Año 2024

Sistemas Paralelos

Trabajo Práctico N.º 3 –

Programación en pasaje de mensajes / Programación híbrida

Grupo 3
AGUSTINA SOL ROJAS – ANTONIO FELIX GLORIOSO CERETTI

Características de hardware y software

Cluster remoto: Cluster Multicore (partición Blade) conformado por 16 nodos. Cada nodo posee 8GB de RAM y 2 procesadores Intel Xeon E5405 de cuatro 4 cores cada uno que operan a 2.0GHz. Compilador gcc (Debian 8.3.0-6) 8.3.0

Enunciado

Punto N.º 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.

1. 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?
2. 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?

Respuesta

Importante: se hizo una mínima corrección en los códigos dados por la catedra (blocking.c y non-blocking.c) para que se calcule bien el tiempo total de ejecución.

1. Ejemplo de ejecución siendo P = 4

   Blocking

```
Tiempo transcurrido 0.000002 (s):       proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 1)
Tiempo transcurrido 2.000084 (s):       proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 1
Tiempo transcurrido 2.000098 (s):       proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 2)
Tiempo transcurrido 4.000125 (s):       proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 2
Tiempo transcurrido 4.000137 (s):       proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 3)
Tiempo transcurrido 6.000075 (s):       proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 3

Tiempo total = 6.000088 (s)
```

   Non-Blocking

```
Tiempo transcurrido 0.000002 (s):       proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 1)
Tiempo transcurrido 0.000047 (s):       proceso 0, MPI_IRecv() devolvio el control..
                                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 1.999980 (s):       proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 1
Tiempo transcurrido 1.999992 (s):       proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 2)
Tiempo transcurrido 2.000000 (s):       proceso 0, MPI_IRecv() devolvio el control..
                                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 4.000083 (s):       proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 2
Tiempo transcurrido 4.000097 (s):       proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 3)
Tiempo transcurrido 4.000105 (s):       proceso 0, MPI_IRecv() devolvio el control..
                                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 6.000090 (s):       proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 3

Tiempo total = 6.000103 (s)
```

Con todos los P solicitados siempre retorna antes el control el receive no bloqueante (`MPI_Irecv()` de *non_blocking.c*) ya que este tipo de receive lo retorna inmediatamente, pero no garantiza que la operación haya finalizado, haciendo que sea necesario el uso de `MPI_Wait()` a futuro. Por otro lado, el receive bloqueante (`MPI_Recv()` en *blocking*.c) retorna el control una vez que se haya copiado el mensaje recibido en el buffer, haciendo que tarde más en devolver el control.

2. Ejemplo de ejecución siendo P = 4

   Non-Blocking

```
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..
Tiempo transcurrido 0.000066 (s):       proceso 0, operacion receive completa con mensaje: No deberia estar leyendo esta frase.
Tiempo transcurrido 0.000079 (s):       proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 2)
Tiempo transcurrido 0.000091 (s):       proceso 0, MPI_IRecv() devolvio el control..
                                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 0.000104 (s):       proceso 0, operacion receive completa con mensaje: No deberia estar leyendo esta frase.
Tiempo transcurrido 0.000111 (s):       proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 3)
Tiempo transcurrido 0.000123 (s):       proceso 0, MPI_IRecv() devolvio el control..
                                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 0.000139 (s):       proceso 0, operacion receive completa con mensaje: No deberia estar leyendo esta frase.

Tiempo total = 0.000148 (s)
```

Si se elimina la operación `MPI_Wait()` no se imprimen correctamente los mensajes enviados, en su lugar se imprime "*No debería estar leyendo esta frase*". Esto se debe a que, al utilizar el receive no bloqueante, la función retorna el control de manera inmediata, sin esperar a que la operación de recepción finalice. Como el maestro no espera a que se complete la recepción antes de imprimir el mensaje, la información del buffer no está actualizada con el mensaje adecuado.

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?

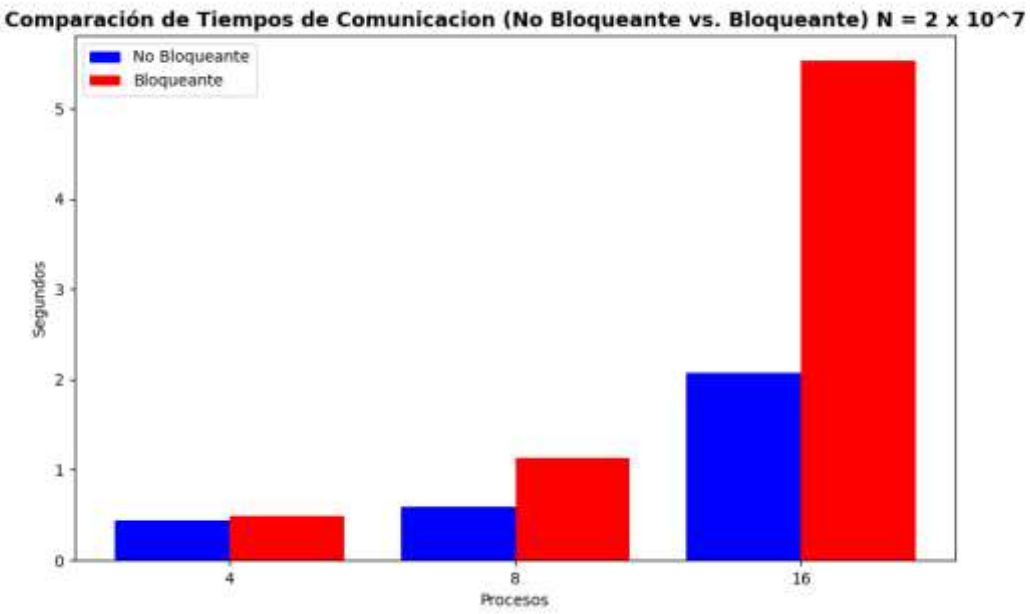
Respuesta

Blocking

N	*10000000*	*20000000*	*40000000*
Procesos			
4	0.250136	0.495287	0.986783
8	0.566437	1.130321	2.235916
16	2.777193	5.532839	11.028000

Non-blocking

N	*10000000*	*20000000*	*40000000*
Procesos			
4	0.216979	0.442921	0.881251
8	0.293724	0.592507	1.180763
16	1.030215	2.071593	4.172160



Siempre requiere menos tiempo de comunicación el algoritmo no bloqueante (*non-blocking-ring.c*) debido a que el mismo utiliza envíos y recepciones no bloqueantes (`MPI_Isend()` y `MPI_Irecv()`) permitiendo que los procesos comiencen un envío e inmediatamente pasen a realizar una recepción sin que el envío se realice completamente. Luego estos esperan a que estas comunicaciones finalicen efectivamente en un `MPI_Wait()`.

Por otra lado, en el algoritmo bloqueante (*blocking-ring.c*) los procesos primero realizan una recepción bloqueante (`MPI_Recv()`) y recién cuando se completa la operación realizan él envió (`MPI_Send()`). Solo es diferente el proceso P-1 que primero realiza él envió bloqueante y luego la recepción. Esto hace que los procesos deban esperar que se complete la operación de recepción (o envió) antes de pasar a la siguiente, generando que la comunicación tarde más.

Punto N.° 2

Dada la siguiente expresión:

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

- Donde A, B, C, D y R son matrices cuadradas de NxN con elementos de tipo double.
- MaxA, MinA y PromA son los valores máximo, mínimo y promedio de la matriz A, respectivamente.
- MaxB, MinB y PromB son los valores máximo, mínimo y promedio de la matriz B, respectivamente.

Desarrolle 2 algoritmos que computen la expresión dada:

1. Algoritmo paralelo empleando MPI

2. Algoritmo paralelo híbrido empleando MPI+OpenMP

Respuesta

Algoritmo y consideraciones de diseño

En todos los algoritmos se establece 64 como tamaño del bloque, ya que es el que retorna mejores resultados. Hay algunos casos donde este valor genera errores dado que se producen fallas de segmento al ingresar a posiciones de memoria que no corresponden a ese proceso. En estas situaciones se utilizan las siguientes fórmulas para establecer el tamaño de bloque:

(a) Algoritmo paralelo empleando MPI

$$bs = \frac{N}{P}$$

(b) Algoritmo paralelo hibrido empleando MPI+OpenMP

$$bs = \frac{N}{P * T}$$

- Siendo bs el tamaño de bloque, P la cantidad de procesos y T la cantidad de hilos.

Además, se utilizan las funciones provistas por MPI para la realización de las comunicaciones colectivas, ya que las mismas no solo facilitan la programación sino que también mejoran el rendimiento. Estas mismas son llamadas por todos los procesos.

Algoritmo secuencial

El algoritmo secuencial es el mismo empleado en la entrega anterior.

Algoritmo paralelo empleando MPI

El algoritmo paralelo que emplea MPI realiza lo siguiente:

1. Declaración e implementación de funciones y variables necesarias para el cómputo y el cálculo de tiempo de ejecución.
2. Se verifican los parámetros de entrada.
3. Se inicializa el entorno MPI con `MPI_Init()`, se obtiene la cantidad procesos con `MPI_Comm_size()` y se obtiene el rank del proceso dentro del comunicador global con `MPI_Comm_rank()`.
4. Se verifica si el tamaño de la matriz es válido.
5. Se realizan los cálculos para dividir el trabajo de los procesos.
6. Se establece el bs.
7. Se aloca memoria para las matrices. El coordinador aloca memoria para la matrices A, C, R y resultMatriz de forma completa, mientras que el resto de los procesos aloca solamente para la porción de datos que le corresponde. En cuanto a las matrices B y D, tanto el coordinador como el resto de los procesos reservan memoria para la matrices de forma completa.
8. El coordinador inicializa las matrices A, B, C, D con los valores 1.0; R y resultMatriz con los valores 0.0.
9. Se realiza una barrera para que los procesos inicien el trabajo al mismo tiempo.
10. El proceso coordinador envía los datos de las matrices (inicializadas por él) al resto de los procesos haciendo uso de `MPI_Scatter()` y `MPI_Bcast()`.
    a. Se utiliza `MPI_Scatter()` para enviar las matrices que se encuentran del lado izquierdo de la multiplicación. Esto se realiza de dicha manera ya que cada proceso va a trabajar sobre un conjunto de filas de esta, pero nunca sobre la matriz completa. Esto permite reducir la cantidad de datos a transferir, enviando solo aquellos que realmente van a ser utilizados por los procesos.
    b. Se utiliza `MPI_Bcast()` para enviar las matrices que se encuentran del lado derecho de la multiplicación. Esto se realiza de dicha manera ya que

todos los procesos trabajan sobre todos los datos dentro de esas matrices por la forma en la que se realiza la multiplicación de matrices.

11. Cada proceso realiza los cálculos locales de mínimos, máximos y sumas de la matriz A. Como a cada proceso se le envía una parte de la matriz, no debe calcular sobre qué porción trabaja, lo único que debe hacer es recorrer la matriz desde la primera posición hasta la última.
12. Cada proceso realiza los cálculos locales de mínimos, máximos y sumas de la matriz B. Como la matriz se pasa de forma completa, cada proceso debe calcular la porción sobre la cual debe trabajar, es decir, debe calcular la posición inicial y final.
13. Cada proceso realiza la multiplicación de las matrices A y B y de las matrices C y D para obtener los resultados de sus porciones correspondientes a las matrices resultMatriz y R.
14. Se utiliza la operación colectiva `MPI_Reduce()` para combinar los cálculos locales de mínimos, máximos y sumas de cada proceso, almacenando los mismos en el coordinador.
15. El coordinador realiza el cálculo del escalar.
16. El coordinador envía el escalar al resto de procesos a través de la función `MPI_Bcast()`.
17. Cada proceso suma los resultados de las multiplicaciones de sus submatrices y realiza la multiplicación por el escalar.
18. El proceso coordinador a través de la función `MPI_Gather()` recolecta las partes de la matriz R de cada proceso y las concatena en su matriz R.
19. Se realizan dos `MPI_Reduce()` necesarios para el cálculo del tiempo.
20. Se finaliza el entorno MPI con `MPI_Finalize()`.
21. El procesos coordinador termina de realizar los cálculos del tiempo y comunicación y los imprime en pantalla.
22. Se libera la memoria.

Algoritmo paralelo hibrido empleando MPI+ OpenMP

El algoritmo paralelo hibrido que emplea MPI+OpenMP parte de la solución anteriormente descripta, solo que en este caso se le agrega paralelismo de grano fino disponible a nivel de bucle haciendo uso de OpenMP.

A veces se utilizan las directivas “nowait” para evitar la barrera implícita al final del bucle for, cuando sea necesario, y “schedule(static) para distribuir las iteraciones entre los hilos, asignadolas en forma round-robin.

Para la programación hibrida se utiliza MPI_THREAD_FUNNELED(Nivel 1) en la cual los procesos pueden ser multihilados pero todas las comunicaciones las realizará el hilo master:

1. Luego del repartimiento inicial de matrices (véase punto 10 de “Algoritmo paralelo empleando MPI”) se especifica la región paralela con la directiva “#pragma omp parallel” , indicando el número de hilos (recibido por parámetro) y que las variables “i, j, k, offsetI, offsetJ, posA, posB” son privadas para cada hilo. Las variables no indicadas son por defecto compartidas.
2. Se realizan los cálculos de mínimos, máximos y sumas de las matriz A distribuyendo los mismos en varios hilos de un proceso utilizando la directiva “#pragma omp for” con la cláusula “reduction” para evitar interferencias y resultados incorrectos. Se utiliza “nowait“ y “schedule(static)”.
3. Se realizan los cálculos de mínimos, máximos y sumas de las matriz B distribuyendo los mismos en varios hilos de un proceso utilizando la directiva “#pragma omp for” con la cláusula “reduction”. En este caso no se utiliza “nowait” ya que es necesario que todos los hilos de los procesos hallan calculado los mínimos, máximos y sumas locales para el futuro calculo correcto del escalar.
4. Se realiza la multiplicación de las matrices A con B y C con D en bloques distribuyendo la misma en varios hilos de un proceso utilizando la directiva “#pragma omp for”. Se utiliza “nowait” y “schedule(static)”.
5. El hilo master de los procesos hace uso de la operación colectiva `MPI_Reduce()` para combinar los cálculos locales de mínimos, máximos y sumas de cada proceso, almacenando los mismos en el coordinador.
6. El hilo master del coordinador realiza el cálculo del escalar.

7. El hilo master del coordinador envía el escalar a los hilos master del resto de procesos a través de la función `MPI_Bcast()`.
8. Se realiza una barrera para que los hilos de un proceso puedan seguir con su ejecución una vez que el hilo master haya enviado/recibido el escalar para evitar inconsistencias en los cálculos.
9. Se suma el resultado de las multiplicaciones de matrices y se multiplica por el escalar utilizando nuevamente la directiva “#pragma omp for”. Se usan las directivas “nowait” y “schedule(static)”
10. Finaliza la región paralela.
11. Se continua de igual manera que el algoritmo paralelo con MPI a partir del punto 18.

Análisis del algoritmo

Tabla con los tiempos de ejecución obtenidos en el cluster remoto al compilar el algoritmo paralelo que utiliza MPI "matricesCalculoMPI.c" con nivel de optimización O3

N	*512*	*1024*	*2048*	*4096*
Procesos				
8	0.087715	0.551936	4.140267	32.068369
16	0.108042	0.557391	3.039733	20.028511
32	0.148887	0.549346	2.466385	14.833311

Tabla con los tiempos de comunicación obtenidos en el cluster remoto al compilar el algoritmo paralelo que utiliza MPI "matricesCalculoMPI.c" con nivel de optimización O3

N	*512*	*1024*	*2048*	*4096*
Procesos				
8	0.031423	0.086097	0.367934	1.486572
16	0.098959	0.345060	1.196975	4.947848
32	0.231195	0.692132	1.697522	8.925775

Tabla con los tiempos de ejecución obtenidos en el cluster remoto al compilar el algoritmo paralelo hibrido que utiliza MPI+OpenMP "matricesCalculoHibrido.c" con nivel de optimización O3

N	*512*	*1024*	*2048*	*4096*
Procesos				
16	0.104401	0.518092	2.987598	19.449267
32	0 .110975	0.485166	2.344343	13.307388

Tabla con los tiempos de comunicación obtenidos en el cluster remoto al compilar el algoritmo paralelo hibrido que utiliza MPI+OpenMP "matricesCalculoHibrido.c" con nivel de optimización O3

N	*512*	*1024*	*2048*	*4096*
Procesos				
16	0.072457	0.284830	1.132428	4.459937
32	0.097646	0.365172	1.443600	6.117768

Comparación de rendimiento del algoritmo MPI con híbrido

Gráfico con los tiempos de ejecución obtenidos en el cluster remoto al compilar el algoritmo MPI y el algoritmo MPI+OpenMP con nivel de optimización O3 y P=16

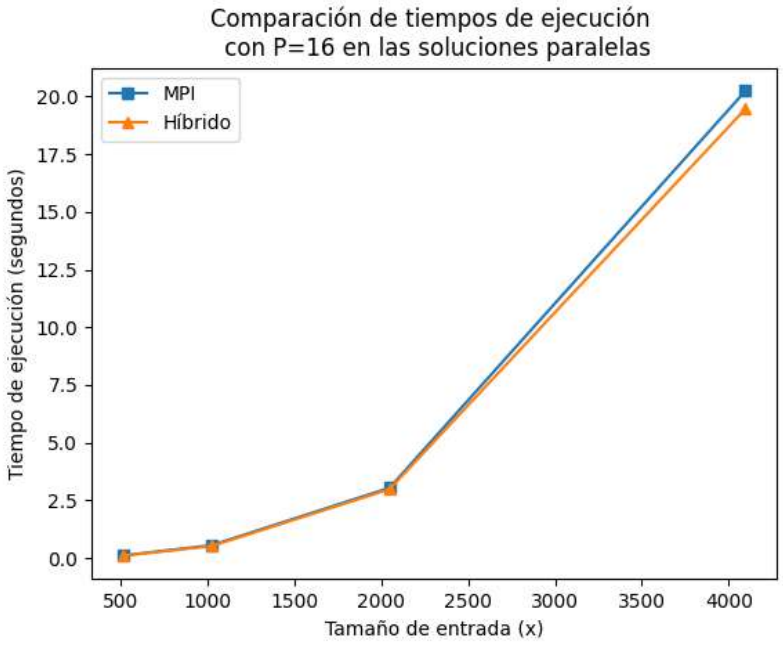
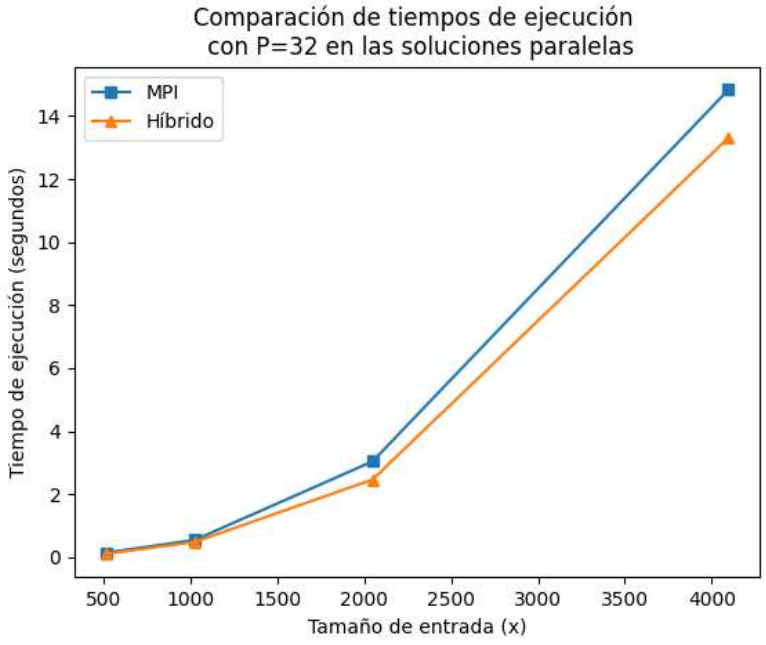


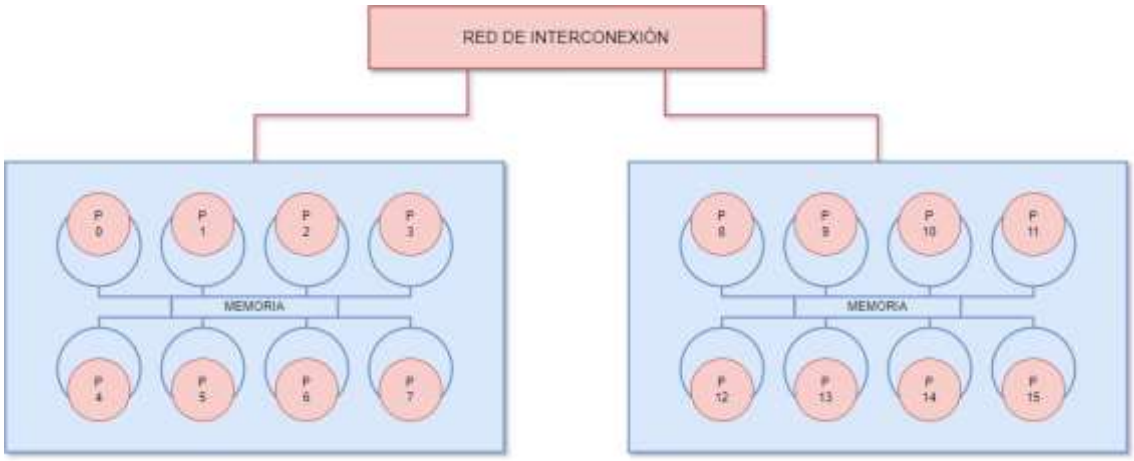
Gráfico con los tiempos de comunicación obtenidos en el cluster remoto al compilar el algoritmo MPI y el algoritmo MPI+OpenMP con nivel de optimización O3 y P=32



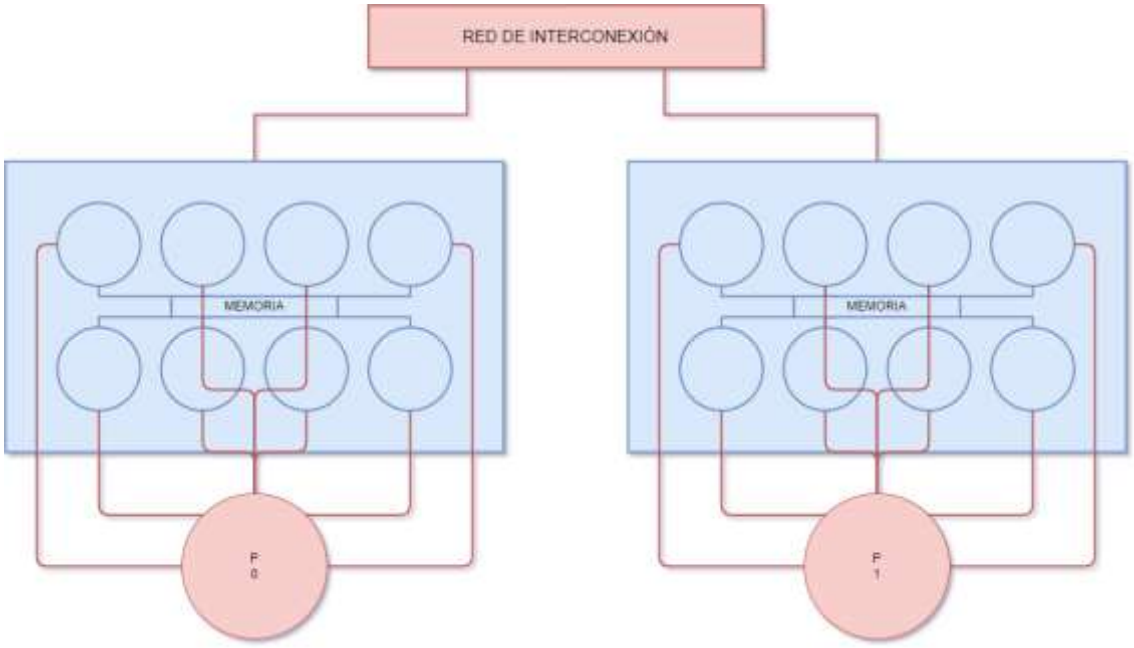
Se puede notar en los gráficos y tablas que la solución hibrida (MPI+OpenMP) es más rápida en cuanto al tiempo de ejecución. Esto ocurre ya que no se tiene el overhead de comunicación que existe en MPI. Se ahorra el overhead de pasaje de mensajes entre procesos que están dentro de un mismo nodo, debido a que en la solución hibrida estos serán hilos que se comunicarán (y sincronizarán) haciendo uso de la memoria compartida (menor latencia).

En el siguiente diagrama se puede ver una comparación entre la cantidad de procesos que se tienen en ambas soluciones con P=16

Solución MPI



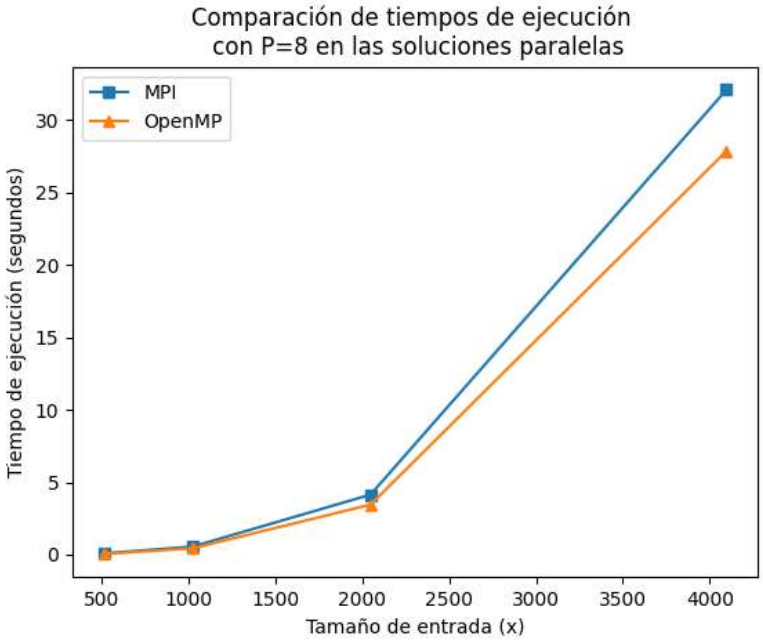
Solución hibrida



En el caso de MPI se tendrán 16 procesos que se comunicarán a través de la red de interconexión, mientras que en el caso de la solución hibrida se tendrán solamente 2, reduciendo así el overhead asociado a la misma (menos procesos, menos mensajes).

Comparación de rendimiento del algoritmo MPI con OpenMP

Gráfico con los tiempos de ejecución obtenidos en el cluster remoto al compilar el algoritmo MPI y el algoritmo OpenMP con nivel de optimización O3 y P=8



La solución con OpenMP tiene mejor tiempo de ejecución debido a lo explicado anteriormente. En OpenMP se tendrán 8 hilos que se comunicaran a través de la memoria compartida, mientras que en MPI se tendrán 8 procesos que intercambiaran mensajes a través de la red de interconexión. Esto hace que el tiempo de ejecución se vea afectado debido al pasaje de mensajes y la latencia asociada con la red de interconexión.

Speedup y Eficiencia

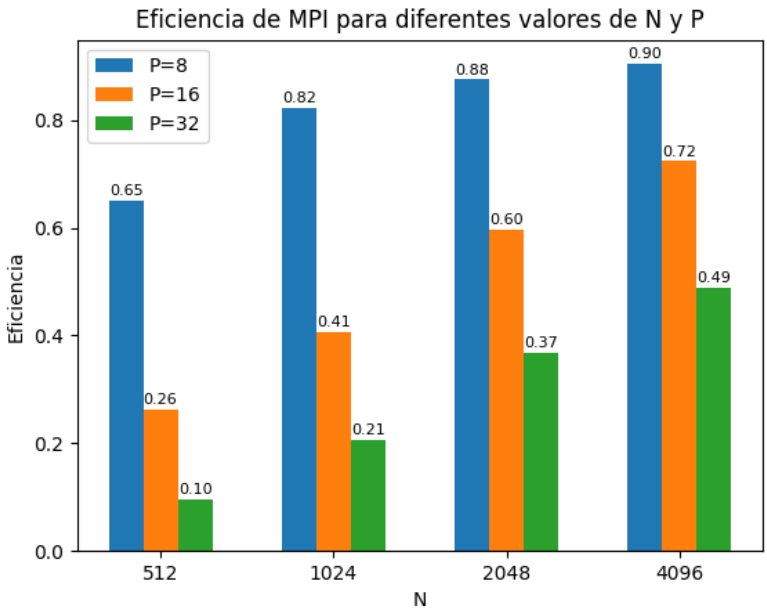
SPEEDUP – MPI

N	*512*	*1024*	*2048*	*4096*
Procesos				
8	5.195200365	6.579132363	7.001016118	7.233046277
16	4.21777642	6.514744587	9.535730934	11.58109043
32	3.060690322	6.61015098	11.7524539	15.63723682

EFICIENCIA – MPI

N	*512*	*1024*	*2048*	*4096*
Procesos				
8	0.6494000456	0.8223915454	0.8751270147	0.9041307846
16	0.2636110262	0.4071715367	0.5959831834	0.7238181519
32	0.09564657256	0.2065672181	0.3672641844	0.4886636506

Gráfico con la eficiencia del algoritmo MPI



SPEEDUP – MPI+OPENMP

N	*512*	*1024*	*2048*	*4096*
Procesos				
16	4.364871984	7.008909615	9.702133955	11.92600199
32	4.106303221	7.484572291	12.3642641	17.43031743

EFICIENCIA – MPI+OPENMP

N	*512*	*1024*	*2048*	*4096*
Procesos				
16	0.272804499	0.4380568509	0.6063833722	0.7453751244
32	0.1283219757	0.2338928841	0.3863832531	0.5446974197

Gráfico con la eficiencia del algoritmo MPI+OpenMP

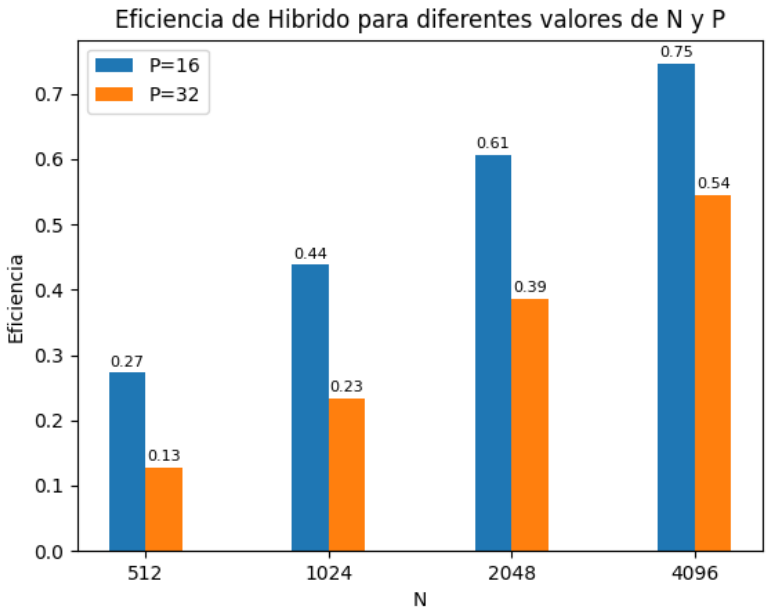
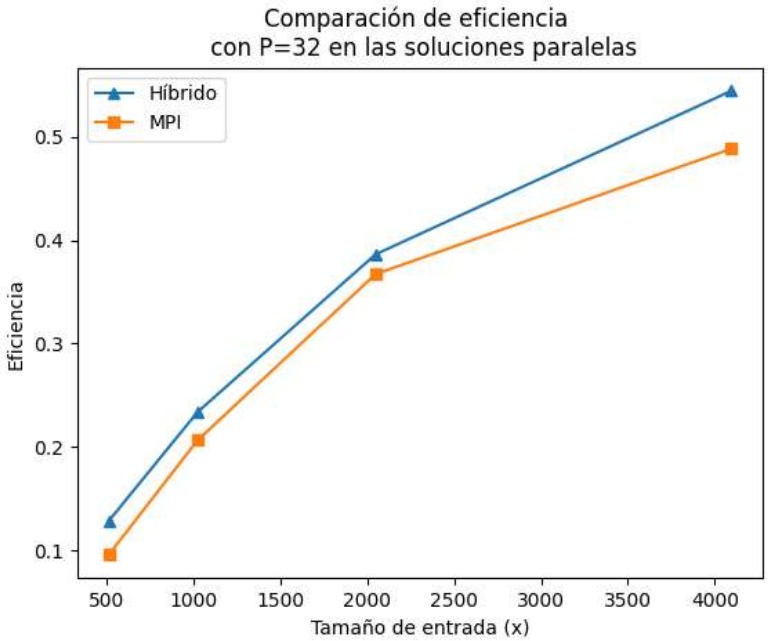


Gráfico con la eficiencia obtenida en los algoritmos MPI y MPI+OpenMP



Se puede notar en los gráficos lo siguiente:

- La eficiencia aumenta mientras se incrementa la entrada.

    - Como N es mas grande, los procesos pasaran mas tiempo haciendo uso de la CPU (porque se tienen mas datos por proceso) que comunicándose, lo que hará que se aproveche más el uso de dicho recurso.
- La eficiencia se reduce cuando se incrementa la cantidad de procesos.
    - Se tendrá un mayor número de mensajes y cada proceso trabajará sobre una menor cantidad de datos (definición de variable stripSize en el código) haciendo que no aprovechen completamente el tiempo de CPU.
- La eficiencia de la solución hibrida es mayor a la de MPI.
    - Esto sucede debido a que se tienen menos procesos comunicándose a través de la red de interconexión (explicado en la comparación de rendimiento)

Conclusiones Generales

A lo largo del trabajo se hace notar que la solución hibrida es mejor respecto a la solución MPI en cuando a tiempo de ejecución y eficiencia. Como se explico varias veces, esto está relacionado al overhead de comunicación asociado con el pasaje de mensajes en MPI, el cual se reduce en el caso del algoritmo hibrido.

También se notó que mientras más aumenta N mayor será la eficiencia, pero si se incrementa la cantidad de procesos, esta decae. Para evitar esto se deberá asignar una proporción adecuada de procesos para un N determinado. Un ejemplo de esto sería el caso de la solución hibrida donde P=16 parece ser la mejor opción, pero va a llegar un punto donde se tendrá un N tan grande que esa cantidad de procesos no bastará para tener una solución con tiempo de ejecución eficiente.