacerlo. Esta ejecución se debe realizar desde un solo equipo, que le enviará las señales a los demás.

1. Compilación: Para compilar un código en C que use MPI, se debe utilizar un comando con la siguiente estructura: ***"mpicc matmul.c -O3 -o matmul"***.
   - mpicc: Es el compilador de MPI para C, en realidad, es una mezcla sobre gcc, el compilador normal. Cumple la función de añadir automaticamente las librerias y headers necesarias para MPI, manejar los enclaes a OpenMPI y facilitar que el programa pueda manejar comandos de MPI como Init o Send.
   - matmul.c: Es un archivo fuente en C de ejemplo, en este caso, fue el utilizado para la multiplicación de matrices. Es indispensable que este archivo incluya por lo menos la libreria mpi.h con #include <mpi.h>para poder usar las funciones de MPI.
   - -O3: Es una opción de optimización del compilador, que le brinda máxima optimización, para que el compilador intente Vectorizar buvles, reordenar instrucciones y eliminar código, de tal forma aumenta la velocidad. Esto se debe usar en clusters, pues la velócidad es un tema crítico en su uso.
   - -o matmul: Es opcional, sirve para indicar el nombre que tendrá el ejecutable final, en este caso, se le coloca matmul.

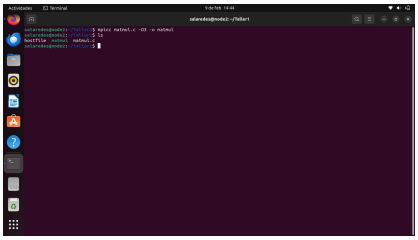

Figura 10. Ejemplo de la compilación del programa.

2. Ejecución: Para ejecutar un código compilado con MPI, se usa un comando como este: ***"mpirun -np 20 –hostfile hostfile ./matmul"***.
   - mpirun: Es el ejecutor de MPI, arranca los procesos MPI en los nodos indicados y coordina la comunicación, trabaja con ssh.
   - -np 20: Indica el número de procesos que ejecutarán el programa, en este caso, como se usan 5 nodos con 4 núcleos cada uno, se indica que el programa se distribuirá en 20 procesos diferentes.
   - –hostfile hostfile: Indica el archivo en donde se menciona cuáles son los nodos que existen y cuantos slots o núcleos tienen, mpi lo usa para decidir donde iniciar cada proceso y cómo conectarlos.
   - ./matmul: Es el ejecutable que se compilo anteriormente, contiene el código que se desea distribuir y cada proceso MPI ejecutará una copia de este programa en paralelo.

   Ahora, se adjunta una captura de pantalla del comando top, que muestra los recursos que están siendo usados en el computador. Para una ejecución como la anterior, donde se usan todos los slots de todos los núcleos, el gráfico deberia ser igual en todos los PC.

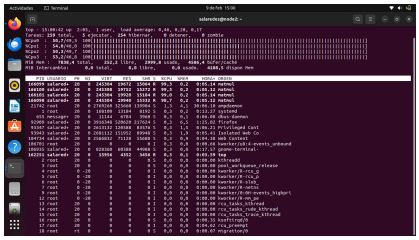

Figura 11. Consumo de recursos luego de la ejecución.

V. Diseño de Experimentos

V-A. *Objetivo del experimento*

En esta sección tuvimos como objetivo comprobar, con datos medibles, cómo cambia el rendimiento de la multiplicación de matrices cuando:

- Aumenta el tamaño de la matriz N x N
- Aumenta la cantidad de procesos MP(np)
- Se ejecuta en un solo equipo vs distribuido en el clúster

La idea es responder, de forma concreta y verificable:

1. ¿En qué tamaños distribuir ayuda de verdad?
2. ¿En qué tamaños distribuir no vale la pena porque coordinarse cuesta más que calcular?

V-B. *Variables del experimento*

- **tamaño de matriz (N):** Se usaron tamaños cuadrados N × N, tomando diferentes valores(200, 400, 800, 1600, 3200).
  La razón de probar varios tamaños es simple:
  1. una matriz pequeña se calcula rápido, pero repartirla por red puede ser caro.
  2. una matriz grande tarda mucho, y ahí paralelizar puede empezar a ayudar.
- **número de procesos MPI(np):** Se probaron las siguientes configuraciones de procesos:
  - np = 4
  - np = 20

  Como ya se había mencionado anteriormente, np es el número de procesos que se ejecutan en un mismo programa.
  Para que el programa sea exacto ya que se divide el trabajo por bloques, se requiere que todos los valores de N sean divisibles por todos los valores de np.

- **Número de repeticiones (runs):** cantidad de veces que se ejecuta exactamente el mismo caso (mismo N y mismo np) para obtener un promedio estable.
- **Entorno de ejecución:** mismos equipos, misma red ymismas condiciones de laboratorio para evitar cambios que alteren el tiempo (por ejemplo, cargas distintas en los portátiles).

V-C. Constantes del experimento

- **Hardware homogéneo:** mismos equipos del laboratorio (mismo modelo y especificaciones).
- **Sistema operativo homogéneo:** misma distribución Linux en todos los nodos.
- **Red y condiciones de laboratorio:** misma red local y misma sesión de laboratorio, evitando cargas externas innecesarias.
- **Binario y flags:** se mantiene el mismo ejecutable durante toda la serie de mediciones y se compila con optimización alta (`-O3`). (La compilación y ejecución ya fueron descritas en la Sección V)
- **Hostfile:** se mantiene el mismo archivo de nodos/slots para garantizar que el mapeo de procesos por nodo sea consistente. (La estructura del hostfile y el significado de *slots* ya se explicó en la Sección V.)

V-D. Métricas

V-D1. Tiempo promedio de ejecución: Si para un caso (N, np) se obtienen r mediciones $t_1, t_2, \ldots, t_r$, el promedio se calcula como:

$$\bar{t}(N, np) = \frac{1}{r} \sum_{i=1}^{r} t_i$$

Adicionalmente, se calcula dispersión con la desviación estándar:

$$s(N, np) = \sqrt{\frac{1}{r-1} \sum_{i=1}^{r} (t_i - \bar{t})^2}$$

Interpretación simple: $\bar{t}$ es el "tiempo típico" del caso y s muestra qué tanto varía entre corridas.

V-D2. Intervalo de confianza (95 %): Para reportar estabilidad, se usa un intervalo de confianza del 95 % para el promedio:

$$IC_{95\,\%} = \bar{t} \pm t_{\alpha/2, r-1} \cdot \frac{s}{\sqrt{r}}$$

donde $t_{\alpha/2,r-1}$ es el valor crítico de la distribución t de Student.

Interpretación simple: el intervalo indica un rango donde se espera que esté el promedio real, considerando variabilidad entre corridas.

V-D3. Speedup (aceleración): Como en este laboratorio no se usó un caso explícito de 1 solo proceso como referencia universal, el speedup se define con referencia al caso medido de menor paralelismo disponible (por ejemplo $np = 4$):

$$S_{np|4}(N) = \frac{\bar{t}(N, 4)}{\bar{t}(N, np)}$$

Interpretación simple: si $S_{20|4} = 2$, entonces con $np = 20$ fue el doble de rápido que con $np = 4$ para el mismo N.

V-D4. Eficiencia: La eficiencia se define como:

$$E_{np|4}(N) = \frac{S_{np|4}(N)}{np/4}$$

Interpretación simple: mide qué tan cerca está el speedup observado del ideal (donde crecer procesos debería acelerar).

V-D5. Overhead (costo extra por distribuir): Para cuantificar el costo adicional de ccomunicación, se estima un "tiempo ideal" suponiendo escalabilidad:

$$t_{\text{ideal}}(N, np) = \bar{t}(N, 4) \cdot \frac{4}{np}$$

y el overhead se calcula como:

$$O(N, np) = \bar{t}(N, np) - t_{\text{ideal}}(N, np)$$

Interpretación simple: overhead es el tiempo "extra" que aparece por repartir, sincronizar y comunicar, comparado con un mundo ideal.

V-E. Recolección de datos y evidencia (CSV)

Un CSV(Comma-Separated Values) es un archivo de texto estructurado para guardar información en forma de tabla. Cada línea es una fila y cada dato es una columna, separada por coma o punto y coma.

V-E1. Evidencia mínima por caso: Cada caso (N, np) genera un archivo CSV con el patrón:

`matricesde<N>_np<np>.csv`

Cada fila representa una corrida y registra: `run` (número de repetición) y `time_seconds` (tiempo total de ejecución).

V-E2. Por qué se usan múltiples corridas: Una sola ejecución puede variar por: carga del sistema, variaciones de red, o planificación de procesos. Repetir $runs$ veces permite:

- estimar un promedio estable,
- medir variabilidad,
- y calcular intervalos de confianza.

V-F. Automatización de pruebas

Para esta sección se realizo un script el cual automatiza las corridas repeticas y genera los CSV. Este Script se llamó benchmark$_m$*pi.pl*

V-F1. ¿Qué es un archivo con extensión .pl?: Un archivo con extensión .pl es un script escrito en Perl. Perl es un lenguaje muy usado para automatización y procesamiento de texto. Un .pl no es un binario compilado: es un archivo de texto que se ejecuta normalmente con el siguiente comando: "perl benchmark$_m$*pi.pl*"

En el experimento se uso el .pl para:

- Evitar correr todo a mano.
- Ejecutar muchas veces cada caso para que el resultado sea más confiable.
- Dejar evidencia organizada (CSV).

V-F2. *Funcionamiento interno del script:*

1. Define una lista de tamaños N y valores de np.
2. Para cada combinación (N, np), repite runs veces:
   - ejecuta mpirun/matmul
   - captura el tiempo que imprime el programa
   - registra ese tiempo en el CSV correspondiente
3. Produce un archivo por caso, con estructura uniforme.

V-G. *Control de calidad de las pruebas*

Se consideró que un caso (N, np) es válido si:

- el programa termina sin errores de MPI,
- el CSV contiene exactamente $runs$ mediciones,
- los tiempos registrados son numéricos y no están vacíos,
- y se mantuvieron constantes el binario, hostfile y condiciones generales del laboratorio.

VI. Análisis de Resultados

VI-A. *Tiempo promedio de ejecución*

La gráfica de promedios (ver figura 12) incluye un intervalo de confianza del 95 %, lo que indica que, con un 95 % de confianza, el valor real del promedio se encuentra dentro de ese rango estimado. Asimismo, se observa que la distribución no resulta conveniente a menos que el tamaño de la matriz sea lo suficientemente grande. En matrices pequeñas o medianas, el overhead asociado a la comunicación y sincronización entre procesos hace que la multiplicación distribuida tarde más que la ejecución en un solo equipo. En nuestros experimentos, la mejora en rendimiento solo se hizo evidente a partir de matrices de tamaño 3200×3200, ya que en el caso de 1600×1600 la distribución aún se mostraba ineficiente. Asimismo, se observa un crecimiento significativo en los tiempos promedio, lo cual se explica por la complejidad temporal del algoritmo de multiplicación de matrices, que es $O(n^3)$ (ver figura 13). En consecuencia, cuando el tamaño de la matriz se duplica, el tiempo de ejecución aumenta al menos en un factor de 8 (sin considerar el overhead asociado a la distribución y comunicación entre procesos).

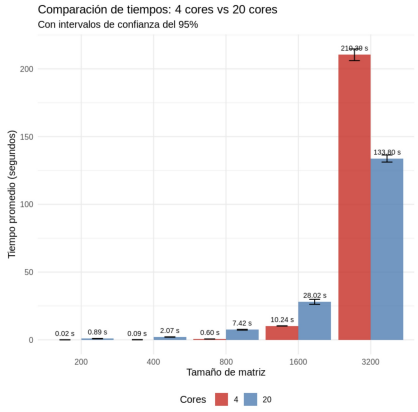

Figura 12. Comparación de tiempos - Tiempo vs Tamaño de matriz.

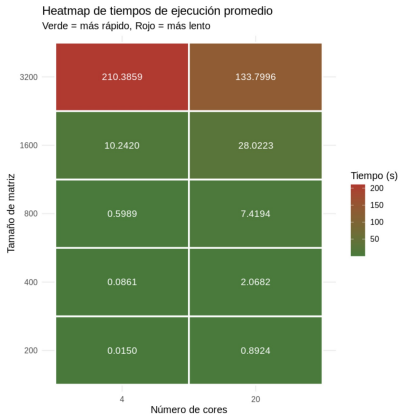

Figura 13. Heatmap de tiempos de ejecución promedio.

VI-B. *Eficiencia de paralelización*

También se evaluó la eficiencia del paralelismo; sin embargo, es importante señalar que un análisis más riguroso requeriría un tamaño de muestra mayor para obtener conclusiones más sólidas. Según la gráfica presentada (ver Figura 14), el comportamiento observado está lejos del ideal teórico, en el cual se aprovecharía el 100 % de los núcleos distribuidos utilizados. La pérdida significativa de eficiencia se debe principalmente a los costos de comunicación asociados a las operaciones MPI_Scatter, MPI_Bcast y MPI_Gather. Adicionalmente, la latencia de red entre los portátiles influyó de manera considerable, especialmente porque el experimento se realizó utilizando conexión Wi-Fi en lugar de una red cableada, lo que incrementa los tiempos de transmisión y sincronización.

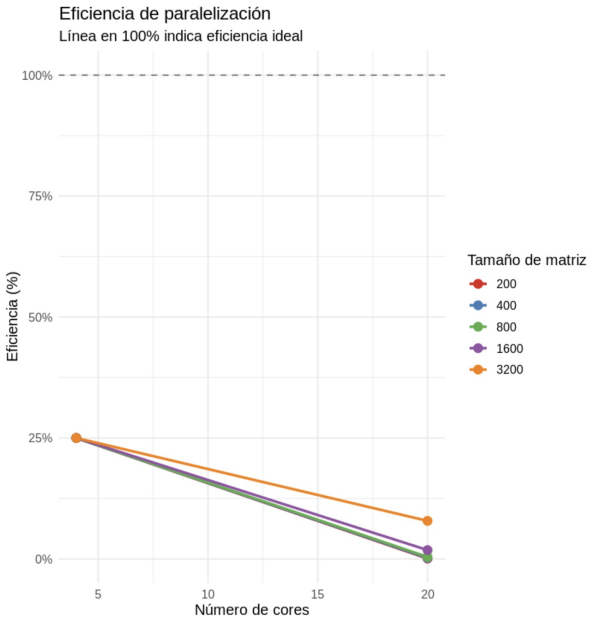

Figura 14. Gráfica indicadora de eficiencia de paralelización.

VI-C. *Overhead de paralelización*

El overhead se calculó como la diferencia entre el tiempo de ejecución observado utilizando 20 núcleos y el tiempo teórico esperado bajo una paralelización ideal. Este tiempo ideal se estimó asumiendo un speedup lineal, es decir, dividiendo el tiempo secuencial entre el número de procesos. Los resultados muestran que el overhead es significativo y crece proporcionalmente con el tiempo total de ejecución para cada tamaño de matriz. Esto sugiere que los costos adicionales asociados a la comunicación, sincronización y distribución de datos escalan junto con la carga computacional, afectando directamente la eficiencia global del sistema distribuido.

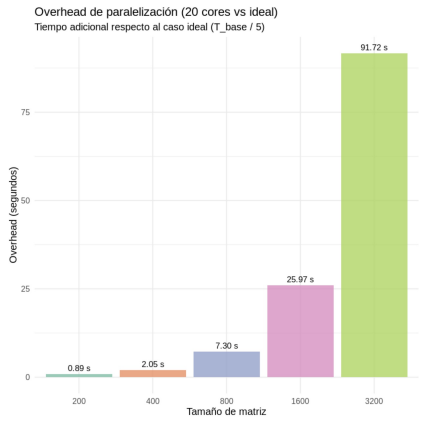

Figura 15. Gráfica indicadora de eficiencia de paralelización.

VI-D. *Limitaciones del experimento*

Es importante señalar ciertas limitaciones metodológicas del experimento:

- Únicamente se evaluaron dos configuraciones de procesos: 4 y 20.
- No se analizaron configuraciones intermedias (por ejemplo, 5, 8, 10 o 15 procesos), lo que impide observar con mayor precisión la escalación del sistema.

Finalmente, el punto de transición en el que la paralelización comienza a ser efectiva probablemente se encuentra entre los tamaños 1600×1600 y 3200×3200. Sin embargo, no se evaluaron dimensiones intermedias (por ejemplo, 2000×2000, 2400×2400 o 2800×2800), lo que deja un vacío experimental relevante e impide determinar con precisión el umbral de eficiencia.

VII. Conclusiones

- La distribución es efectiva únicamente cuando la carga computacional compensa el costo de comunicación.
  Los resultados experimentales demostraron que la ejecución distribuida mediante MPI no siempre implica una mejora inmediata en el rendimiento. En matrices pequeñas y medianas, el tiempo adicional generado por la comunicación entre nodos, la sincronización de procesos y la distribución de datos superó los beneficios obtenidos por la división del trabajo. Solo a partir de matrices de mayor tamaño se observó una mejora clara en el desempeño, confirmando que el paralelismo es especialmente útil en problemas con alta complejidad temporal como $O(n^3)$.
- La escalabilidad real del sistema se encuentra lejos del modelo ideal de speedup lineal.
  Teóricamente, al utilizar 20 procesos distribuidos en 5 nodos, se esperaría un speedup cercano a 20 bajo condiciones ideales. Sin embargo, los resultados mostraron que la eficiencia fue considerablemente menor al 100 %, lo que indica pérdidas de rendimiento. Estas pérdidas se deben principalmente a factores como la latencia de red y los costos asociados a las operaciones colectivas de MPI. Esto confirma que el número de núcleos no es el único determinante del rendimiento: la arquitectura de comunicación y la infraestructura física juegan un papel fundamental en la escalabilidad.
- El overhead crece proporcionalmente con el tamaño del problema y afecta directamente la eficiencia global.
  El cálculo permitió evidenciar que la diferencia entre el tiempo ideal y el tiempo real aumenta conforme crece el tamaño de la matriz. Esto indica que los costos de comunicación y sincronización no son constantes, sino que escalan junto con la cantidad de datos que deben transferirse y coordinarse entre los nodos. Aunque el tiempo total aumenta debido a la complejidad cúbica del algoritmo, también lo hace la cantidad de información intercambiada, lo que impacta directamente la eficiencia global del sistema distribuido.
- La implementación práctica del cluster permitió comprender tanto las ventajas como las limitaciones reales de MPI. Permitió comprender de forma experimental conceptos como comunicación entre procesos, coordinación distribuida, sincronización, latencia de red y escalabilidad. Más allá de los resultados numéricos, la experiencia evidenció que el diseño de sistemas distribuidos implica considerar tanto la capacidad de cómputo como los costos de comunicación y la arquitectura de red. En este sentido, el taller no solo permitió evaluar el desempeño de un algoritmo paralelo, sino también reforzar la comprensión de los principios teóricos enfrentándolos con limitaciones reales de implementación.

Referencias

[1] M. van Steen y A. S. Tanenbaum, "A brief introduction to distributed systems," *Computing*, vol. 98, n.º 10, págs. 967-1009, ago. de 2016, ISSN: 1436-5057. DOI: 10.1007/s00607-016-0508-7. en línea: http://dx.doi.org/10.1007/s00607-016-0508-7.

[2] P. Powell e I. Smalley, *¿Qué es la computación distribuida?* Nov. de 2025. en línea: https://www.ibm.com/es-es/think/topics/distributed-computing.

[3] C. Pham et al., "Failure Diagnosis for Distributed Systems Using Targeted Fault Injection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, n.º 2, págs. 503-516, 2017. DOI: 10.1109/TPDS.2016.2575829.

[4] M. Parashar, "Parallel and Distributed Systems," English, *COMPUTER*, vol. 53, n.º 11, págs. 7-8, nov. de 2020, ISSN: 0018-9162. DOI: 10.1109/MC.2020.3017320.

[5] C. J. Goodrum, C. P. Shields y D. J. Singer, "Understanding cascading failures through a vulnerability analysis of interdependent ship-centric distributed systems using networks," *Ocean Engineering*, vol. 150, págs. 36-47, feb. de 2018, ISSN: 0029-8018. DOI: 10.1016/j.oceaneng.2017.12.039. en línea: http://dx.doi.org/10.1016/j.oceaneng.2017.12.039.

[6] B. Barney, *Introduction to parallel computing tutorial*. en línea: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial.

[7] A. Grama, G. Karypis, V. Kumar y A. Gupta, *Introduction to parallel computing*, en, 2.ª ed. Boston, MA: Addison Wesley, feb. de 2003.

[8] D. Hyde, 1995. en línea: https://www.eg.bucknell.edu/~cs366/textbook-pdf/parallel98.pdf.

[9] J. Burkardt, *Distributed memory programming with MPI*, oct. de 2013. en línea: https://people.sc.fsu.edu/~jburkardt/workshops/mpi_2013_fsu/mpi_2013_fsu.pdf.

[10] G. Thiruvathukal, "Guest Editors' Introduction: Cluster Computing," *Computing in Science Engineering*, vol. 7, n.º 2, págs. 11-13, 2005. DOI: 10.1109/MCSE.2005.33.

[11] S. Mittal y P. Suri, "A Comparative Study of various Computing Processing Environments: A Review," *International Journal of Computer Science and Information Technologies (IJCSIT)*, vol. 3, págs. 5215-5218, ene. de 2012.

[12] D. A. Reed, "Beowulf Clusters: From Research Curiosity to Exascale," en *Proceedings of the 20 Years of Beowulf Workshop on Honor of Thomas Sterling's 65th Birthday*, ép. Beowulf '14, Annapolis, MD, USA: Association for Computing Machinery, 2014, págs. 28-33, ISBN: 9781450330312. DOI: 10.1145/2737909.2737913. en línea: https://doi.org/10.1145/2737909.2737913.

[13] T. Sterling, "[Front Matter]," en *Beowulf Cluster Computing with Linux*, The MIT Press, oct. de 2001, ISBN: 9780262286770. DOI: 10.7551/mitpress/1556.003.0030. eprint: https://direct.mit.edu/book/chapter-pdf/2320467/f010002_9780262286770.pdf. en línea: https://doi.org/10.7551/mitpress/1556.003.0030.

[14] M. Forum, nov. de 2023. en línea: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[15] W. Gropp, *T uto rial on MPI: The message-P assing interface Willi am Gropp A R G O N N E*. en línea: https://folk.idi.ntnu.no/elster/tdt4200-f09/gropp-mpi-tutorial.pdf.

[16] OpenMPI. en línea: https://www.open-mpi.org/.

[17] En línea: https://staff.fmi.uvt.ro/~dana.petcu/calcul/PC-2-RO.pdf.

[18] F. C. Team, *Performance metrics and scalability analysis: Parallel and Distributed Computing Class notes*, sep. de 2025. en línea: https://fiveable.me/parallel-and-distributed-computing/unit-6/performance-metrics-scalability-analysis/study-guide/MGfQFn4hMfU6dH2J.

[19] J. Verschelde, *Evaluating parallel performance 1 metrics time metrics and derived metrics 2 | Introduction to Supercomputing*, sep. de 2024. en línea: http://homepages.math.uic.edu/~jan/mcs572/evalperf.pdf.

[20] A. Grama, A. Gupta, G. Karypis y V. Kumar, *Analytical modeling of Parallel Systems*. en línea: https://www.cs.purdue.edu/homes/ayg/CS525_SPR17/chap5_slides.pdf.

[21] X. Wang y V. Roychowdhury, "Modelling and analysis of communication overhead for parallel matrix algorithms," *Mathematical and Computer Modelling*, vol. 32, n.º 3, págs. 349-379, 2000, ISSN: 0895-7177. DOI: https://doi.org/10.1016/S0895-7177(00)00140-0. en línea: https://www.sciencedirect.com/science/article/pii/S0895717700001400.