Sistemas Paralelos - TP 3 - Programación en Pasaje de Mensajes / Programación híbrida

Autores

- Juan Cruz Cassera Botta, 17072/7
- Lautaro Josin Saller, 18696/9

Especificaciones del sistema utilizado

Cluster (usaremos el Multicore Blade)

- Nodos: 16.
- Cada nodo posee 8GB de RAM y 2 procesadores Intel Xeon E5405 de cuatro núcleos cada uno a 2.0GHz.
- Caché L1 DATA: 32K.
- Caché L1 Instruction: 32K.
- Caché L2: 6144K.
- Sistema Operativo: Debian 4.19.0-5-amd64 #1 SMP Debian 4.19.37-5+deb10u2 (2019-08-08) x86_64.

Parte 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?**
- **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?**

Compilación y ejecución

Para compilar usamos:

- **Para blocking.c**: mpicc -o blocking blocking.c
- **Para non-blocking.c**: mpicc -o non-blocking non-blocking.c

Para ejecutar los programas compilados en el cluster, usamos el script ***ejercicio2.sh***:

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --exclusive
#SBATCH --tasks-per-node=4
#SBATCH -o ./output_ejecutable_P_4.txt
#SBATCH -e ./errors_ejecutable_P_4.txt
mpirun ejecutable
```

donde:

- El `-N` es 1 porque usamos un único nodo.
- El `tasks-per-node` (P) lo vamos modificando entre 4, 8 y 16.
- `blocking` luego se intercambia por non-blocking en el script para su ejecución.

- Cuando `tasks-per-node` = 16:
    - Usamos mpirun con argumento `–oversubscribe`.
    - Agregamos la sentencia `#SBATCH --overcommit` al script

Resultados de la ejecución con P = 4

blocking	non-blocking
Tiempo transcurrido 0.000001 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 1) Tiempo transcurrido 2.000148 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 1 Tiempo transcurrido 2.000160 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 2) Tiempo transcurrido 4.000069 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 2 Tiempo transcurrido 4.000081 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 3) Tiempo transcurrido 6.000038 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 3 Tiempo total = 0.000000 (s)	Tiempo transcurrido 0.000002 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 1) Tiempo transcurrido 0.000044 (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 2.000101 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 2) Tiempo transcurrido 2.000109 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 4.000187 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 2 Tiempo transcurrido 4.000200 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 3) Tiempo transcurrido 4.000207 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 5.999991 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 3 Tiempo total = 0.000000 (s)

Resultados de la ejecución con P = 8

blocking	non-blocking
Tiempo transcurrido 0.000002 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 1) Tiempo transcurrido 2.000110 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 1 Tiempo transcurrido 2.000124 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 2) Tiempo transcurrido 3.999957 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 2 Tiempo transcurrido 3.999970 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 3) Tiempo transcurrido 6.000041 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 3 Tiempo transcurrido 6.000055 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 4) Tiempo transcurrido 8.000096 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 4 Tiempo transcurrido 8.000111 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 5) Tiempo transcurrido 10.000089 (s): proceso 0, MPI_Recv() devolvio control	Tiempo transcurrido 0.000002 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 1) Tiempo transcurrido 0.000054 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 2.000261 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 1 Tiempo transcurrido 2.000273 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 2) Tiempo transcurrido 2.000281 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 4.000055 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 2 Tiempo transcurrido 4.000067 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 3) Tiempo transcurrido 4.000075 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido..

con mensaje: Hola Mundo! Soy el proceso 5 Tiempo transcurrido 10.000108 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 6) Tiempo transcurrido 12.000126 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 6 Tiempo transcurrido 12.000141 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 7) Tiempo transcurrido 14.000123 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 7 Tiempo total = 0.000000 (s)	Tiempo transcurrido 6.000139 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 3 Tiempo transcurrido 6.000151 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 4) Tiempo transcurrido 6.000160 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 8.000079 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 4 Tiempo transcurrido 8.000093 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 5) Tiempo transcurrido 8.000100 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 10.000188 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 5 Tiempo transcurrido 10.000206 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 6) Tiempo transcurrido 10.000215 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 12.000164 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 6 Tiempo transcurrido 12.000177 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 7) Tiempo transcurrido 12.000186 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 14.000143 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 7 Tiempo total = 0.000000 (s)

Resultados de la ejecución con P = 16

blocking	non-blocking
Tiempo transcurrido 0.000003 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 1) Tiempo transcurrido 2.000124 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 1 Tiempo transcurrido 2.000139 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 2) Tiempo transcurrido 4.000161 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 2 Tiempo transcurrido 4.000174 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 3) Tiempo transcurrido 6.000013 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 3 Tiempo transcurrido 6.000026 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 4) Tiempo transcurrido 7.999883 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 4 Tiempo transcurrido 7.999896 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 5) Tiempo transcurrido 9.999937 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 5 Tiempo transcurrido 9.999952 (s): proceso 0, llamando a MPI_Recv()	Tiempo transcurrido 0.000001 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 1) Tiempo transcurrido 0.000044 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 2.000035 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 1 Tiempo transcurrido 2.000048 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 2) Tiempo transcurrido 2.000056 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 3.999876 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 2 Tiempo transcurrido 3.999889 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 3) Tiempo transcurrido 3.999897 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 5.999871 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 3

[bloqueante] (fuente rank 6)
Tiempo transcurrido 12.000086 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 6
Tiempo transcurrido 12.000104 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 7)
Tiempo transcurrido 14.000066 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 7
Tiempo transcurrido 14.000086 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 8)
Tiempo transcurrido 16.011840 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 8
Tiempo transcurrido 16.011854 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 9)
Tiempo transcurrido 18.000235 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 9
Tiempo transcurrido 18.000249 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 10)
Tiempo transcurrido 20.000265 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 10
Tiempo transcurrido 20.000280 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 11)
Tiempo transcurrido 22.000124 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 11
Tiempo transcurrido 22.000138 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 12)
Tiempo transcurrido 24.000010 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 12
Tiempo transcurrido 24.000024 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 13)
Tiempo transcurrido 26.000083 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 13
Tiempo transcurrido 26.000104 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 14)
Tiempo transcurrido 28.000182 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 14
Tiempo transcurrido 28.000197 (s): proceso 0, llamando a MPI_Recv() [bloqueante] (fuente rank 15)
Tiempo transcurrido 30.000312 (s): proceso 0, MPI_Recv() devolvio control con mensaje: Hola Mundo! Soy el proceso 15

Tiempo total = 0.000000 (s)

Tiempo transcurrido 5.999884 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 4)
Tiempo transcurrido 5.999892 (s): proceso 0, MPI_IRecv() devolvio el control..
..pero el mensaje no fue aun recibido..
Tiempo transcurrido 7.999992 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 4
Tiempo transcurrido 8.000010 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 5)
Tiempo transcurrido 8.000020 (s): proceso 0, MPI_IRecv() devolvio el control..
..pero el mensaje no fue aun recibido..
Tiempo transcurrido 9.999983 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 5
Tiempo transcurrido 9.999996 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 6)
Tiempo transcurrido 10.000007 (s): proceso 0, MPI_IRecv() devolvio el control..
..pero el mensaje no fue aun recibido..
Tiempo transcurrido 11.999964 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 6
Tiempo transcurrido 11.999979 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 7)
Tiempo transcurrido 11.999987 (s): proceso 0, MPI_IRecv() devolvio el control..
..pero el mensaje no fue aun recibido..
Tiempo transcurrido 13.999932 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 7
Tiempo transcurrido 13.999952 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 8)
Tiempo transcurrido 13.999967 (s): proceso 0, MPI_IRecv() devolvio el control..
..pero el mensaje no fue aun recibido..
Tiempo transcurrido 16.011403 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 8
Tiempo transcurrido 16.011419 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 9)
Tiempo transcurrido 16.011428 (s): proceso 0, MPI_IRecv() devolvio el control..
..pero el mensaje no fue aun recibido..
Tiempo transcurrido 17.999845 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 9
Tiempo transcurrido 17.999858 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 10)
Tiempo transcurrido 17.999867 (s): proceso 0, MPI_IRecv() devolvio el control..
..pero el mensaje no fue aun recibido..
Tiempo transcurrido 20.000125 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 10
Tiempo transcurrido 20.000139 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 11)
Tiempo transcurrido 20.000147 (s): proceso 0, MPI_IRecv() devolvio el control..
..pero el mensaje no fue aun recibido..
Tiempo transcurrido 21.999993 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 11
Tiempo transcurrido 22.000016 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 12)
Tiempo transcurrido 22.000031 (s): proceso 0, MPI_IRecv() devolvio el control..
..pero el mensaje no fue aun recibido..
Tiempo transcurrido 24.000048 (s): proceso 0, operacion receive completa

	con mensaje: Hola Mundo! Soy el proceso 12 Tiempo transcurrido 24.000061 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 13) Tiempo transcurrido 24.000074 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 26.000058 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 13 Tiempo transcurrido 26.000076 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 14) Tiempo transcurrido 26.000085 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 28.000057 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 14 Tiempo transcurrido 28.000071 (s): proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 15) Tiempo transcurrido 28.000080 (s): proceso 0, MPI_IRecv() devolvio el control.. ..pero el mensaje no fue aun recibido.. Tiempo transcurrido 30.000017 (s): proceso 0, operacion receive completa con mensaje: Hola Mundo! Soy el proceso 15 Tiempo total = 0.000000 (s)

¿Cuál de los dos retorna antes el control?

El código que retorna antes el control es ***non-blocking***. Esto es así debido a que non-blocking usa operaciones de comunicación **no bloqueantes** como son `MPI_Irecv()` y `MPI_Isend()`. Estas operaciones comienzan la comunicación e inmediatamente devuelven el control, sin garantizar que la operación haya terminado. Esto se podría aprovechar para realizar cómputo útil mientras finalizan las operaciones de comunicación. De esta forma, se pueden aprovechar los tiempos muertos (aunque esto no se hace en el código dado) de:

- Esperar a que el receptor reciba el mensaje que le envié.
- Esperar a recibir un mensaje de otro proceso.
- **En ambos casos, en vez de esperar sin hacer nada, hago otras cosas hasta que el envío o la recepción hayan terminado.**

¿Qué sucede si se elimina la operación `MPI_Wait()` (línea 52)? ¿Se imprimen correctamente los mensajes enviados? ¿Por qué?

Si en non-blocking no se utiliza la función `MPI_Wait()` el proceso master no esperará a que finalicen los requests asociados a las operaciones Irecv. Con lo cual podría finalizar su ejecución antes de que terminen las operaciones de comunicación.

Los mensajes enviados no se imprimen correctamente debido a que:

- El proceso master recibe un mensaje con `MPI_Irecv()`.
- Como es una operación no bloqueante, continúa su ejecución.
- Como la función `MPI_Wait()` fue eliminada, el proceso master continúa su ejecución sin garantía de que la operación `MPI_Irecv()` asociada a request haya finalizado.
- Siendo así, el proceso master itera una y otra vez sin esperar a que finalice la recepción de los mensajes y termina imprimiendo de forma continua lo que tenía dentro del buffer desde el principio: “No debería estar leyendo esta frase”.

```
Tiempo transcurrido 0.000002 (s):       proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 1)
Tiempo transcurrido 0.000044 (s):       proceso 0, MPI_IRecv() devolvio el control..
                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 0.000062 (s):       proceso 0, operacion receive completa con mensaje: No deberia estar leyendo esta frase.
Tiempo transcurrido 0.000069 (s):       proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 2)
Tiempo transcurrido 0.000077 (s):       proceso 0, MPI_IRecv() devolvio el control..
                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 0.000090 (s):       proceso 0, operacion receive completa con mensaje: No deberia estar leyendo esta frase.
Tiempo transcurrido 0.000096 (s):       proceso 0, llamando a MPI_IRecv() [no bloqueante] (fuente rank 3)
Tiempo transcurrido 0.000103 (s):       proceso 0, MPI_IRecv() devolvio el control..
                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 0.000115 (s):       proceso 0, operacion receive completa con mensaje: No deberia estar leyendo esta frase.

Tiempo total = 0.000000 (s)
```

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?

Nota: Para el caso de P = 16, agregue la línea --overcommit al script de SLURM y el flag --oversubscribe al comando mpirun.

Compilación y ejecución

NOTA:
N = 40000000 arroja un error cuando P = 16 tanto en blocking-ring como non-blocking ring, como se puede ver a continuación:

blocking-ring:

Primary job terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been aborted.

---

mpirun noticed that process rank 0 with PID 3982 on node nodo1 exited on signal 9 (Killed).

---

non-blocking-ring:

[nodo1:03661] Read -1, expected 320000000, errno = 3

Primary job terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been aborted.

---

mpirun noticed that process rank 3 with PID 3660 on node nodo1 exited on signal 9 (Killed).

Por esto, decidimos usar N = 30000000 en vez de 40000000.

Para compilar usamos:

- **Para blocking-ring.c**: mpicc -o blocking-ring blocking-ring.c
- **Para non-blocking-ring.c**: mpicc -o non-blocking-ring non-blocking-ring.c

Para ejecutar los programas compilados en el cluster, usamos el script ***ejercicio3.sh***:

```
#!/bin/bash
#SBATCH -N 1
#SBATCH --exclusive
#SBATCH --tasks-per-node=4
#SBATCH -o ./output_ejecutable_P_4_N_10000000.txt
#SBATCH -e ./errors_ejecutable_P_4_N_10000000.txt
mpirun ejecutable $1
```

donde:

- El `-N` es 1 porque usamos un único nodo.
- El `tasks-per-node` (P) lo vamos modificando entre 4, 8 y 16.
- `ejecutable` se intercambia entre non-blocking-ring y blocking-ring.
- El N se va reemplazando por los 3 valores pedidos:
  - **10000000**
  - **20000000**
  - **30000000**
- Cuando `tasks-per-node` = 16:
  - Usamos mpirun con argumento `–oversubscribe`.
  - Agregamos la sentencia `#SBATCH –overcommit` al script.

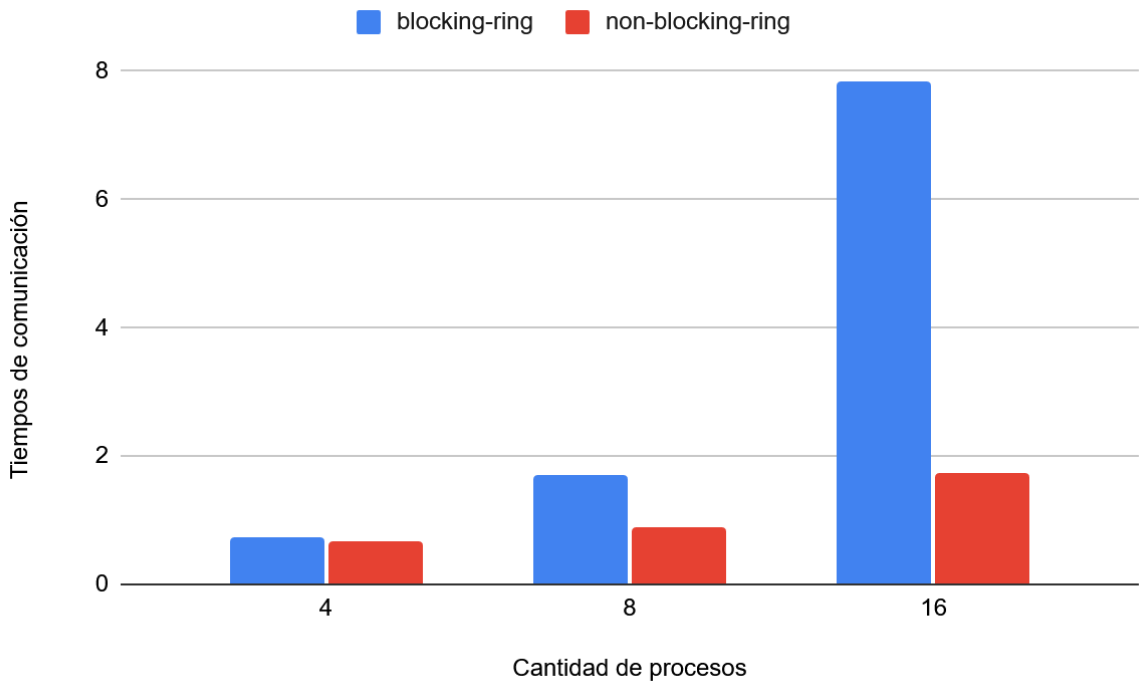
Tiempos de comunicación blocking-ring

Cantidad de procesos	**N**		
	10000000	**20000000**	**30000000**
4	0.249420	0.496207	0.741071
8	0.572591	1.137441	1.698961
16	2.367807	4.615786	7.851606

Tiempos de comunicación non-blocking-ring

Cantidad de procesos	N		
	10000000	20000000	30000000
4	0.217382	0.442413	0.668639
8	0.294817	0.588605	0.884322
16	0.611664	1.221354	1.721714

Comparación de tiempos de comunicación, N = 30000000



¿Cuál de los dos algoritmos requiere menos tiempo de comunicación? ¿Por qué?

non-blocking-ring.c requiere menos tiempo de comunicación debido a que utiliza las operaciones no bloqueantes de MPI:

- `MPI_Isend()`
- `MPI_Irecv()`

Estas operaciones permiten que los procesos comiencen un envío e inmediatamente pasen a realizar una recepción sin que el envío se realice completamente.

Por otro lado, en el algoritmo **blocking-ring.c** cada proceso recibe datos del proceso anterior y envía datos al proceso siguiente, con excepción del proceso 0 que recibe de task - 1 cerrando así la topología de anillo. Debido a cómo funciona esta topología y al uso de las operaciones bloqueantes [`MPI_Send()` y `MPI_Recv()`] cada proceso puede continuar únicamente cuando se completa la operación de recepción (o envío). Esto genera peores tiempos de comunicación.

Parte 2)

Dada la siguiente expresión:

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

- **Donde A, B, C 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.**
- B^T **es la matriz transpuesta de B.**

Desarrolle 2 algoritmos que computen la expresión dada:

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

Mida el tiempo de ejecución de los algoritmos en el cluster remoto. Las pruebas deben considerar la variación del tamaño de problema (N = {512, 1024, 2048, 4096}) y la cantidad de núcleos:

- **P = {8, 16, 32} para el algoritmo MPI.**
  - **1, 2 y 4 nodos, respectivamente.**
- **P = {16, 32} para el algoritmo híbrido.**
  - **2 y 4 nodos, respectivamente.**

Además de los algoritmos (archivos .c), debe elaborar un informe en PDF que describa brevemente las soluciones planteadas (puede emplear pseudo-código, diagramas, figuras, etc) e incluya las tablas y gráficos con los tiempos de ejecución y valores de Speedup, Eficiencia y overhead de comunicación para cada caso, siguiendo las recomendaciones vistas en clase. Además, debe analizar su rendimiento y escalabilidad (individual y comparativamente):

- **En el caso de P = 8, compare el rendimiento del algoritmo MPI con el de Pthreads / OpenMP (el que mejor rendimiento haya tenido) del Trabajo Práctico Nº 2.**

- **En el caso de P = {16, 32}, compare el rendimiento del algoritmo MPI con el del híbrido.**

Por último, recuerde aplicar las técnicas de programación y optimización vistas en clase.

Explicación de los algoritmos

Algoritmo secuencial

Explicado en el TP anterior.

Algoritmo paralelo empleando MPI

1. Para empezar, se usan las funciones de inicialización `MPI_Init()`, `MPI_Comm_size()` y `MPI_Comm_rank()` para, respectivamente:
    a. **Inicializar el entorno MPI;**
    b. **Obtener la cantidad de procesos involucrados;**
    c. **Que cada proceso obtenga su ID único**.
2. Después se calcula:
    a. El tamaño de las submatrices de A, C, a_por_b, c_por_bt, y R que tendrá cada proceso.
    b. El **tamaño de bloque** en base al tamaño de las submatrices: si este último es **menor** que el tamaño de bloque, el tamaño de bloque será igual al tamaño de las submatrices. La idea es evitar acceder a posiciones de memoria inválidas.
        - El tamaño de bloque por defecto es **128**.
        - Ejemplo 1: Con N = 512 y 32 procesos, el tamaño de las submatrices es 16, pero el tamaño de bloque es 128. Entonces el tamaño de bloque pasa a ser de 16.
        - Ejemplo 2: Con N = 4096 y 32 procesos, el tamaño de las submatrices es 128, lo cual **no es menor** que el tamaño de bloque por defecto, por ende éste último no se modifica, y se mantiene en 128.
3. Se **aloca memoria** para las matrices:
    a. El proceso master aloca memoria para las matrices enteras (de tamaño $N \times N$) A, C, a_por_b, c_por_bt y R.
    b. Los workers alocan memoria para submatrices (de tamaño $N \times tamañoSubmatriz$) A, C, a_por_b, c_por_bt y R.
    c. Por otro lado, todos los procesos necesitan de las matrices B y B^T **completas**, (cada worker necesita todas las columnas de B y B^T para poder hacer las multiplicaciones) es por esto que tanto el master como los workers las alocan con tamaño $N \times N$.

4. Las matrices A y C se inicializan como **matrices identidad** mientras que la matriz B se inicializa con valores de 1 a $N \times N$.
5. Se usa una barrera después de las inicializaciones y antes de seguir, para asegurar que todos los procesos empiecen a hacer sus cálculos aproximadamente al mismo tiempo de forma ordenada.
6. Se calcula el tiempo de inicio del cómputo utilizando la función `dwalltime()`. Además, se medirán los tiempos de cada comunicación MPI involucrada en el envío o recepción de mensajes.
7. El proceso master le envía a cada worker un pedazo de las matrices A y B para que inicialicen de esa forma sus submatrices, vía un **Scatter,** y la matriz B completa vía un **Broadcast**.
    a. **Primera comunicación** que se mide.
8. Luego, cada worker calcula el máximo, mínimo y promedio de su submatriz A y B:
    a. Como cada worker recibió la matriz B completa, se debe especificar en el bucle for la porción que recorrerá cada uno para evitar pisarse.
    b. Distinto es con el recorrido de la matriz A, ya que cada worker tiene su porción de la matriz (porque se utilizó **Scatter**) que recorre por completo.
    c. Cada worker usa tres arreglos de dos posiciones cada uno donde almacenar los máximos, mínimos y promedios, y de esta forma poder comunicarlos en 3 comunicaciones (donde se envía cada arreglo) en vez de 6.
    d. Una vez realizados los cálculos anteriores, todos los procesos ejecutan **Allreduce** con las operaciones `MPI_MAX`, `MPI_MIN` y `MPI_SUM`, para obtener los máximos, mínimos y promedios globales.
    e. **Segunda comunicación** que se mide.
9. Tras esto, todos los procesos calculan los dos promedios y el cociente.
10. A continuación, el proceso master inicializa la matriz B^T y se la envía a todos los workers usando **Broadcast**.
    a. **Tercera comunicación** que se mide.
11. Se resuelven las multiplicaciones de matrices.

12. Luego, cada worker resuelve de forma local la operación: $(Cociente \times [Resultado\ de\ A \times B]) + [Resultado\ de\ B \times B^T]$ y guarda los resultados en su porción de R.
13. Finalmente, utilizando **Gather**, el proceso master obtiene todos los resultados parciales de R y los junta en la matriz R, obteniendo el resultado final.
    a. **Cuarta comunicación** que se mide.
14. Para finalizar la medición del tiempo de ejecución el master calcula el tiempo de fin usando la función `dwalltime()`. Luego, obtiene el tiempo de comunicación total de cada proceso y los suma en la variable `tiempo_comunicacion_programa` mediante la función `MPI_Reduce()` con la operación `MPI_SUM.`
15. El master obtiene las submatrices de a_por_b y c_por_bt y las combina en matrices totales para poder chequear los resultados.
    a. Como esto es técnicamente opcional, no se mide su tiempo de comunicación.
16. El master obtiene el tiempo total de ejecución del programa calculando la diferencia entre `tiempo_fin` y `tiempo_inicio`. Para obtener el tiempo total de comunicación en promedio del programa, promedia la suma de todos los tiempos de comunicación obtenidos de cada proceso. Finalmente imprime los resultados.
17. El master chequea que los resultados sean correctos.
18. Se libera la memoria de las matrices.
19. Se cierra el entorno MPI.

Algoritmo paralelo híbrido empleando MPI + OpenMP

1. La cantidad de hilos que cada proceso usará se obtiene de la variable de entorno `OMP_NUM_THREADS`, la cual se inicializa dentro del script .sh.
2. Para empezar, se usan las funciones de inicialización `MPI_Init_thread()`, `MPI_Comm_size()` y `MPI_Comm_rank()` para, respectivamente:
    a. **Inicializar el entorno MPI para procesos multihilados con `MPI_THREAD_FUNNELED`, lo cual indica que los procesos pueden ser multi-hilo pero todas las comunicaciones las realizará solo el hilo master;**
    b. **Obtener la cantidad de procesos involucrados;**

    c. **Que cada proceso obtenga su ID único**.
3. Luego se calcula el **tamaño de bloque** en base al tamaño de las submatrices que crea cada proceso worker, pero además, se tiene en cuenta la cantidad de hilos a utilizar.
4. Creación de las matrices (asignación de memoria):
    a. Después el proceso master crea las matrices enteras A, C, a_por_b, c_por_bt, y R.
    b. Los procesos workers crean sus submatrices locales de esas mismas matrices.
    c. Ambos tipos de procesos crean B y B_T.
5. El master y los workers inicializan sus matrices.
6. El master le envía a cada worker su porción correspondiente de las matrices A y C vía un **Scatter** y la matriz B completa vía un **Broadcast**.
    a. **Primera comunicación** que se mide.
7. Cada proceso define su región paralela de OpenMP, donde sus 8 hilos hacen lo siguiente:
    a. Se usa `schedule(static)` al operar con las matrices para que OMP haga la división automáticamente y no tengamos que usar variables de inicio y fin para que los hilos no se pisen.
    b. Se calcula el máximo, mínimo y promedio de la submatriz de A de ese proceso usando **reductions** igual que en el TP2.
        i. Se usa `nowait` ya que no es necesario esperar a que los 8 hilos terminen para poder pasar a los cálculos de B.
    c. Para obtener el máximo, mínimo y promedio de B, cada proceso tiene una copia de la matriz B **entera** y no una submatriz. Es por eso que se especifica en la instrucción for el rango de inicio y fin adecuado a cada proceso, según su rank.
        i. Nuevamente se usa `nowait` por la misma razón.
    d. Se resuelve la multiplicación de matrices AxB.
    e. Ahora **solo el hilo master de cada proceso** hace lo siguiente:
        i. Todos los procesos worker le envían al **proceso master** sus mínimos máximos y promedios, los cuales se reducen a los mínimos máximos y promedios finales vía un `MPI_Reduce()`.
            1. **Segunda comunicación** que se mide.
        ii. Con estos datos, el **proceso master** calcula el cociente, inicializa la matriz B_T y utiliza el proceso de comunicación

`MPI_Bcast()` para enviar estos datos a todos los procesos worker.

1. **Tercera comunicación** que se mide.

f. Se usa `omp barrier` para evitar que los hilos empiecen a calcular la multiplicación de CxB_T antes de haber recibido B_T completa.
g. Se resuelve CxB_T.
h. Por último, se resuelve la operación final para obtener los resultados parciales de la matriz R.

8. Finalmente, utilizando `MPI_Gather()`, el proceso master obtiene todos los resultados parciales de R y los junta en la matriz R, obteniendo el resultado final.
    a. **Cuarta comunicación** que se mide.
9. El master obtiene las submatrices de a_por_b y c_por_bt y las combina en matrices totales para poder chequear los resultados.
    a. Como esto es técnicamente opcional, no se mide su tiempo de comunicación.
10. El master calcula el tiempo total de ejecución y el tiempo de comunicación total en promedio y los imprime. **Estos dos tiempos se obtienen siguiendo exactamente la misma metodología del algoritmo MPI, que ya fue explicada**.
11. El master chequea que los resultados sean correctos.
12. Se libera la memoria de las matrices.
13. Se cierra el entorno MPI.

Tiempos de ejecución y comunicación

Algoritmo secuencial

- Compilado con: **gcc -O3 -Wall -o secuencial secuencial.c**
- Enviado a ejecutar al cluster con el script: **script-secuencial.sh**

N			
512	**1024**	**2048**	**4096**
0.418919	3.410937	26.937330	214.812314

Algoritmo paralelo empleando MPI

- Compilado con: **mpicc -O3 -Wall -o mpi mpi.c**
- Enviado a ejecutar al cluster con el script: **script-mpi.sh**

Ejecución (1, 2 y 4 nodos, 8 procesos monohilo por nodo)

Cantidad de procesos monohilo	**N**			
	512	**1024**	**2048**	**4096**
8	0.09218	0.62347	4.424171	96.346883
16	0.107204	0.637293	3.429710	50.239625
32	0.124605	0.554346	2.791522	55.949959

Comunicación (1, 2 y 4 nodos, 8 procesos monohilo por nodo)

Cantidad de procesos monohilo	**N**			
	512	**1024**	**2048**	**4096**
8	0.031512	0.155129	0.697771	15.361687
16	0.067169	0.368723	1.474979	13.417805
32	0.092201	0.385745	1.723771	20.681750

Algoritmo paralelo híbrido empleando MPI + OpenMP

- Compilado con: **mpicc -O3 -Wall -fopenmp -o hibrido hibrido.c**
- Enviado a ejecutar al cluster con el script: **script-hibrido.sh**

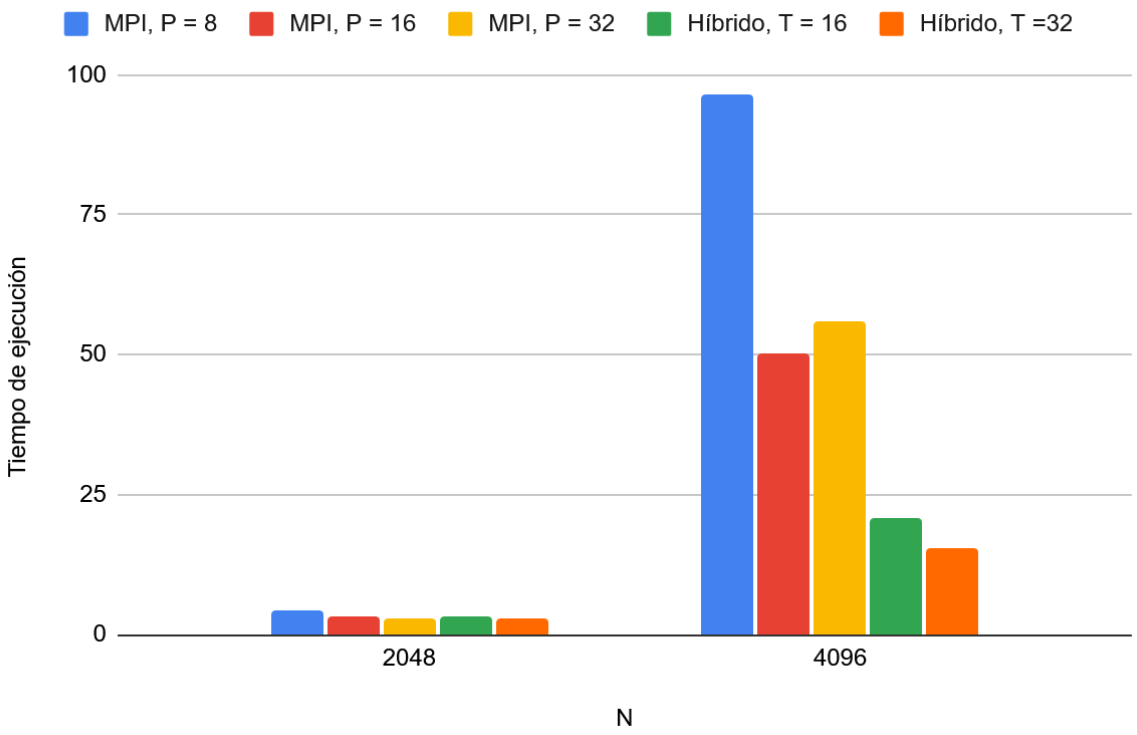
Ejecución (2 y 4 nodos, un proceso por nodo, 8 hilos por proceso)

Cantidad de hilos	**N**			
	512	**1024**	**2048**	**4096**
16	0.107996	0.603035	3.361438	20.9922
32	0.113555	0.559191	2.757774	15.34206

Comunicación (2 y 4 nodos, un proceso por nodo, 8 hilos por proceso)

Cantidad de hilos	**N**			
	512	**1024**	**2048**	**4096**
16	0.070427	0.318095	1.328953	5.317734
32	0.087097	0.397237	1.669952	7.052963

Gráfico comparativo con N = 2048 y N = 4096



Overhead de las comunicaciones

El overhead de las comunicaciones de un sistema paralelo se define como la **relación entre el tiempo requerido por las comunicaciones de la solución y el tiempo total que ésta requiera**. Se define por la fórmula:

$$OverheadComunicaciones(n) = \frac{Tcomunicaciones_p(n)}{Ttotal_p(n)}$$

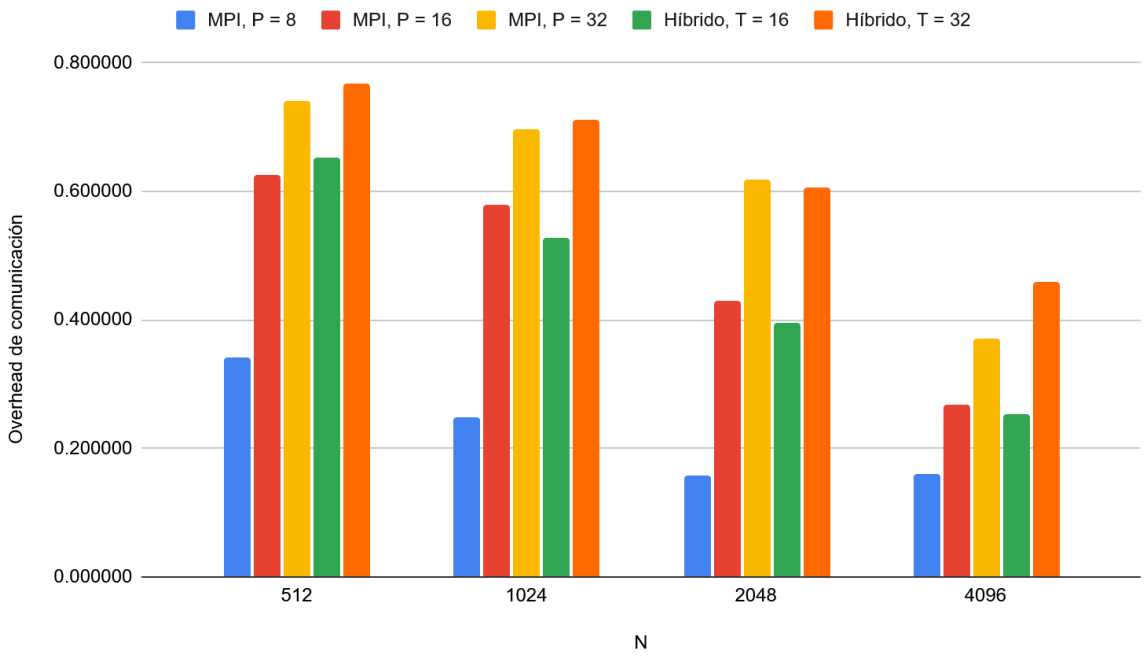
Algoritmo paralelo empleando MPI

Cantidad de procesos monohilo	N			
	512	**1024**	**2048**	**4096**
8	0.341853	0.248816	0.157718	0.159441
16	0.626553	0.578577	0.430059	0.267076
32	0.739946	0.695856	0.617502	0.369647

Algoritmo paralelo híbrido empleando MPI + OpenMP

Cantidad de hilos	N			
	512	**1024**	**2048**	**4096**
16	0.652126	0.527490	0.395353	0.253320
32	0.767003	0.710378	0.605543	0.459714

Gráfico comparativo



Speedup

El Speedup indica qué tanto beneficio obtenemos al usar la solución paralela comparado a usar la solución secuencial. Se define por la fórmula:

$$\frac{TiempoSecuencial}{TiempoParalelo}$$

donde:

- El tiempo secuencial debe obtenerse usando el algoritmo secuencial más rápido/óptimo posible para el problema dado.
- Ambos tiempos dependen del tamaño de problema (N).
- El tiempo paralelo también depende de la cantidad de hilos que se usan.
- Si el resultado del cociente es mayor a 1, la solución paralela es mejor.
- Si el resultado del cociente es igual a la cantidad de procesadores, el speedup es óptimo.
- Si el resultado del cociente es mayor a la cantidad de procesadores, el speedup es superlineal.

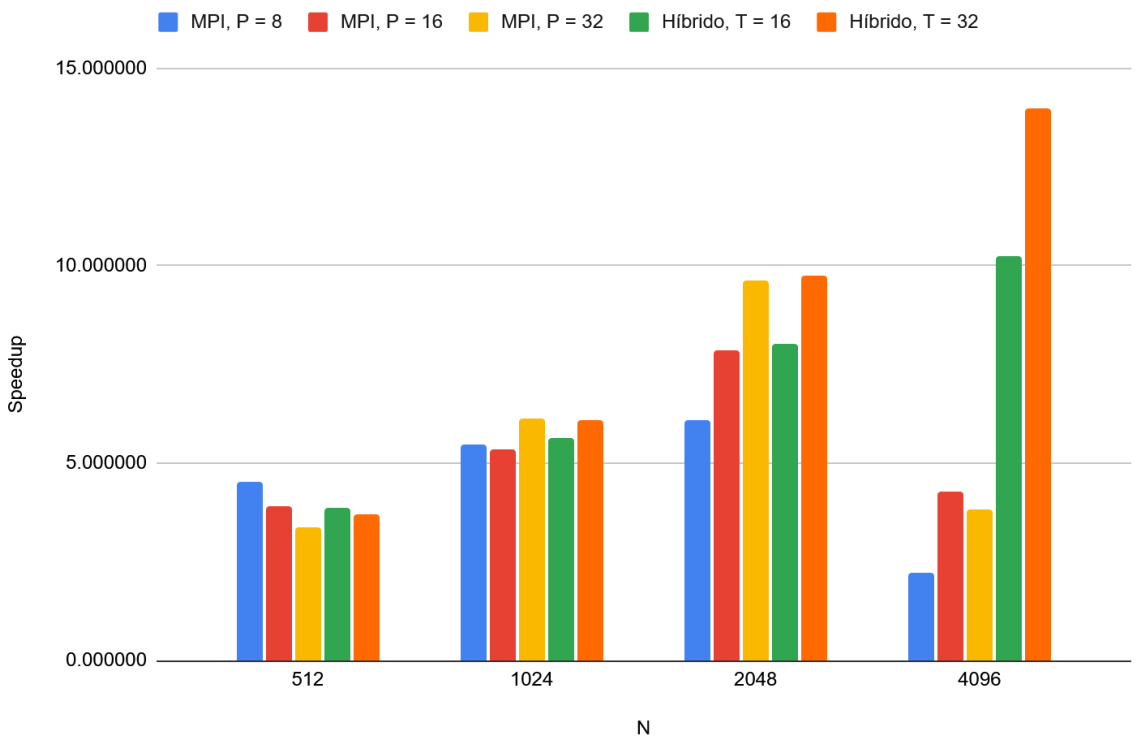
Algoritmo paralelo empleando MPI

Cantidad de procesos monohilo	**N**			
	512	**1024**	**2048**	**4096**
8	4.544576	5.470892	6.088673	2.229572
16	3.907681	5.352227	7.854113	4.275755
32	3.361976	6.153083	9.649693	3.839365

Algoritmo paralelo híbrido empleando MPI + OpenMP

Cantidad de hilos	**N**			
	512	**1024**	**2048**	**4096**
16	3.879023	5.656284	8.013633	10.232959
32	3.689129	6.099771	9.767780	14.001530

Gráfico comparativo



Eficiencia

La eficiencia indica qué tan bien se están aprovechando los recursos que se tienen, es decir los procesadores. En arquitecturas homogéneas, como la del Multicore Blade que usamos, se define por la fórmula:

$$\frac{Speedup}{Número\ de\ procesadores}$$

donde:

- Como nunca usaremos más hilos que la cantidad de núcleos que tiene Blade (8), si usamos X hilos, se usarán X procesadores.
- El resultado está entre 0 y 1.
- Cuanto más se acerca a 1, más se está aprovechando la arquitectura.
- Cuanto más se acerca a 0, menos se está aprovechando la arquitectura.

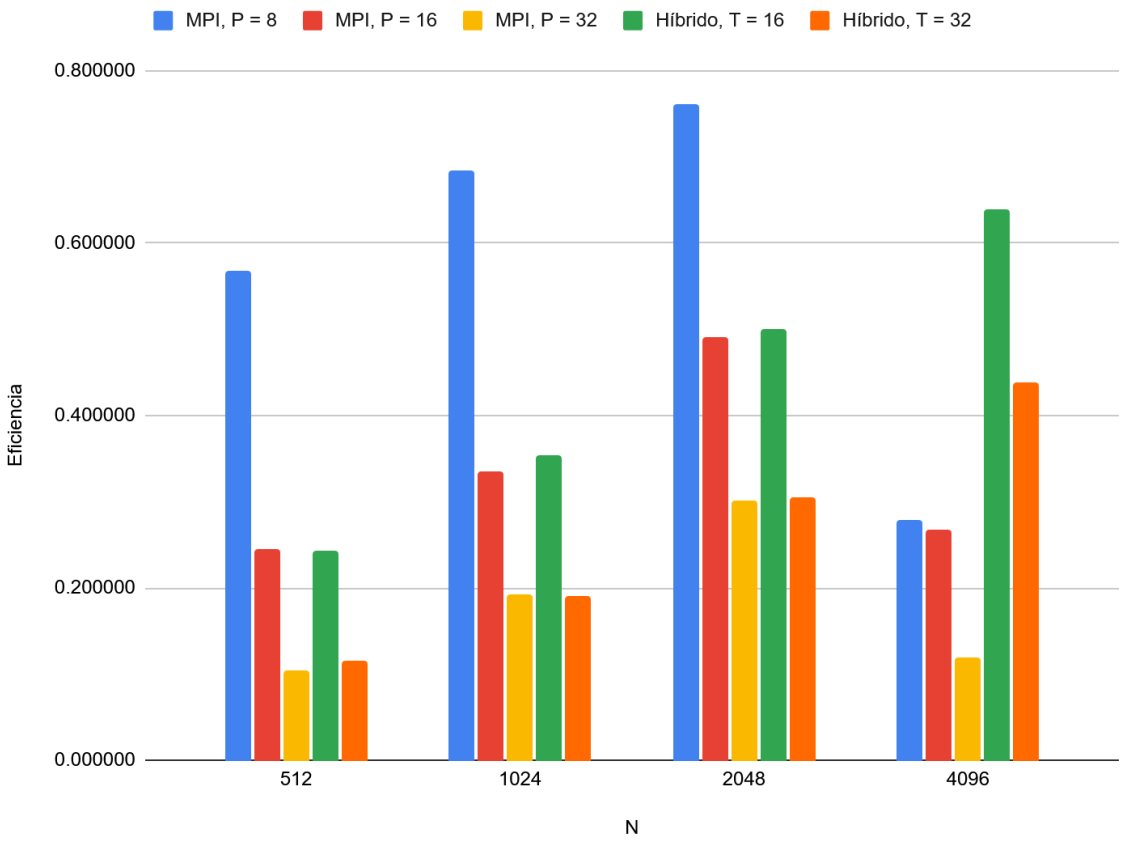
Algoritmo paralelo empleando MPI

Cantidad de procesos monohilo	**N**			
	512	**1024**	**2048**	**4096**
8	0.568072	0.683861	0.761084	0.278697
16	0.244230	0.334514	0.490882	0.267235
32	0.105062	0.192284	0.301553	0.119980

Algoritmo paralelo híbrido empleando MPI + OpenMP

Cantidad de hilos	**N**			
	512	**1024**	**2048**	**4096**
16	0.242439	0.353518	0.500852	0.639560
32	0.115285	0.190618	0.305243	0.437548

Gráfico comparativo



Análisis de rendimiento y escalabilidad

Análisis individual del algoritmo paralelo empleando MPI

- **Tiempo de ejecución**: Presenta tiempos de ejecución muchísimo mejores que el algoritmo secuencial, pero en general peores que el algoritmo híbrido.
    - Se puede ver que con `N = 4096`, el tiempo de ejecución mejora sustancialmente cuando pasamos de `P = 8` → `P = 16`, pero sin embargo empeora cuando pasamos de `P = 16` → `P = 32`, indicando que el overhead aumenta y ya no hay mejora de rendimiento, si no todo lo contrario, una empeora.
- **Overhead de las comunicaciones:** Presenta niveles de overhead de este tipo bastante altos, debido a la gran cantidad de procesos que utiliza, a diferencia del híbrido que usa solo 2 y 4.
    - A medida que N aumenta, el overhead disminuye. Al mismo tiempo, el overhead aumenta cuanto mayor es el número de procesos (P), lo cual es lógico ya que a más procesos, más comunicaciones hay entre ellos.
- **Speedup:** Al incrementar el número de unidades de procesamiento (usando más nodos) el speedup mejora, especialmente en tamaños intermedios de matrices (`N = 2048`). Sin embargo, En `N = 4096` el Speedup decae fuertemente. Esto es llamativo ya que el overhead de comunicación pasando de 2048 → 4096 baja, y sin embargo el Speedup con `N = 4096` cae: se supone que una baja de overhead implica una mejora del Speedup.
- **Eficiencia:** La eficiencia empeora cuantos más procesos usamos, aunque mejora a medida que N crece, excepto cuando `N = 4096`. En ese valor de N la eficiencia colapsa, debido a que no se está aprovechando bien la arquitectura: nuevamente esto es llamativo debido a que el overhead de comunicación con ese valor de N no es tan alto como para que la eficiencia empeore a tal nivel.

Dado este análisis, podemos decir que el algoritmo **no es escalable,** debido a que el algoritmo no es capaz de mantener un buen nivel de eficiencia a medida que N y P crecen. Además, el Speedup no crece de forma proporcional al número de unidades de procesamiento.

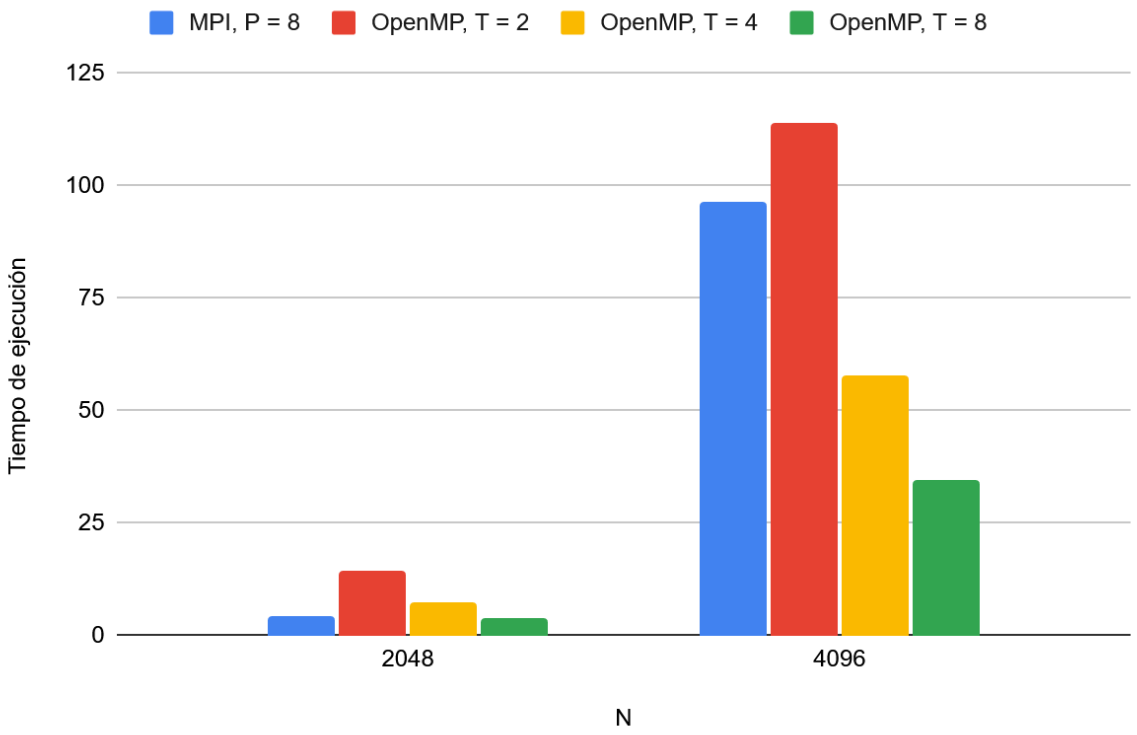
Análisis individual del algoritmo paralelo híbrido empleando MPI + OpenMP

- **Tiempo de ejecución**: Presenta tiempos de ejecución mejores que en el algoritmo MPI puro a medida que **N crece**.
    - Además, aumentar la cantidad de hilos mejora el tiempo de ejecución, como es de esperarse.
- **Overhead de las comunicaciones:** Presenta niveles de overhead de este tipo similares aunque en algunos casos mejores al algoritmo MPI debido a que usa muchos menos procesos y por ende produce muchas menos comunicaciones entre ellos. Además, este overhead se va reduciendo a medida que N crece.
- **Speedup:** Al incrementar el número de unidades de procesamiento el speedup **mejora siempre** a medida que N aumenta.
- **Eficiencia:** Al igual que con el Speedup, la eficiencia mejora siempre a medida que N aumenta. Es decir, cuanto mayor sea N, más se aprovecha la arquitectura usando este algoritmo.

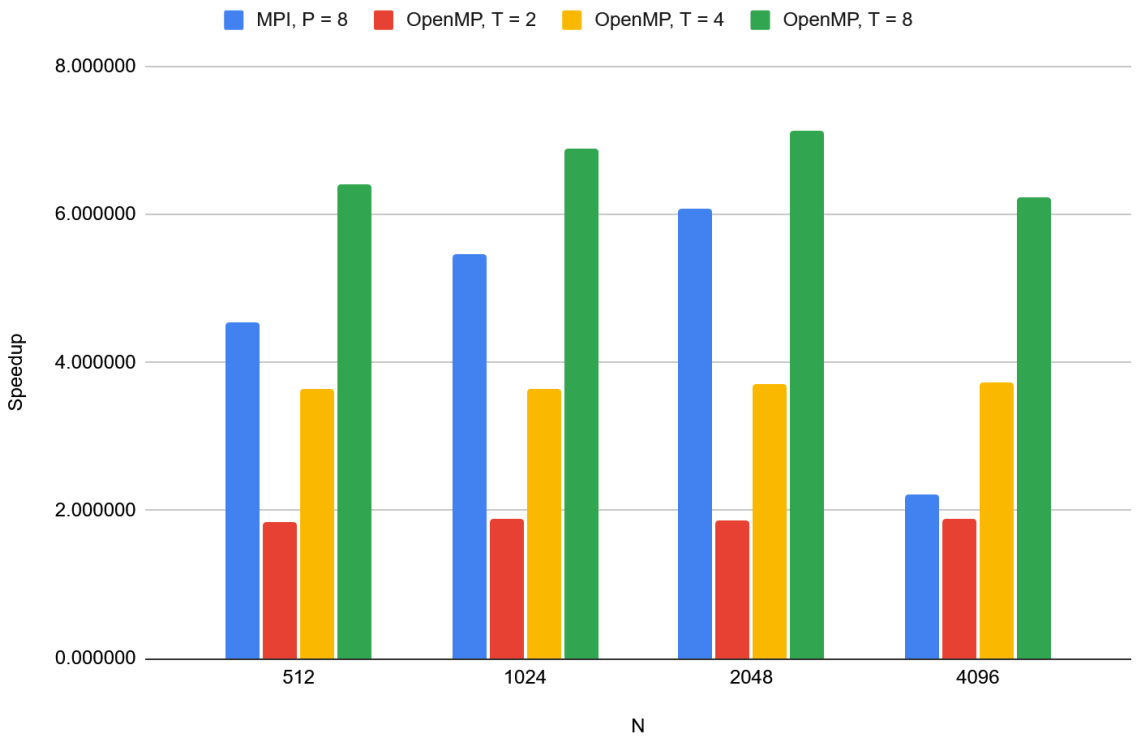
Dado este análisis, podemos decir que el algoritmo **ES escalable,** debido a que es capaz de ir mejorando el nivel de eficiencia a medida que N y P crecen.

En términos de escalabilidad débil vs fuerte, podemos determinar que el algoritmo híbrido presenta escalabilidad de tipo **débil**, ya que la eficiencia decae cuando usamos más hilos pero mantenemos el N fijo. Esto se puede ver fácilmente en el gráfico comparativo de eficiencia, donde en cada valor de N, si duplicamos la cantidad de hilos de 16 a 32, la eficiencia baja. **Por consiguiente, para mantener la eficiencia o aumentarla es necesario que tanto N como la cantidad de unidades de procesamiento crezcan, y no sólo ésta última**.

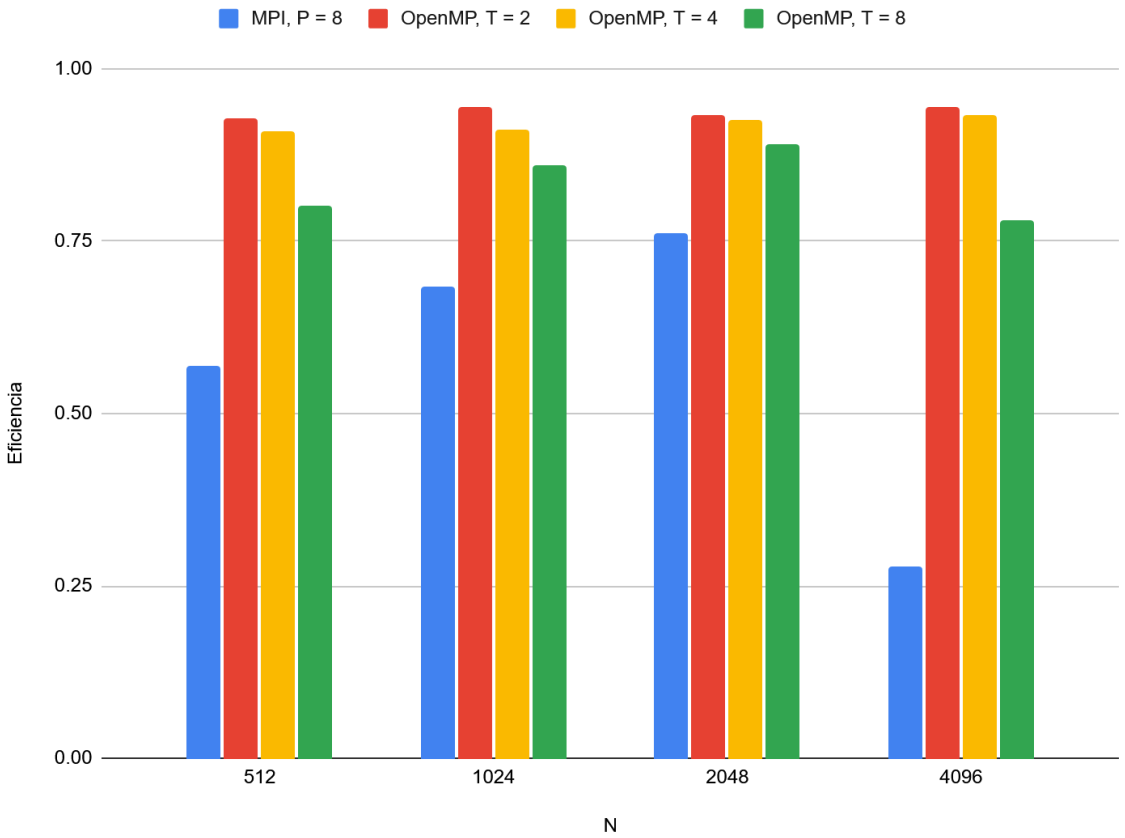
Comparación algoritmo paralelo empleando MPI, con P = 8, vs OpenMP del TP N°2



Al comparar los **tiempos de ejecución** encontramos que el algoritmo de OpenMP presenta mejores tiempos a medida que N crece (excepto cuando se usan 2 hilos, lo cual es entendible). Esto probablemente se debe a que en la versión de OpenMP no hay ningún overhead de comunicación entre los procesos debido al pasaje de mensajes. El algoritmo OpenMP usa exclusivamente memoria compartida.



De la misma manera, el **speedup** es mejor en la versión de OpenMP a medida que N crece, **siempre y cuando se use una cantidad razonable de hilos y no pocos**. Resulta superior usar 8 hilos en un mismo proceso (OpenMP) que usar 8 procesos con un hilo cada uno (MPI), ya que de esta manera se aprovecha la memoria compartida de ese único proceso y no se necesita ninguna comunicación vía redes de interconexión que son mucho más lentas.



Por último, el gráfico comparativo de la **eficiencia** de ambos algoritmos muestra claramente que la solución con OpenMP aprovecha mucho mejor la arquitectura en todos los casos, sin importar el valor de N.

Comparación algoritmo paralelo empleando MPI vs algoritmo paralelo híbrido empleando MPI + OpenMP, con P = {16, 32}

- **Menor tiempo total de ejecución en algoritmo híbrido:**
  - La diferencia es más apreciable cuanto mayor es N.
  - Algoritmo Híbrido con `T = 32` y `N = 4096` → ~15s.
  - Algoritmo MPI con `P = 32` y `N = 4096` → ~55s.
- **Similar overhead de comunicación:**
  - El overhead es similar en ambos algoritmos, lo cual resulta ser un comportamiento llamativo ya que se esperaría que en el caso del algoritmo híbrido el overhead fuera menor debido al menor número de procesos involucrados en el cómputo.
- **Mejor speedup en el algoritmo híbrido cuando N es grande:**

    - Esto sugiere que la versión híbrida es más escalable → mejora el rendimiento al aumentar el número de nodos/hilos y N.
- **Mejor eficiencia en el algoritmo híbrido cuando N es grande:**
    - La **eficiencia del algoritmo híbrido mejora** a medida que se incrementa N:
        - Teniendo en cuenta N = 4096:
        - La versión híbrida presenta una eficiencia de 0.639 con 16 hilos y 0.437 con 32.
        - En contraste, la eficiencia de MPI puro cae a valores muy malos de 0.267 con 16 procesos y 0.119 con 32 procesos.
    - Sin embargo, para los casos de N = 512, 1024 y 2048, el algoritmo MPI presenta una buena eficiencia con 8 procesos, superando a las demás opciones.
    - En definitiva, el **algoritmo híbrido presenta una mejor eficiencia si el objetivo es escalar**, especialmente en problemas grandes y con más hilos. La causa principal de este comportamiento es la siguiente:
        - **MPI puro escala mediante más procesos**, lo que genera mayor overhead de comunicación a medida que aumenta P
        - **MPI + OpenMP mantiene pocos procesos**, y aprovecha el paralelismo de OpenMP, que es más eficiente dentro de los nodos al aprovechar la memoria compartida.

Conclusiones

Como conclusión del trabajo, se puede mencionar que la solución híbrida resulta generalmente mejor con respecto a la solución MPI en cuanto a los tiempos de ejecución, speedup y eficiencia. Además, debido a las características del sistema utilizado, el algoritmo híbrido resulta más escalable porque hace un mejor aprovechamiento de lo recursos

Sin embargo, la elección del tipo de algoritmo a utilizar dependerá por un lado de la **arquitectura** de la que se disponga, y por otro lado del **tamaño** y **tipo** del problema a resolver:

- Para problemas grandes donde se requiere escalabilidad, la opción híbrida haciendo uso de un cluster es la mejor.
- Si no se tiene acceso a un cluster, la opción más viable es usar OpenMP.
- La alternativa de MPI es viable siempre y cuando el número de procesos se adecúe a la arquitectura.