Sistemas Paralelos

Trabajo Práctico 3

Programación

en pasaje de mensajes -

Programación híbrida

Grupo 13

AÑO 2023 – 1° Semestre

Alumnos

- Garofalo, Pedro 17136/5
- Morena, Nahuel 16290/1

Características del equipo hogareño utilizado para las siguientes consignas:

- Procesador: AMD Ryzen 9 5950X (16 núcleos, 32 hilos)
- Memoria: 64Gb DDR4 3.600MHz
- Sistema Operativo: Arch Linux (Linux 6.2.10)
- Compilador: GCC 12.2.1

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

1.2. 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?**

  Salida de *blocking.c* (con P = 4):

```
Tiempo transcurrido 0.000002 (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
Tiempo transcurrido 2.000105 (s):    proceso 0, llamando a
MPI_Recv() [bloqueante] (fuente rank 2)
Tiempo transcurrido 4.000075 (s):    proceso 0, MPI_Recv() devolvio
control con mensaje: Hola Mundo! Soy el proceso 2
Tiempo transcurrido 4.000087 (s):    proceso 0, llamando a
MPI_Recv() [bloqueante] (fuente rank 3)
Tiempo transcurrido 6.000065 (s):    proceso 0, MPI_Recv() devolvio
control con mensaje: Hola Mundo! Soy el proceso 3

Tiempo total = 0.000000 (s)
```

  Salida de *non-blocking.c* (con P = 4):

```
Tiempo transcurrido 0.000001 (s):    proceso 0, llamando a MPI_IRecv()
[no bloqueante] (fuente rank 1)
Tiempo transcurrido 0.000039 (s):    proceso 0, MPI_IRecv() devolvio
el control..
                                     ..pero el mensaje no fue aun
recibido..
Tiempo transcurrido 2.000088 (s):    proceso 0, operacion receive
completa con mensaje: Hola Mundo! Soy el proceso 1
Tiempo transcurrido 2.000100 (s):    proceso 0, llamando a MPI_IRecv()
[no bloqueante] (fuente rank 2)
Tiempo transcurrido 2.000108 (s):    proceso 0, MPI_IRecv() devolvio
el control..
                                     ..pero el mensaje no fue aun
recibido..
Tiempo transcurrido 3.999990 (s):    proceso 0, operacion receive
completa con mensaje: Hola Mundo! Soy el proceso 2
Tiempo transcurrido 4.000003 (s):    proceso 0, llamando a MPI_IRecv()
[no bloqueante] (fuente rank 3)
Tiempo transcurrido 4.000011 (s):    proceso 0, MPI_IRecv() devolvio
el control..
                                     ..pero el mensaje no fue aun
recibido..
Tiempo transcurrido 5.999987 (s):    proceso 0, operacion receive
completa con mensaje: Hola Mundo! Soy el proceso 3

Tiempo total = 0.000000 (s)
```

En las salidas mostradas anteriormente podemos ver la diferencia entre *MPI_Recv()* y *MPI_Irecv()*: el primero no retorna el control hasta que no se haya recibido y leído el mensaje solicitado completamente al buffer. En cambio, el segundo retorna el control inmediatamente, requiriendo más adelante llamar a *MPI_Wait()* sobre la request hecha con *MPI_Irecv()* para esperar a que se termine de leer el mensaje.

Se puede observar lo mismo para todos los P solicitados.

- **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?**

  Salida de *non-blocking.c* comentando la línea 52 (con P = 4):

```
Tiempo transcurrido 0.000001 (s):     proceso 0, llamando a
MPI_IRecv() [no bloqueante] (fuente rank 1)
Tiempo transcurrido 0.000038 (s):     proceso 0, MPI_IRecv()
devolvio el control..
                                      ..pero el mensaje no fue aun
recibido..
Tiempo transcurrido 0.000061 (s):     proceso 0, operacion receive
completa con mensaje: No deberia estar leyendo esta frase.
Tiempo transcurrido 0.000071 (s):     proceso 0, llamando a
MPI_IRecv() [no bloqueante] (fuente rank 2)
Tiempo transcurrido 0.000079 (s):     proceso 0, MPI_IRecv()
devolvio el control..
                                      ..pero el mensaje no fue aun
recibido..
Tiempo transcurrido 0.000092 (s):     proceso 0, operacion receive
completa con mensaje: No deberia estar leyendo esta frase.
Tiempo transcurrido 0.000098 (s):     proceso 0, llamando a
MPI_IRecv() [no bloqueante] (fuente rank 3)
Tiempo transcurrido 0.000108 (s):     proceso 0, MPI_IRecv()
devolvio el control..
                                      ..pero el mensaje no fue aun
recibido..
Tiempo transcurrido 0.000121 (s):     proceso 0, operacion receive
completa con mensaje: No deberia estar leyendo esta frase.

Tiempo total = 0.000000 (s)
```

No, en vez de imprimir los mensajes enviados por los procesos se puede leer la frase “No debería estar leyendo esta frase”, que es el valor inicial asignado al buffer. Esto ocurre porque al no esperar la llegada del mensaje con *MPI_Wait()*, el proceso *MASTER* imprime inmediatamente el contenido del buffer que no se actualizó con el mensaje esperado.

Se puede observar lo mismo para todos los P solicitados.

1.3. 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?

	P = 4	P = 8	P = 16
N = 10.000.000	0,25	0,55	2,34
N = 20.000.000	0,49	1,09	4,66
N = 40.000.000	0,99	2,17	9,27

Tiempo de comunicación en segundos de *blocking-ring.c* en clúster remoto.

	P = 4	P = 8	P = 16
N = 10.000.000	0,22	0,30	1,02
N = 20.000.000	0,44	0,61	1,78
N = 40.000.000	0,88	1,18	4,00

Tiempo de comunicación en segundos de *non-blocking-ring.c* en clúster remoto.

El algoritmo no bloqueante (*non-blocking-ring.c*) requiere menos tiempo de comunicación. Observando el código podemos ver que la diferencia está en el uso de *MPI_Recv()* y *MPI_Send()* (bloqueantes) contra *MPI_Irecv()* y *MPI_Isend()* (no bloqueantes) para enviar los vectores. En *blocking-ring.c*, todos los procesos menos el último (P-1) primero realizan un *MPI_Recv()* y luego un *MPI_Send()*, al ser *MPI_Recv()* bloqueante, estos procesos no pueden comenzar a enviar sus vectores *sendbuff* hasta que no recibieron completamente el vector *recvbuff*. En cambio, *non-blocking-ring.c* al utilizar versiones no bloqueantes para el envío y recepción de mensajes no tiene este problema, puede enviar y recibir ambos vectores en simultáneo y luego esperar a que ambas operaciones terminen.

2. Dada la siguiente expresión:

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

- **Donde A, B, C y R son matrices cuadradas de NxN con elementos de tipo double.**
- **D es una matriz de enteros de NxN y debe ser inicializada con elementos de ese tipo (NO float ni double) en un rango de 1 a 40.**
- **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.**
- **La función Pot2(D) aplica potencia de 2 a cada elemento de la matriz D.**

Desarrolle 2 algoritmos que computen la expresión dada:

1. **Algoritmo paralelo empleando MPI.**
2. **Algoritmo paralelo híbrido empleando MPI + OpenMP.**

Los resultados deben validarse comparando la salida del algoritmo secuencial con la salida del algoritmo paralelo. Posiblemente deban tener en cuenta algún grado de error debido a la precisión en el cálculo.

Mida el tiempo de ejecución de los algoritmos en el clúster remoto. Las pruebas deben considerar la variación del tamaño de problema (N = { 512, 1024, 2048, 4096 }) y, en el caso de los algoritmos paralelos, también la cantidad de núcleos (P = { 8, 16, 32 }) para MPI, es decir, 1, 2 y 4 nodos, respectivamente; P = { 16, 32 } para híbrido. es decir, 2 y 4 nodos, respectivamente).

Para resolver los algoritmos solicitados se partió de la base creada en la entrega anterior: reutilizamos la librería '*matlib.h*', la cual contiene procedimientos comunes a todos los algoritmos sobre vectores y matrices. Tuvimos que adaptarla para que inicialice, cargue y almacene las matrices correctamente teniendo en cuenta que se pueden ejecutar varias instancias del programa debido a MPI. Continuamos utilizando '*matrixtool.py*' y '*runtests.sh*' (con mínimas adaptaciones necesarias) para probar la validez de los algoritmos.

Tanto la librería *'matlib.h'* y los algoritmos solicitados: *'mat_mpi.c'* para el algoritmo MPI y *'mat_hybrid.c'* para el algoritmo híbrido, se encuentran en el directorio *'src'*.

Para la implementación del algoritmo MPI, decidimos utilizar *MPI_Bcast()* y *MPI_Scatter()* para que el proceso coordinador, el cual inicializa las matrices, envíe los datos al resto de los procesos. Utilizamos *MPI_Scatter()* para enviar los datos de las matrices que se encontraban del lado izquierdo de la multiplicación, ya que dividimos el trabajo por filas y los procesos sólo acceden al rango de filas asignado a cada uno; de esta forma reducimos la cantidad de datos a transferir. Para las matrices del lado derecho de la multiplicación, utilizamos *MPI_Bcast()* (para la matriz B) y *MPI_Allgather()* (para la matriz resultante de la operación Pot2(D)).

Luego, para implementar el algoritmo híbrido, simplemente partimos de la solución anterior y agregamos paralelismo de grado fino a nivel de bucle con OpenMP.

Continuamos, como en las entregas anteriores, utilizando multiplicación de matrices por bloque para maximizar el rendimiento del programa. Decidimos usar tamaño de submatrices de 64x64 elementos ya que en las anteriores entregas pudimos comprobar que daba los mejores resultados comparado con los otros tamaños probados. En algunos casos de prueba, tuvimos que utilizar tamaños de submatrices más pequeños cuando las matrices eran pequeñas y la cantidad de tareas era alta, ya que la cantidad de elementos que debía procesar cada tarea era menor al tamaño de bloque pedido, y esto hacía que el programa funcione incorrectamente.

Resultados obtenidos

Tiempos (MPI):

	N = 512	N = 1024	N = 2048	N = 4096
P = 8	0,09	0,58	4,31	33,90
P = 16	0,12	0,56	3,10	20,63
P = 32	0,12	0,49	2,48	13,96

Speedup (MPI):

	N = 512	N = 1024	N = 2048	N = 4096
P = 8	5,80	6,83	7,28	7,40
P = 16	4,20	7,06	10,15	12,15
P = 32	4,26	8,08	12,69	17,96

Eficiencia (MPI):

	N = 512	N = 1024	N = 2048	N = 4096
P = 8	0,73	0,85	0,91	0,92
P = 16	0,26	0,44	0,63	0,76
P = 32	0,13	0,25	0,40	0,56

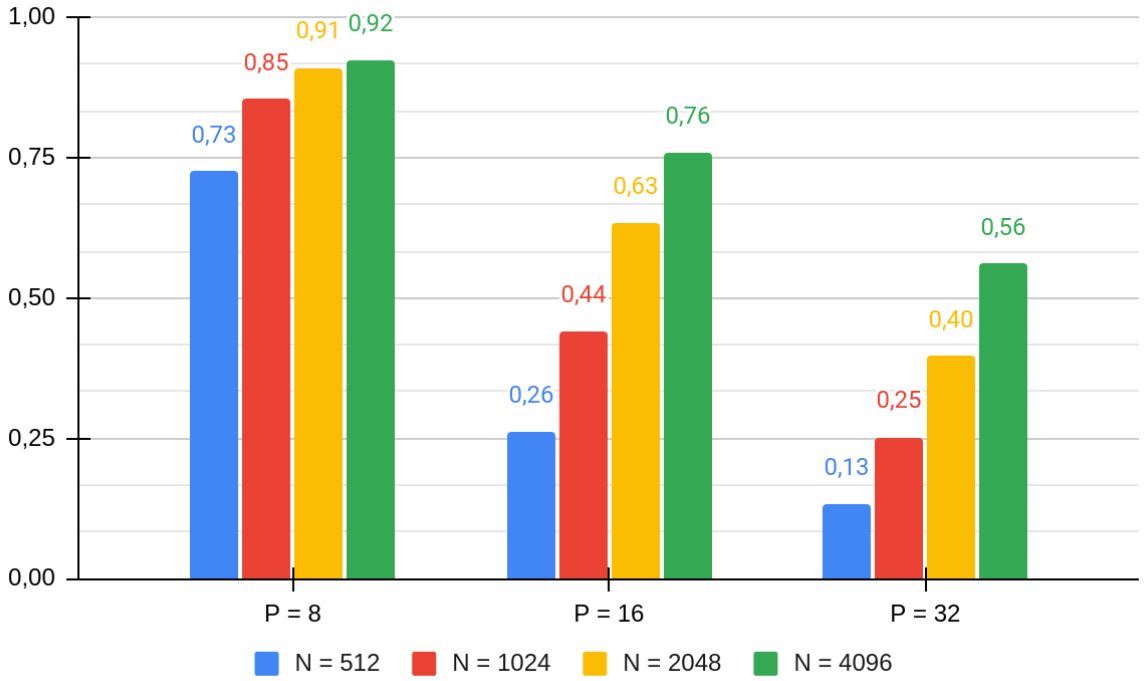


Gráfico de la tabla anterior (eficiencia del algoritmo MPI).

Tiempos (híbrido):

	N = 512	N = 1024	N = 2048	N = 4096
P = 16	0,10	0,51	2,99	19,76
P = 32	0,11	0,47	2,35	13,54

Speedup (híbrido):

	N = 512	N = 1024	N = 2048	N = 4096
P = 16	4,86	7,70	10,49	12,69
P = 32	4,53	8,33	13,39	18,51

Eficiencia (híbrido):

	N = 512	N = 1024	N = 2048	N = 4096
P = 16	0,30	0,48	0,66	0,79
P = 32	0,14	0,26	0,42	0,58

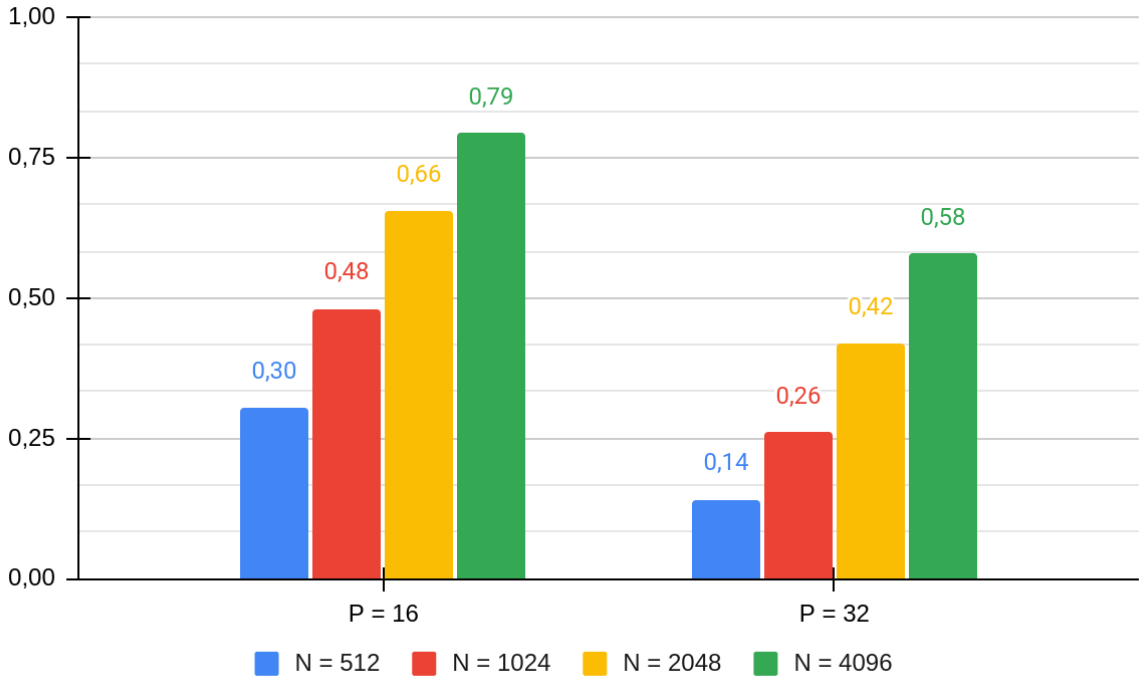
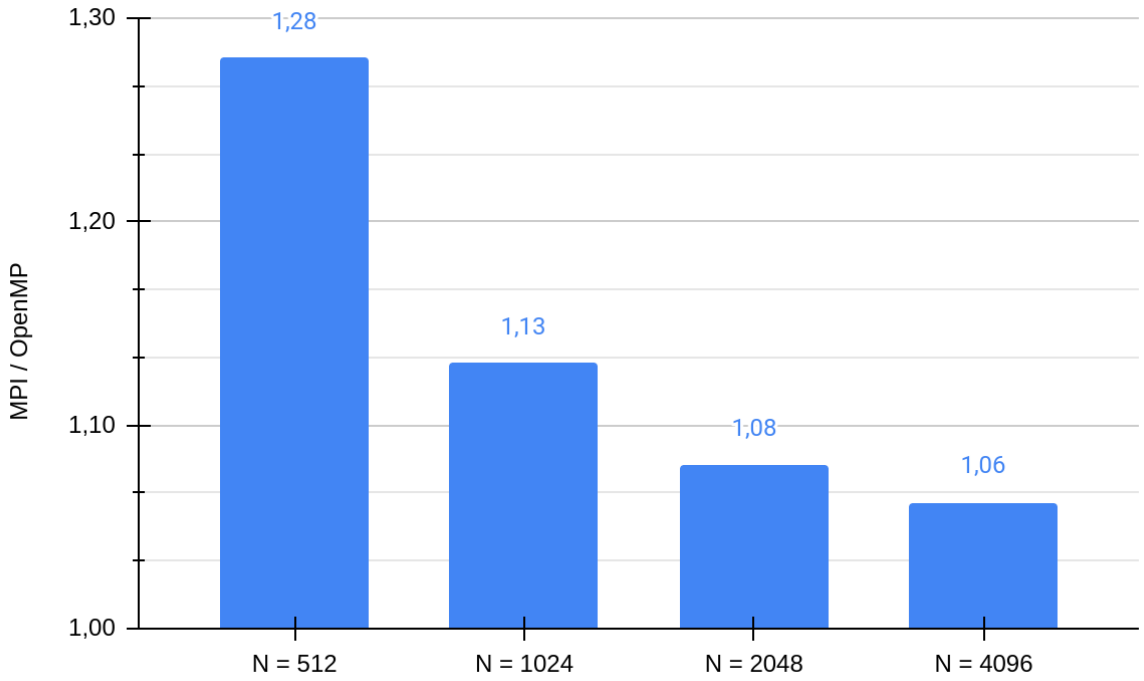


Gráfico de la tabla anterior (eficiencia del algoritmo híbrido).

Comparación de tiempos del algoritmo MPI y OpenMP:

	N = 512	N = 1024	N = 2048	N = 4096
OpenMP	0,07	0,51	3,99	31,92
MPI	0,09	0,58	4,31	33,90

Tiempos en segundos obtenidos para el algoritmo OpenMP y MPI para P = 8.



Factor obtenido de la división de los tiempos del algoritmo MPI sobre los tiempos del algoritmo OpenMP.

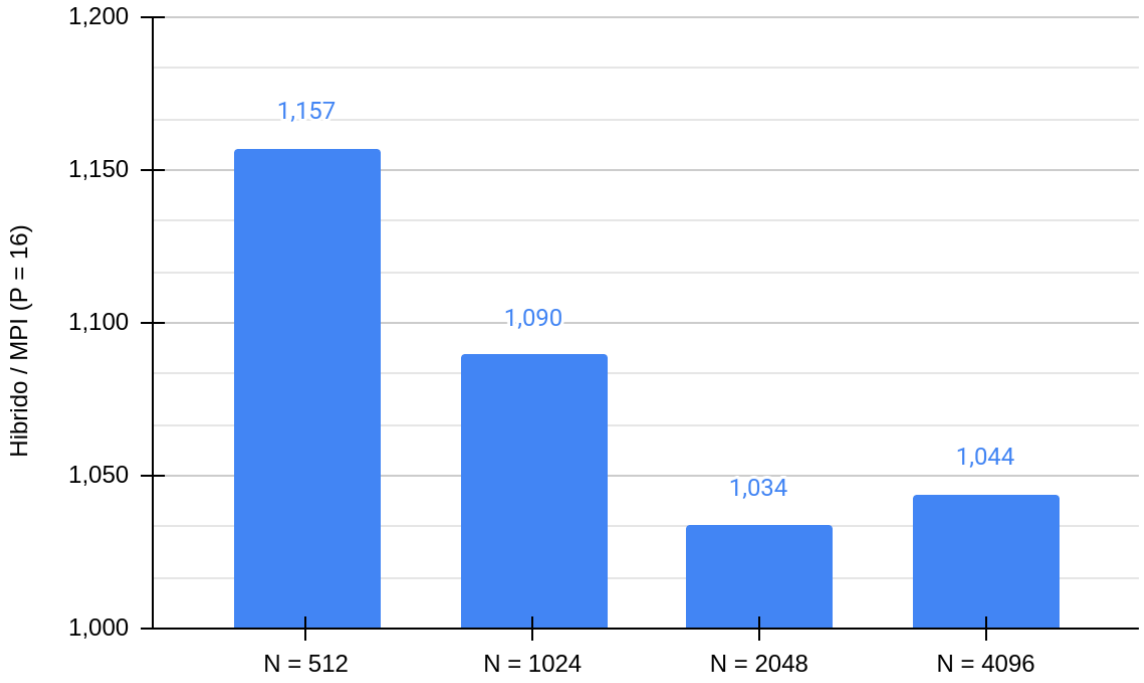
Comparación del algoritmo MPI e hibrido:

	N = 512	N = 1024	N = 2048	N = 4096
MPI	0,12	0,56	3,10	20,63
Híbrido	0,10	0,51	2,99	19,76

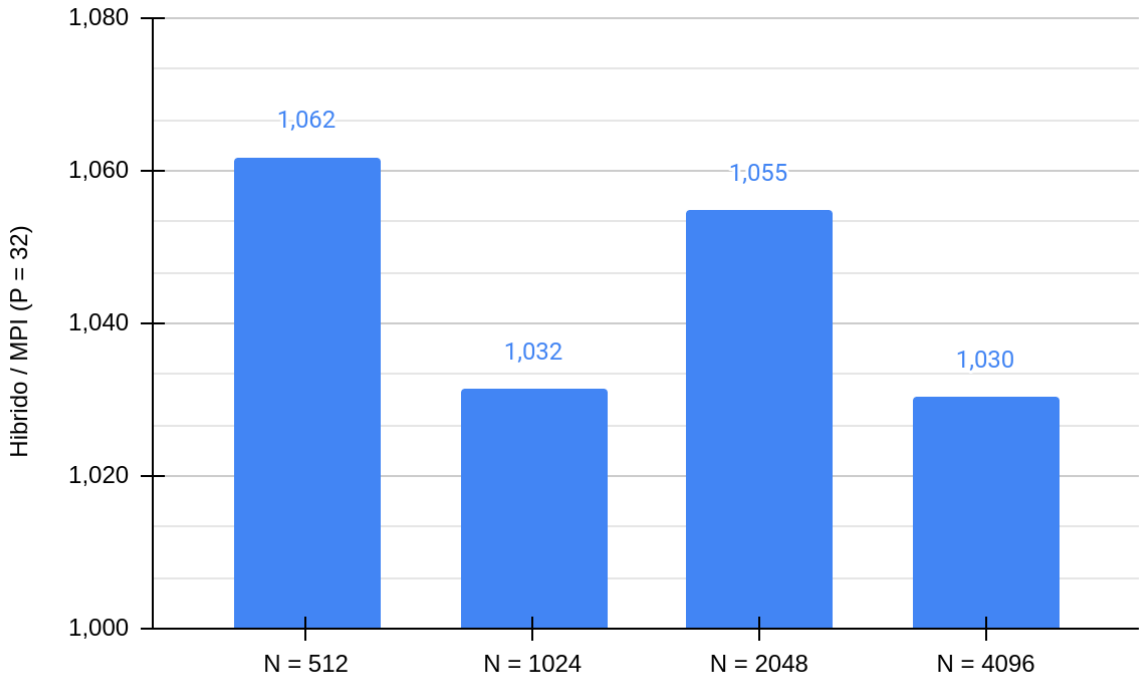
Tiempo en segundos de los algoritmos MPI e híbrido para P = 16.

	N = 512	N = 1024	N = 2048	N = 4096
MPI	0,12	0,49	2,48	13,96
Híbrido	0,11	0,47	2,35	13,54

Tiempo en segundos de los algoritmos MPI e híbrido para P = 32.



Factor obtenido de la división de los tiempos del algoritmo híbrido sobre los tiempos del algoritmo MPI (para P = 16)



. Factor obtenido de la división de los tiempos del algoritmo híbrido sobre los tiempos del algoritmo MPI (para P = 32)

En las tablas y gráficos anteriores podemos observar que, para ambos algoritmos, la eficiencia disminuye cuando la cantidad de nodos utilizados aumenta. Esto muestra que el algoritmo tiene un overhead de comunicación bastante alto. También podemos ver que el algoritmo híbrido es más rápido para todos los tamaños de matrices y cantidad de procesos solicitados, esto ocurre porque nos ahorramos el overhead que trae el pasaje de mensajes dentro de un mismo nodo.