Sistemas Paralelos
Entrega 3

Integrantes:

13926/1 Cazorla, Ignacio Nicolás
13655/7 Delle Donne, Matías Adrián

1. Ejercicio 2

El algoritmo non-blocking retorna inmediatamente el control. Esto sucede porque retorna el control al haber depositado la solicitud en el buffer local del sistema, en cambio el blocking debe esperar a que el receptor reciba el mensaje.

El non-blocking devuelve aproximadamente 2 seg antes el control ya que no debe esperar el sleep de 2 seg como si lo hace el blocking.

Ambos tardan en terminar lo mismo en tiempo, ya que el proceso en el non-blocking tiene una cláusula wait para esperar la recepción del mensaje completo antes de pasar al siguiente emisor.

Si sacamos el ***MPI_Wait()*** los mensajes no se imprimen correctamente ya que el proceso master hará el for sin esperar a cada worker porque realiza un receive no bloqueante, es decir, solo dejara la solicitud pendiente en el buffer local del sistema y seguirá ejecutando.

Por eso imprime siempre "Este mensaje no se debería ver", siendo el valor por defecto que tiene la variable message.

1 nodo con 4 procesos => 4

Blocking	Non-blocking
2.000092	0.000011
4.000087	2.000089
6.000117	4.000066

2 nodos con 2 procesos => P=4

Blocking	Non-blocking
1.999964	0.00001
4.020545	2.000109
6.010489	4.010842

1 nodo con 8 procesos => P=8

Blocking	Non-blocking

2.000105	0.00001
4.00429	2.000081
6.004282	4.000087
8.000097	6.000075
10.000093	8.000287
12.000057	10.000265
14.000048	12.00008

Blocking	**Non-blocking**
2.000114	0.000016
4.000027	2.000183
6.000001	4.000133
8.000043	5.999989
9.999997	8.000119
12.000136	10.000251
14.000041	12.000097
16.011687	14.000229
18.010418	16.002053
20.01054	18.000984
22.010499	20.001134
24.010605	22.000999
26.010553	24.001207
28.010544	26.001209
30.010469	28.001105

2. Ejercicio 3

El non-blocking requiere menos tiempo de comunicación. En blocking los procesos se comunican en forma de cascada (un proceso no puede continuar hasta que el anterior le envíe sus datos).

El último proceso será el primero en enviar sus datos invirtiendo las sentencias receive/send para no generar deadlock al utilizar comunicación bloqueante cada proceso debe esperar a recibir el vector completo antes de poder enviar su vector al proceso siguiente, esto hace que el último proceso sea el último en recibir su vector entonces se produce ocio.

En non-blocking una vez que las solicitudes de comunicación de recepción y envío fueron depositadas en el buffer local del sistema se devuelve inmediatamente el control al proceso.

Por lo tanto no se debe esperar al proceso anterior para enviar su vector al siguiente como sucede en el algoritmo blocking.

En este caso, todas las solicitudes de comunicación se realizan en paralelo, solo se debe esperar al final con la sentencia wait para garantizar que los envíos y las recepciones fueron recibidas por el proceso receptor.

1 nodo con 4 procesos => P=4

N	Tiempo de comunicación	Tiempo de comunicación
----------	Blocking	Non-blocking
10000000	0.248657	0.221127
20000000	0.494817	0.440358
40000000	0.980674	0.88357

2 nodos con 2 procesos => 4

N	Tiempo de comunicación	Tiempo de comunicación
----------	Blocking	Non-blocking
10000000	1.491586	0.749504
20000000	2.969807	1.544038
40000000	5.924863	3.045919

1 nodo con 8 procesos => P=8

N	Tiempo de comunicación	Tiempo de comunicación
----------	Blocking	Non-blocking
10000000	0.551336	0.297088
20000000	1.098626	0.590614
40000000	2.183434	1.175593

2 nodos con 8 procesos => P=16

N	Tiempo de comunicación	Tiempo de comunicación
----------	Blocking	Non-blocking
10000000	2.339335	0.94722
20000000	4.66059	2.028696
40000000	9.278954	3.964059

Dada la siguiente expresión:

$$R = \frac{Max\_F.Min\_F}{Prom\_F}.[A.B.C + D.Fib(F)]$$

− Donde A, B, C, D y R son matrices cuadradas de NxN con elementos de tipo double.
− F 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.
− Max_F y Min_F son los valores máximo y mínimo de la matriz F, respectivamente.
− Prom_F es el valor promedio de los elementos de la matriz F.
− La función Fib(F) aplica Fibonacci a cada elemento de la matriz F.

Desarrolle dos algoritmos que computen la expresión dada:

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

3. Algoritmo paralelo utilizando MPI

El algoritmo planteado para resolver la ecuación está implementado utilizando MPI, para obtener una solución con memoria distribuida. Los cálculos se llevarán a cabo por procesos que conocen una porción del problema que deben resolver. Ahí se encuentra la principal diferencia respecto a las soluciones anteriores (secuencial, OpenMP, Pthreads).

Se analizan entonces para esta sección dos cuestiones: cómo se reparten los datos entre los procesos, por un lado; cómo se realizan las comunicaciones, por el otro.

La **información fue distribuida** entre los procesos utilizando funciones que provee MPI, de la siguiente manera:

- ***MPI_Scatter()*** : para que un proceso (coordinador) reparta porciones (filas) de matrices al resto de los procesos.
- ***MPI_Bcast()*** : para que un proceso (coordinador) reparta matrices completas al resto de los procesos (matrices por columnas)
- ***MPI_Gather()*** : para que un proceso (coordinador) pueda recolectar los resultados obtenidos.
- ***MPI_Allgather()*** : para que todos recolecten información. Para nuestro caso la matriz con el cálculo de los valores de Fibonacci, que primero se distribuye por porciones a todos los procesos, pero luego es necesaria completa en todos para realizar una multiplicación.
- ***MPI_Allreduce()*** : para permitir determinar entre todos un máximo, mínimo y la suma en base a los valores locales que cada uno obtuvo.

Lo que se pretende, es aprovechar la descomposición brevemente descrita, para que cada proceso solo se encargue de resolver su parte del problema. Al final, un sólo proceso recolecta todos los resultados.

Los **tiempos de comunicación** se calculan de la siguiente manera: para cada etapa de comunicación se toma el tiempo antes y después. Para ello se utiliza la función ***MPI_Wtime()***.
La solución planteada realiza tres etapas de comunicación. La primera, donde se reparten todos los datos desde el proceso coordinador al resto de los procesos. La segunda, cuando todos reciben la matriz que tiene los valores de Fibonacci, más los valores máximo, mínimo y suma. La tercera, cuando todos le envían al coordinador sus porciones de resultados.

Los tiempos se calculan de la siguiente manera:

1. el tiempo total que toma resolver el problema se calcula como primer tiempo antes de la primera comunicación menos el tiempo después de la última comunicación. Es decir:

tiempoTotal = tiempo[n] - tiempo[0]

2. Tiempo de comunicación total, se sumó la diferencia entre cada tiempo previo y posterior a la comunicación. Es decir:

```
tiempoComTotal = (tiempo[n] - tiempo[n - 1]) + … + (tiempo[1] - tiempo[0])
```

Compilación

```
$ mpicc mpi-distribuido.c mmblk.c -o mpi-distribuido -O2
```

Ejecuciones

Los valores expresados entre paréntesis indican que la cantidad de filas que se le distribuyen a cada proceso es menor que el tamaño especificado de bloque. Esto se debe a la imposibilidad de asignarle una cantidad de filas acorde al tamaño de bloque, ya que la porción que le toca a cada proceso es muy pequeña.

N	BS	Tiempo secuencial	Tiempo paralelo	Tiempo de comunicación	Speedup	Eficiencia	Overhead de comunicaciones
512	64	0,716878	0,128058	0,034798	5,598072748	0,699759094	0,271736245
1024	64	5,715443	0,850408	0,117840	6,720824592	0,840103074	0,138568781
2048	64	45,611940	6,316809	0,547564	7,220724894	0,902590612	0,08668364
4096	64	372,316681	50,303595	1,894130	7,401393837	0,92517423	0,037653969

1 nodo (8 procesos)

N	BS	Tiempo secuencial	Tiempo paralelo	Tiempo de comunicación	Speedup	Eficiencia	Overhead de comunicaciones
512	64 (32)	0,716878	0,242138	0,192463	2,960617499	0,185038594	0,794848392
1024	64	5,715443	0,896835	0,530297	6,3729036	0,398306475	0,591298288
2048	64	45,611940	4,707953	1,796485	9,688274288	0,605517143	0,38158516
4096	64	372,316681	30,766454	6,717290	12,101384222	0,756336514	0,218331628

2 nodos (16 procesos)

N	BS	Tiempo secuencial	Tiempo paralelo	Tiempo de comunicación	Speedup	Eficiencia	Overhead de comunicaciones
512	64 (16)	0,716878	0,394440	0,367058	1,817457661	0,056795552	0,930580063
1024	64 (32)	5,715443	0,900692	0,704938	6,345613151	0,198300411	0,782662664
2048	64	45,611940	3,669880	2,213222	12,428727915	0,388397747	0,603077485
4096	64	372,316681	20,436369	8,397362	18,21833815	0,569323067	0,410902837

4 nodos (32 procesos)

Análisis de rendimiento y escalabilidad

A continuación se presenta la tabla de **speedup** indicando cuántas veces más rápido se ejecutará la solución con 8, 16 y 32 unidades de procesamiento, respecto de la secuencial:

Unidades de procesamiento	Tamaño de problema			
	512	1024	2048	4096
8	5,598072748	6,720824592	7,220724894	7,401393837
16	2,960617499	6,3729036	9,688274288	12,101384222
32	1,817457661	6,345613151	12,428727915	18,21833815

Speedup

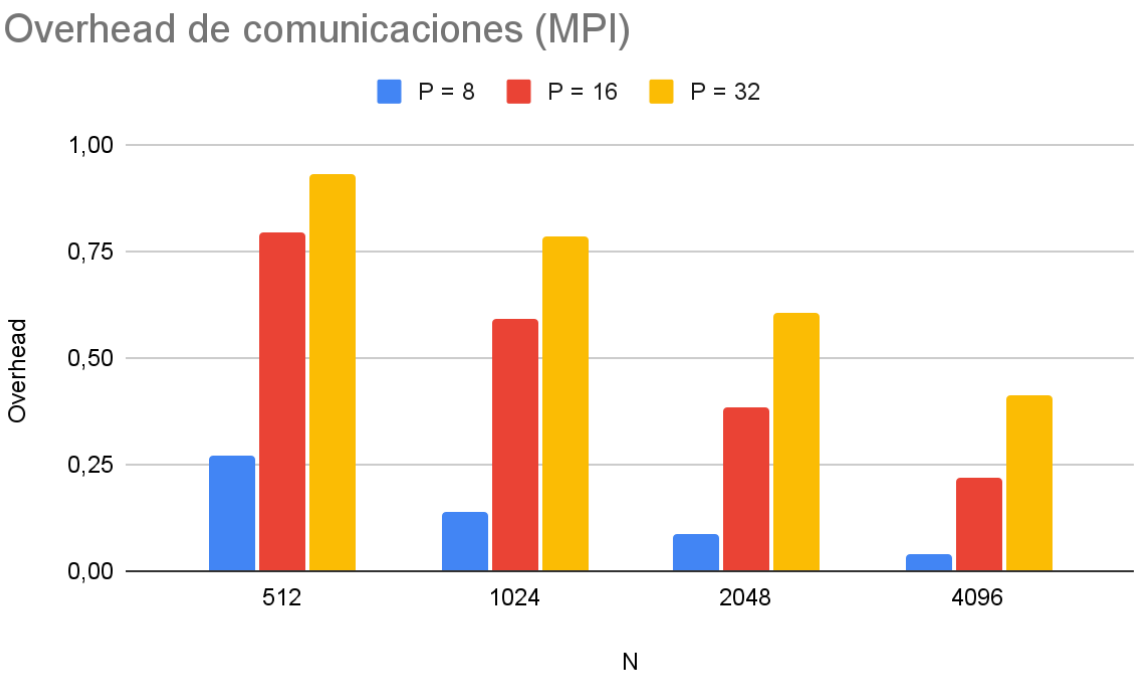
Los valores obtenidos de **eficiencia** parecen indicar que para problemas grandes obtendremos una mayor utilidad de las unidades de procesamiento:

Unidades de procesamiento	Tamaño de problema			
	512	1024	2048	4096
8	0,699759094	0,840103074	0,902590612	0,92517423
16	0,185038594	0,398306475	0,605517143	0,756336514
32	0,056795552	0,198300411	0,388397747	0,569323067

Eficiencia

Los resultados obtenidos indican una **escalabilidad débil**. Para mantener la eficiencia fija a medida que aumenta la cantidad de unidades de procesamiento, necesitamos aumentar el tamaño del problema.

El **overhead de comunicaciones** se representa en el gráfico siguiente:



El overhead se reduce a medida que crece el tamaño de problema. Mientras menos cantidad de procesos haya, menor será el overhead introducido por las comunicaciones.

4. Algoritmo paralelo utilizando MPI híbrido (MPI + OpenMP)

En base a la solución anterior, se plantea una versión que combina memoria distribuida (MPI) con memoria compartida (OpenMP).

Como una medida para reducir la cantidad de procesos y las comunicaciones entre ellos, se utilizarán hilos. Los hilos permitirán reducir la cantidad de procesos a uno por nodo, dónde cada proceso activo se encargará de enviar y recibir porciones del problema (aún tenemos la distribución por proceso) y además cada nodo contará con una serie de hilos resolviendo cada porción del problema.

Las operaciones a resolver se van a distribuir entre los hilos de cada nodo. La primera comunicación y la última se producen fuera del contexto objetivo de los hilos. De todas maneras la segunda comunicación entre los procesos se produce dentro de este contexto (para obtener la matriz que contiene los resultados de Fibonacci que es necesaria, y los valores máximo, mínimo y la suma). Esta

comunicación es llevada por un solo hilo de cada nodo, por lo que se establece una barrera para que el resto de los hilos espere.

Luego, las multiplicaciones entre las matrices contienen algunas dependencias. Se especificó que no haya una barrera implícita en las multiplicaciones, por lo que es necesario utilizar dos barreras más.

Finalmente, se realiza la suma correspondiente entre las matrices y se multiplica a cada valor por un escalar (calculado con el máximo, mínimo y el promedio de la matriz con los valores de Fibonacci), nuevamente evitando la barrera implícita de la directiva *for*.

Compilación

```
$ mpicc mpi-hibrido.c mmblk.c -o mpi-hibrido -O2 -fopenmp
```

Ejecuciones

N	BS	Tiempo secuencial	Tiempo paralelo	Tiempo de comunicación	Speedup	Eficiencia	Overhead de comunicaciones
512	64	0,716878	0,201127	0,105947	3,564305 141	0,222769 071	0,52676667
1024	64	5,715443	0,772551	0,398698	7,398143 294	0,462383 956	0,516079845
2048	64	45,611940	4,419554	1,519571	10,32048 4827	0,645030 302	0,343829038
4096	64	372,316681	29,186437	5,463878	12,75649 6485	0,797281 03	0,187206064

2 nodos (1 proceso por nodo con 8 hilos cada uno)

N	BS	Tiempo secuencial	Tiempo paralelo	Tiempo de comunicación	Speedup	Eficiencia	Overhead de comunicaciones
512	64	0,716878	0,245282	0,153651	2,922668 602	0,091333 394	0,62642591
1024	64	5,715443	0,874785	0,507116	6,533540 241	0,204173 133	0,579703584
2048	64	45,611940	3,390706	1,921215	13,45204 804	0,420376 501	0,566612086
4096	64	372,316681	19,457804	7,591907	19,13456 8372	0,597955 262	0,390172858

4 nodos (1 proceso por nodo con 8 hilos cada uno)

Análisis de rendimiento y escalabilidad

Para esta solución el **speedup** presenta una mejora más alta para tamaños de problema de 512 y 1024 (más chicos) con 16 hilos, y presenta una mejora más alta para los tamaños 2048 y 4096 (más grandes) con 32 hilos.

Unidades de procesamiento	Tamaño de problema			
	512	1024	2048	4096
16	3,564305141	7,398143294	10,320484827	12,756496485
32	2,922668602	6,533540241	13,45204804	19,134568372

Speedup

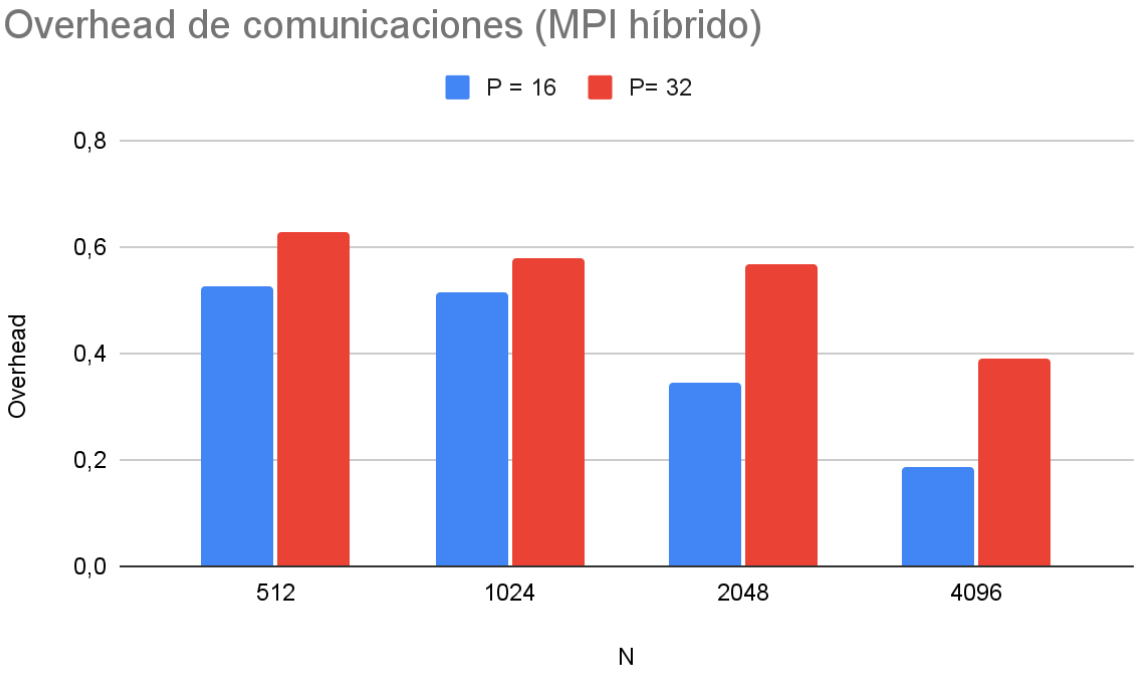
Se puede apreciar en la tabla de **eficiencia** que con 16 hilos y el tamaño de problema más grande, se obtuvo un valor de casi 80% de tiempo útil de las unidades de procesamiento.

Unidades de procesamiento	Tamaño de problema			
	512	1024	2048	4096
16	0,222769071	0,462383956	0,645030302	0,79728103
32	0,091333394	0,204173133	0,420376501	0,597955262

Eficiencia

La **escalabilidad es débil**. Es necesario aumentar el tamaño del problema si queremos obtener mayor eficiencia, cuando tenemos más cantidad de unidades de procesamiento.

En cuanto al **overhead de comunicaciones**, mientras mayor cantidad de procesos tengan que comunicarse, tendremos mayor overhead, esto se refleja en el siguiente gráfico.



La ejecución con 16 hilos cuenta con 2 procesos, mientras que la de 32 hilos cuenta con 4 procesos.

5. Comparativa de rendimiento MPI y Pthreads

	MPI 1 nodo - 8 procesos				**Pthreads**		
N	Tiempo paralelo	Tiempo de comunicación	Speedup	Eficiencia	Tiempo paralelo	Speedup	Eficiencia
512	0,128058	0,034798	5,598072748	0,699759094	0,104325	6,871583992	0,858947999
1024	0,850408	0,117840	6,720824592	0,840103074	0,774455	7,379954936	0,922494367
2048	6,316809	0,547564	7,220724894	0,902590612	5,940648	7,677940184	0,959742523
4096	50,303595	1,894130	7,401393837	0,92517423	46,772974	7,960081414	0,995010177

P = 8

Si observamos los valores de la tabla, podremos notar rápidamente que el algoritmo en Pthreads es mejor desde el punto de vista de rendimiento.
La comunicación que tenemos en memoria distribuida es el pasaje de mensajes. Utilizando esta técnica se introduce mayor ocio por el retardo que producen las comunicaciones. Al tener mayor ocio se ven afectados los valores de **eficiencia** y de **speedup**. En *Pthreads*, tenemos un esquema de memoria compartida, por lo que los retardos de comunicación son mucho menos significativos.

6. Comparativa de rendimiento MPI y MPI híbrido

	2 nodos - 8 procesos por nodo				**2 nodos - 1 proceso por nodo - 8 hilos por nodo**			
N	Tiempo paralelo	Tiempo de comunicación	Speedup	Eficiencia	Tiempo paralelo	Tiempo de comunicación	Speedup	Eficiencia
512	0,242138	0,192463	2,960617499	0,185038594	0,201127	0,105947	3,564305141	0,222769071
1024	0,896835	0,530297	6,3729036	0,398306475	0,772551	0,398698	7,398143294	0,462383956
2048	4,707953	1,796485	9,688274288	0,605517143	4,419554	1,519571	10,32048482 7	0,645030302
4096	30,766454	6,717290	12,101384222	0,756336514	29,186437	5,463878	12,756496485	0,79728103

P = 16

	4 nodos - 8 procesos por nodo				**4 nodos - 1 proceso por nodo - 8 hilos por nodo**			
N	Tiempo paralelo	Tiempo de comunicación	Speedup	Eficiencia	Tiempo paralelo	Tiempo de comunicación	Speedup	Eficiencia
512	0,394440	0,367058	1,817457661	0,056795552	0,245282	0,153651	2,922668602	0,091333394
1024	0,900692	0,704938	6,345613151	0,198300411	0,874785	0,507116	6,533540241	0,204173133
2048	3,669880	2,213222	12,428727915	0,388397747	3,390706	1,921215	13,45204804	0,420376501
4096	20,436369	8,397362	18,21833815	0,569323067	19,457804	7,591907	19,134568372	0,597955262

P = 32

Con el algoritmo híbrido se consiguieron menores tiempos totales de ejecución. Este permite explotar distintos niveles de paralelismo lo que lo hace más eficiente que la solución que utiliza solo MPI.

Además se puede ver en las tablas que los tiempos de comunicación son menores. El motivo de este fenómeno es que en el híbrido encontramos una menor cantidad de procesos que se comunican, esto reduce el ocio y aumenta el cómputo (viéndose reflejado en el valor obtenido de eficiencia).