Entrega 3 - Sistemas Paralelos

Alumnos: Gaitan Catalina, Lizondo Santiago.

Las pruebas fueron realizadas sobre uno o mas nodos del cluster provisto por la cátedra, que tiene las siguientes características:

- 8GB de memoria RAM
- Dos procesadores Intel Xeon E5405

Características del procesador Intel Xeon E5405

- CPU class: Server
- Clock Speed: 2,0GHz
- Turbo Speed: Not Supported
- Numero de cores físicos: 4
- Numero de hilos por core: 4
- Cache: L1: 256KB, L2: 12.0MB, L3: No tiene
- Intel Hyper-Threading: Not Supported
- Velocidad del bus: 1333MHz

Enunciados

1. Resuelva los ejercicios 2 y 3 de la práctica 4.

   Ejercicio 2

   Los códigos blocking.c y non-blocking.c siguen el patrón master-worker, donde los procesos worker le envían un mensaje de texto al master empleando operaciones de comunicación bloqueantes y no bloqueantes, respectivamente.

   - Compile y ejecute ambos códigos usando P={4,8,16} (no importa que el número de núcleos sea menor que la cantidad de procesos). ¿Cuál de los dos retorna antes el control?
     - Las capturas de pantalla fueron sacadas de la ejecución del algoritmo con P=16 (dos nodos, cada uno con ocho task)
     - El código blocking.c: Utiliza las funciones Send y Recv (son bloqueantes). Se devuelve el control aproximadamente después de los 2 segundos de hacer el llamado.

       ```
       Tiempo transcurrido 0.000001 (s):   proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 1)
       Tiempo transcurrido 2.000092 (s):   proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 1
       ```

     - El codigo non-blocking.c: Utiliza las funciones ISend y IRecv (son no bloqueantes). Devuelve el control a los 0.000049 segundos.

       ```
       Tiempo transcurrido 0.000002 (s):   proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 1)
       Tiempo transcurrido 0.000049 (s):   proceso 0, MPI_IRecv() devolvio el control..
                                  ..pero el mensaje no fue aun recibido..
       ```

Ambas operaciones son de comunicación punto a punto, es decir que involucran a dos procesos en cada comunicación. Se puede ver que el algoritmo que retorna antes el control es el non-blocking.c. Esto se debe a que el proceso0 no espera a que el proceso1 reciba el mensaje para volver a tener el control de la ejecución. Sin embargo depende de la cláusula MPI_Wait() para asegurarse que la sincronización se realice con éxito, como consecuencia demandando un mayor costo de implementación.

Si analizamos el tiempo total de los dos algorimos:

Tiempo total: non-blocking

```
Tiempo transcurrido 30.000957 (s):  proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 15
```

Tiempo total: blocking

```
Tiempo transcurrido 30.010688 (s):  proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 15
```

Como se puede ver en las imágenes, los dos programas tardan en total prácticamente lo mismo. Pero las operaciones no bloqueantes, al devolver antes el control nos permiten aprovechar mejor el tiempo, pudiendo usar esos segundos en computo útil para el programa (siempre y cuando no se necesite el dato que se esta enviando).

- En el caso de la versión no bloqueante, ¿qué sucede si se elimina la operación MPI_Wait() (línea 52)? ¿Se imprimen correctamente los mensajes enviados? ¿Por qué?

  La operacion *MPI_Wait()* se utiliza para esperar la finalización de una operación de comunicación asincrónica en MPI. Al eliminarla de la linea 52 del código los procesos slave ejecutan el receive, pero al ser no bloqueante, estos no esperan a que el master realmente les mande el mensaje. En otras palabras, no se garantiza que las operaciones de comunicación hayan finalizado antes de que el programa continúe su ejecución.

  Como resultado obtenemos las siguientes salidas en ejecución en este programa en particular:

  > 📎 *Tiempo transcurrido 0.000063 (s): proceso 0, operación receive completa con mensaje: No debería estar leyendo esta frase.*

  Esto sucede porque se lee la variable message[BUFSIZ] (que esta inicializada con un “no debería estar leyendo esto”), antes que el proceso master le envíe el verdadero valor.

  Por lo tanto si se desea usar operaciones punto a punto no bloqueantes el programador debe prestar especial atención al realizar las comunicaciones y analizar cada caso en particular si se justifica la difícil implementación para el beneficio que se podría obtener.

  Conclusión:

  Las operaciones no bloqueantes nos aportan una mayor rapidez y un posible mejor aprovechamiento del tiempo para el computo del programa, pero se debe tener en cuenta el uso de operaciones extra que deben utilizarse, complejizando la programación y comprensión del código.

  Las operaciones bloqueantes nos ofrecen una seguridad y facilidad al programar, pero no nos permiten aprovechar el tiempo entre que se envían y se reciben los datos.

Ejercicio 3

Los códigos blocking-ring.c y non-blocking-ring.c comunican a los procesos en forma de anillo empleando operaciones bloqueantes y no bloqueantes, respectivamente. Compile y ejecute ambos códigos empleando P={4,8,16} (no importa que el número de núcleos sea menor que la cantidad de procesos) y N={10000000, 20000000, 40000000, …}. ¿Cuál de los dos algoritmos requiere menos tiempo de comunicación? ¿Por qué?.

Se han compilado y ejecutado ambos algoritmos (siendo N el tamaño de la matriz) obteniendo los siguientes resultados:

- Blocking ring:

P / N	10000000	20000000	40000000
4	0,249243	0.495548	0.982684
8	0,551430	1.099071	2.183957
16	2,341407	4.656825	9.292073

- Non-blocking ring:

P / N	10000000	20000000	40000000
4	0.220716	0.442580	0.886234
8	0.295976	0.595469	1.180805
16	0.962132	1.916866	3.960720

A partir de los resultados obtenidos concluimos lo siguiente:

Las operaciones no bloqueantes requieren menos espera de comunicación, ya que devuelve el control sin esperar a que el otro proceso reciba el mensaje, evitando quedarse bloqueado y se puede aprovechar este tiempo de cómputo, sin embargo como contra cara son mucho mas complejas de implementar.

2. Dada la siguiente expresión:

$$R = \frac{(MaxA \times MaxB - MinA \times MinB)}{PromA \times PromB} \times [A \times B] + [C \times Pot2(D)]$$

Desarrolle 3 algoritmos que computen la expresión dada:

- Algoritmo paralelo empleando MPI
- Algoritmo paralelo híbrido empleando MPI+OpenMP

Algoritmo paralelo empleando MPI

Esta solución utiliza la librería OpenMPI que utiliza el estándar MPI (Message Passing Interface).

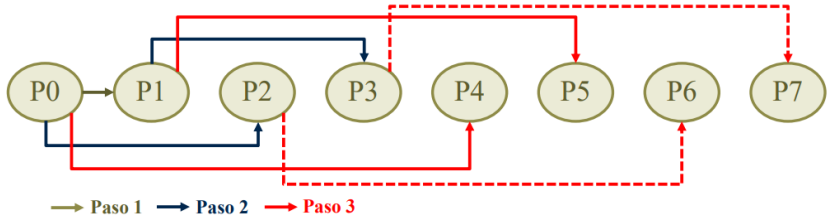
Se utilizan las comunicaciones y sincronizaciones colectivas para distribuir y recibir las matrices. Las operaciones usadas son:

- Scatter: se utiliza para enviar las matrices que se ordenan por filas. Esto se debe a que cada proceso debe tener una parte de la matriz para aplicarle las operaciones que le corresponda.

- Broadcast: se utiliza para enviar las matrices que se ordenan por columnas. Esto se debe a que los procesos necesitan la matriz entera, para poder acceder a todas sus columnas al realizar la multiplicación de matrices.
- AllReduce: se utiliza para poder recibir el máximo, mínimo y suma que encontró cada proceso de las matrices A y B, con el fin de obtener el valor correspondiente.
- Gather: se utiliza para que luego que todos los procesos hayan ejecutado las operaciones necesarios, el proceso coordinador obtenga todas las partes de la matriz resultante.
- Barrier: se utiliza para sincronizar los procesos para poder medir los tiempos correctamente.

La utilización de comunicaciones colectivas, ofrecen múltiples ventajas ante a las comunicaciones punto a punto:

1. Facilitan la programación y compresión del programa.
2. Son mas eficientes. MPI las optimiza aprovechando que en cada instante de tiempo varios procesos conocen el dato a enviar. En la siguiente imagen, se ve como en 3 instantes de tiempo, 7 procesos llegaron a obtener el dato:



Se puede notar que se ha decido que el proceso Coordinador realice la potencia de los elementos de la matriz D, y luego la matriz resultante (D2) a través de un Broadcast. La razón de esta elección es que si se eligiera que cada proceso calcule una porción se deberían realizar las siguientes comunicaciones:

1. El coordinador debe enviar a todos los procesos a través de un Scatter la matriz D inicializada con valores aleatorios del 1 al 40.
2. Todos los procesos deben recibir esta matriz, y calcular la potencia de los elementos.
3. Todos los procesos deben enviar su parte al Coordinador a través de un Gather.
4. Finalmente el coordinador debe hacer un broadcast de D2 (la matriz D, ya con todos sus elementos elevados al cuadrado).

   Una opción mas eficiente que reemplazaria las pasos 3 y 4, seria utilizar un AllGather. Este proporciona un mejor rendimiento, gracias a la optimización que realiza la librería OpenMPI.

   Como el uso de todas estas comunicaciones incrementarían el overhead, y como consecuencia el tiempo de comunicación, es mas pertinente permitir que un solo el proceso Coordinador realice esta operación.

Algoritmo paralelo híbrido empleando MPI+OpenMP

Esta solución híbrida combina la memoria compartida con el pasaje de mensajes aprovechando las potencialidades de cada uno. La comunicación entre hilos que pertenecen a la misma maquina física se realiza utilizando memoria compartida mientras que la comunicación entre ellas se lleva a cabo por medio de pasaje de mensajes.

El algoritmo implementa un enfoque de interacción master/worker, donde el master asume tanto el rol de coordinador como de trabajador. Se utilizan directivas de OMP (OpenMP) para controlar el paralelismo a nivel de hilo dentro de cada proceso, mientras que las comunicaciones de MPI se utilizan para sincronizar y compartir datos entre los distintos procesos.

De esta manera se aprovechan las directivas de OMP: Parallel, Parallel For, Reduction y Single, junto con las operaciones de comunicación colectiva de MPI: Scatter, Broadcast, Reduce, Gather y Barrier.

Al comienzo el coordinador reparte las distintas matrices a los distintos procesos por medio de las comunicaciones colectivas que mencionamos anteriormente. Debido a que todos los procesadores tienen una capacidad de cómputo igual y los bloques a procesar son del mismo tamaño, todos los procesos trabajaran a una velocidad similar. Por esta razón se les reparte de manera equitativa la cantidad de filas de las matrices que computa cada proceso.

Luego se utiliza la macro "*#pragma omp parallel*" para que cada proceso empiece su región paralelo, y que luego pueda realizar el computo con múltiples hilos.

Mediante la macro "*#pragma omp single*" de OpenMP se logra que cierta porción de código se ejecute únicamente una vez por proceso. Esto no significa que vaya a ejecutarse una única vez sino que se hará una vez por cada proceso. Esto resulta especialmente útil para que solo un proceso de cada nodo sea el que establezca la comunicación con los demás.

Finalmente, después de completar las operaciones en paralelo, los resultados se reúnen en el proceso coordinador mediante la comunicación colectiva MPI Gather.

Tiempos de ejecución

Todos los algoritmos se ejecutaran con las siguientes características:

- Optimización O3: debido a que fue la que dio resultados mas favorables en cuanto a performance y precisión matemática.
- Valor 8 como tamaño de bloque: a causa de que fue el valor que mejores tiempos de ejecución dio como resultado.
- Se ejecuta el algoritmo 3 veces y como resultado en la tabla se pone el promedio de los valores.

Algoritmo secuencial

Tiempos de ejecución:

N	512	1024	2048	4096
Tiempo ejecución	0,6302973	5,1153553	40,600667	321,0295176

Algoritmo paralelo con OpenMP

Tiempos de ejecución:

P / N	512	1024	2048	4096
2	0,310462	2,509158	19,915358	157,01270
4	0,158975	1,306469	10,298602	80,7673223
8	0,0850746	0,686168	5,388172	42,056858

Algoritmo paralelo con MPI

Tiempos de ejecución total:

P / N	512	1024	2048	4096
8	0,110689	0,857228	6,277374	46,710498
16	0,118568	0,750244	4,394564	28,268273
32	0,137377	0,583581	3,289972	18,433115

Tiempos de comunicación:

P / N	512	1024	2048	4096
8	0,034082	0,168792	0,801225	3,306287
16	0,088340	0,418695	1,684734	6,579821
32	0,137377	0,445532	1,960995	7,628592

Algoritmo paralelo Hibrido (MPI + OpenMP)

Tiempos de ejecución total:

P / N	512	1024	2048	4096
16	0,119208	0,703574	4,206656	27,362220
32	0,119221	0,658249	3,122089	17,729582

Tiempos de comunicación:

P / N	512	1024	2048	4096
16	0,077540	0,371899	1,567271	6,381743
32	0,100666	0,493513	1,806231	7,243038

Aclaración: en los tiempos de comunicación en el caso del algoritmo con MPI, al trabajar con una matriz de N=512 y P=32, existe un error de precisión al calcular los tiempos. Por consiguiente el tiempo de comunicación dio mayor al tiempo total. Esto podría deberse a una diferencia entre los tiempos de los clocks de los equipos. Para mantener la coherencia de los resultados se opto por escribir el tiempo de comunicación en ambos cuadros.

Calculo de SpeedUp, Eficiencia y Overhead de comunicación

El **SpeedUp** es una de las métricas mas populares. Refleja el beneficio de emplear procesamiento paralelo para resolver un problema dado comparado a realizar en forma secuencial.

Para calcular el SpeedUp se realiza la siguiente división:

$$S_p(n) = \frac{T_s(n)}{T_p(n)}$$

Nota: Para el calculo del tiempo secuencial se utilizo el algoritmo mas optimizado, que se ha explicado en entregas anteriores.

La **eficiencia** es la medida de la fracción de tiempo en la cual las unidades de procesamiento son empleadas en forma útil. Para calcularlo se realiza la siguiente división:

$$E_p(n) = \frac{S_p(n)}{S_{optimo}}$$

- Como la arquitectura del nodo del cluster donde estamos corriendo los algoritmos es homogénea, el SpeedUp Optimo va a ser la cantidad de unidades de procesamiento (P) que le asignemos al código paralelo.

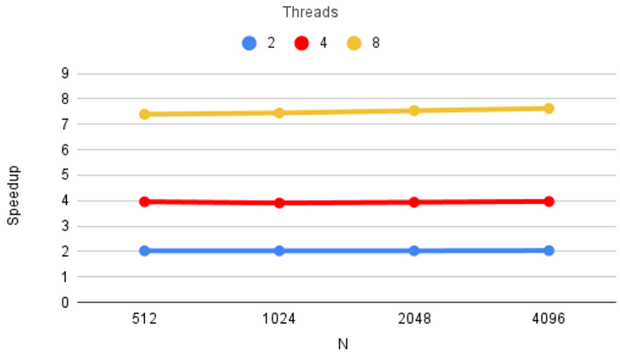
El **overhead de comunicación** la relación entre el tiempo requerido por las comunicaciones de nuestra solución y el tiempo total que esta requiera. Se calcula con la siguiente formula:

$$OC_p(n) = \frac{Tcomm_p(n)}{T_p(n)} \times 100$$

Algoritmo paralelo con OpenMP

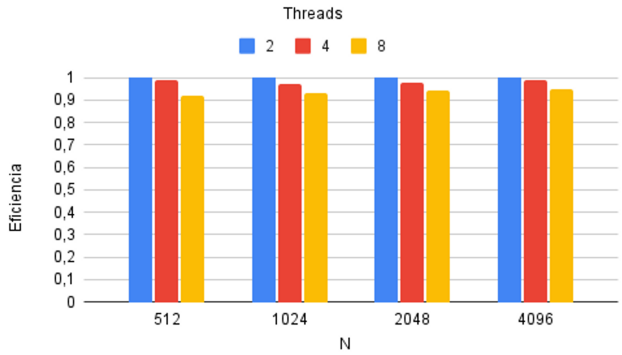
SpeedUp

	Order of matrix			
P	512	1024	2048	4096
2	2,03	2,03	2,03	2,04
4	3,96	3,91	3,94	3,97
8	7,4	7,45	7,54	7,63



Eficiencia

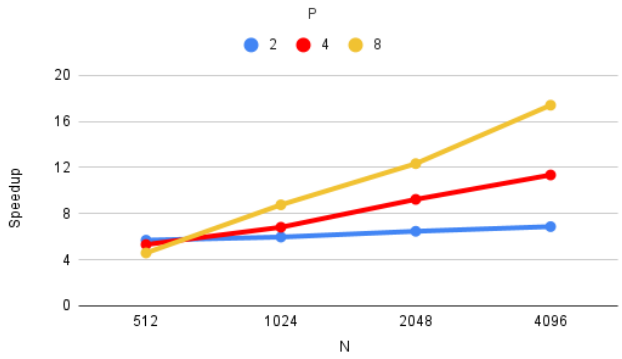
P	Order of matrix			
	512	1024	2048	4096
2	1,01	1,01	1,01	1,02
4	0,99	0,97	0,98	0,99
8	0,92	0,93	0,94	0,95



Algoritmo paralelo con MPI

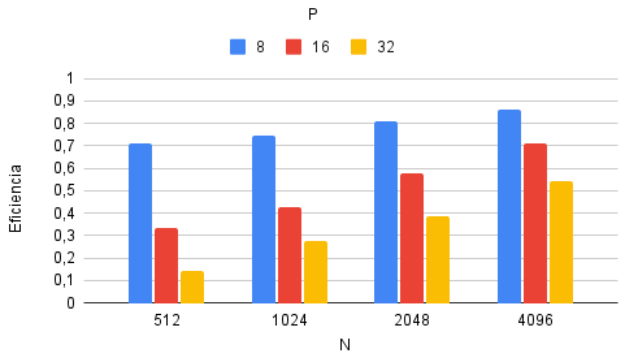
SpeedUp

P	Order of matrix			
	512	1024	2048	4096
8	5,6943	5,9673	6,4678	6,8727
16	5,3159	6,8183	9,2388	11,3565
32	4,5881	8,7655	12,3407	17,4159



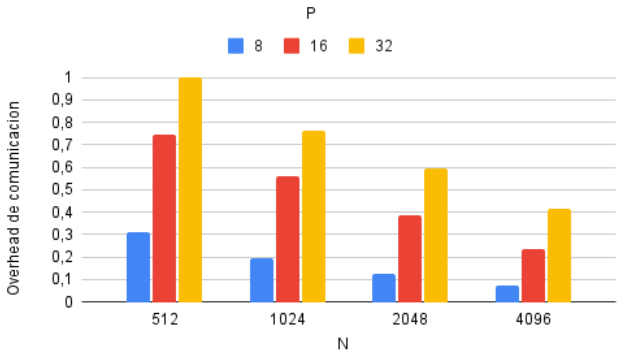
Eficiencia

P	Order of matrix			
	512	1024	2048	4096
8	0,71	0,75	0,81	0,86
16	0,33	0,43	0,58	0,71
32	0,14	0,27	0,39	0,54



Overhead por comunicación

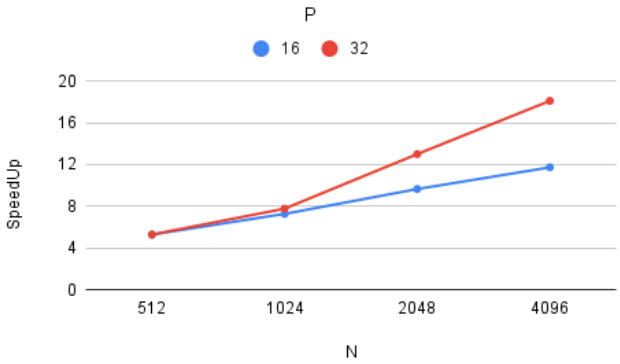
P	Order of matrix			
	512	1024	2048	4096
8	0,308	0,197	0,128	0,071
16	0,745	0,558	0,383	0,233
32	1,000	0,763	0,596	0,414



Algoritmo paralelo híbrido (MPI+OpenMP)

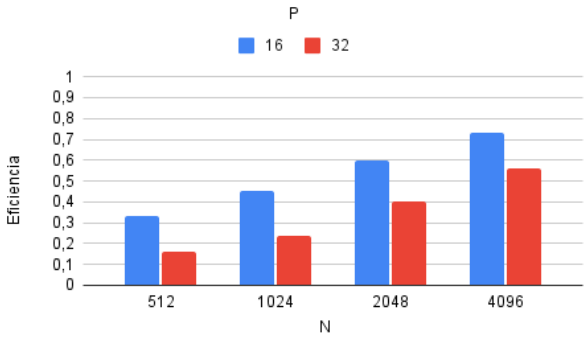
SpeedUp

P	Order of matrix			
	512	1024	2048	4096
16	5,2873	7,2705	9,6515	11,7325
32	5,2867	7,7711	13,0043	18,1069



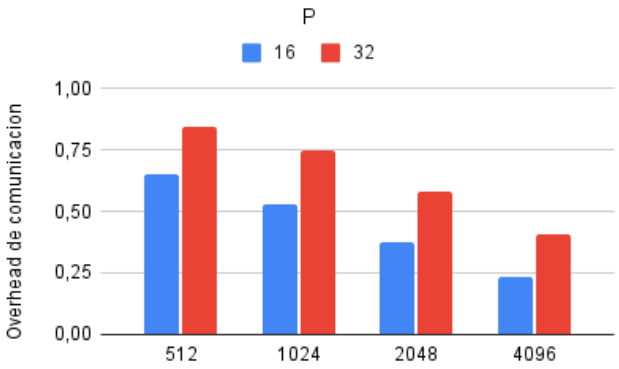
Eficiencia

P	Order of matrix			
	512	1024	2048	4096
16	0,33	0,45	0,6	0,73
32	0,16	0,24	0,4	0,56



Overhead por comunicación

	Order of matrix			
	512	1024	2048	4096
16	0,6504	0,5285	0,3725	0,2332
32	0,8443	0,7497	0,5785	0,4085



Análisis de los datos obtenidos

MPI vs OpenMP

En las tablas de los tiempos se observa que el algoritmo que utiliza OpenMP es mas rápido. Como consecuencia de esto, el SpeedUp y por último la eficiencia de este es mejor en comparación del algoritmo con MPI.

Esto ocurre debido a que MPI utiliza pasaje de mensajes, a diferencia de OpenMP que utiliza memoria compartida. Este sistema de comunicación utiliza un bus (en vez de acceder a la memoria directamente) en donde los procesos deben coordinarse para recibir y enviar la información. De esta manera se genera un overhead que en el caso de OpenMP no sucede.

MPI vs Híbrido (MPI+OpenMP)

Al analizar las tablas, se puede ver que los tiempos de comunicación son mayores en MPI. Esto sucede porque se tienen mas procesos que deben comunicarse a través del pasaje de mensajes. Por ejemplo para P=32, en MPI se tienen 32 procesos que deben comunicarse entre si usando pasaje de mensajes. En cambio en el híbrido, solamente se crean 4 procesos, que luego cada uno crea 8 threads, que utilizan memoria compartida, así disminuyendo el pasaje de mensajes.

Como consecuencia a que los tiempos del híbrido son mejores, el speedUp y la eficiencia de este también tienen resultados mas óptimos.

Puede suceder que no se vea tanta la diferencia entre los tiempos, debido a la carga de trabajo que tiene asignado en cada proceso. Adicionalmente en híbrido existe un overhead por la creación, sincronización y destrucción de hilos, ya que se tienen una cantidad de hilos considerable, pero un N bastante pequeño.

En los gráficos de eficiencia y overhead de comunicación tanto de MPI como del híbrido se puede ver que mas grande son las barras que indican el overhead, más se achica el de la eficiencia. Que esto suceda es muy coherente, ya que a mayor tiempo se gaste realizando las comunicaciones, menor tiempo se van a estar empleando las unidades de procesamiento para realizar computo.

Si vemos el lado izquierdo de los gráficos, se puede constatar que al incrementar el N, al tener que realizar mas computo, el overhead se decrementa y la eficiencia aumenta.

Rendimiento y escalabilidad

Rendimiento: esta característica de los programas paralelos se puede evaluar, analizando el SpeedUp y teniendo en cuenta las dos leyes:

- Ley de Amdahl: Permite estimar el Speedup alcanzable en aquellos programas paralelos que contienen partes secuenciales.
- Ley de Gustafson: Reescribió la ecuación que planteó Amdahl para estimar el SpeedUp máximo alcanzable, pero teniendo en cuenta el tamaño del problema (N).

Escalabilidad: esta característica de los programas paralelos hace referencia a la capacidad que tiene un sistema de mantener un nivel de eficiencia fijo al incrementar tanto el numero de unidades de procesamiento (P), como el tamaño del problema a resolver (N).

Algoritmo paralelo MPI

Rendimiento:

- Ley de Amdahl: en la tabla se puede ver que al incrementar el P, y mantener fijo el N, el **speedUp mejora** (excepto en valores con N chico, ya que se produce un gran overhead de comunicación). Esto es porque al utilizar una mayor cantidad de unidades de procesamiento la solución al problema se obtiene en un tiempo menor.

  Pero si siguiéramos experimentando incrementando el valor de P, podríamos ver que el SpeedUp se estancaría en un valor, sin importar cuantas unidades de procesamiento agreguemos. Esto sucede ya que el algoritmo esta condicionado por secciones secuenciales que presenta el código.
- Ley de Gustafson: al aumentar el tamaño del problema (N) y mantener el P también **mejora el speedUp.** Esto sucede porque la proporción de código que se ejecuta en paralelo, crece mayormente a la sección secuencial.

Escalabilidad:

- No es fuertemente escalable: debido a que si incrementamos el valor de P, y mantenemos fijo el valor de N: la eficiencia no se mantiene en un valor fijo.
- Para analizar si es débilmente escalable podemos mirar en diagonal los valores de la eficiencia. Si quisiéramos hacer una comparación equitativa entre MPI y el híbrido deberíamos mirar la tabla solo cuando P=16 y P=32. Mirando en estos valores podemos concluir que es débilmente escalable, ya que en una de las diagonales vemos que la eficiencia se mantiene entre el 58% y 54%.
- Se concluye entonces que el algoritmo realizado con MPI, es débilmente escalable.
  - Aunque si miráramos la tabla de manera aislada y teniendo en cuenta los 3 valores de P, no encontraríamos una eficiencia que se mantenga fija. Y podríamos decir que no es escalable.

Algoritmo Híbrido (MPI + OpenMP)

Rendimiento:

- Ley de Amdahl: en la tabla se puede ver que al incrementar el P, y mantener fijo el N, el **speedUp mejora**. Esto es porque al utilizar una mayor cantidad de unidades de procesamiento la solución al problema se obtiene en un tiempo menor.

  Pero igualmente como explicamos anteriormente con el algoritmo MPI, si siguiéramos incrementando P, el valor de SpeedUp en algún punto se estancaría debido a las secciones secuenciales del programa.

- Ley de Gustafson: al aumentar el tamaño del problema (N) y mantener el P también **mejora el speedUp.** Esto sucede porque la proporción de código que se ejecuta en paralelo, crece mayormente a la sección secuencial.

Escalabilidad:

- No es fuertemente escalable: debido a que si incrementamos el valor de P, y mantenemos fijo el valor de N, la eficiencia no se mantiene en un valor fijo.
- Si es débilmente escalable: ya que si incrementamos el valor de P y además incrementamos el N (como por ejemplo con “P=16 y N=1024” con “P=32 y N=2048”), se puede ver que la eficiencia se mantiene aproximadamente en un 40%.

Conclusión

A lo largo de este trabajo, se han explorado y comparado tres soluciones paralelas (MPI, OpenMP y MPI+OpenMP) para resolver la ecuación propuesta, considerando diferentes tamaños de matrices y valores de P. Estas soluciones se han contrastado a su vez con una solución secuencial.

Uno de los aspectos en común del algoritmo híbrido y el que utiliza MPI es el impacto del overhead de las comunicaciones. Cuando se trabaja con matrices pequeñas y un P elevado se observa una reducción considerable en el SpeedUp debido a este factor. En el caso de OpenMP esto sucede debido al costo de sincronización entre los hilos.

La solución híbrida que combina MPI y OpenMP resulta la más prometedora, ya que logra aprovechar las ventajas de ambos enfoques (memoria compartida junto con comunicación por mensajes) y muestra una mejor escalabilidad. Aunque en este estudio se utilizaron matrices de tamaño relativamente pequeño, es plausible suponer que la diferencia entre las soluciones sería aún más notable al trabajar con matrices de tamaño mayor: tales como 8192 y 16384. Sería pertinente investigar en futuros trabajos el desempeño de estas soluciones en matrices de mayor tamaño para validar esta hipótesis.

Por último, es importante destacar que la elección de una solución u otra depende de diversos factores. En casos donde se trabaje con matrices muy pequeñas y no sea necesario utilizar más de un nodo o equipo, la solución OpenMP sería la más adecuada si nos basamos en la eficiencia, ya que aprovecha mejor la arquitectura de memoria compartida. Por otro lado, si se trata de una matriz muy grande, si escalabilidad es un factor relevante y/o se necesiten utilizar varios nodos para lograr el poder de computo necesario, es probable que la opción híbrida sea la más conveniente.