# Contenido

# Punto 1

## Ejercicio 2 – Practica 4

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?

| P = 4 | Retorno de control | |
|---|---|---|
| | Blocking.c | non-blocking.c |
| Proceso 1 | 2.000097 | 0.000039 |
| Proceso 2 | 4.000079 | 2.000057 |
| Proceso 3 | 6.000039 | 4.000131 |

| P = 8 | Retorno de control | |
|---|---|---|
| | Blocking.c | non-blocking.c |
| Proceso 1 | 2.000099 | 0.000040 |
| Proceso 2 | 4.000245 | 2.000115 |
| Proceso 3 | 6.000305 | 4.000262 |
| Proceso 4 | 8.010901 | 6.000233 |
| Proceso 5 | 10.011212 | 8.001225 |
| Proceso 6 | 12.010685 | 10.001064 |
| Proceso 7 | 14.010673 | 12.001012 |

| P = 16 | Retorno de control | |
|---|---|---|
| | Blocking.c | non-blocking.c |
| Proceso 1 | 2.000077 | 0.000050 |
| Proceso 2 | 4.000142 | 2.000109 |
| Proceso 3 | 6.000235 | 4.000269 |
| Proceso 4 | 8.010934 | 6.000223 |
| Proceso 5 | 10.010616 | 8.001284 |
| Proceso 6 | 12.010681 | 10.001075 |
| Proceso 7 | 14.011132 | 12.001069 |
| Proceso 8 | 16.010791 | 14.001053 |
| Proceso 9 | 18.010787 | 16.001100 |
| Proceso 10 | 20.011117 | 18.001089 |
| Proceso 11 | 22.010629 | 20.001268 |
| Proceso 12 | 24.010615 | 22.001080 |
| Proceso 13 | 26.010760 | 24.001045 |
| Proceso 14 | 28.010601 | 26.000994 |
| Proceso 15 | 30.010625 | 28.001108 |

El código que usa comunicaciones no bloqueantes retorna antes el control. En este ejemplo, los workers son más lentos que el master, por lo que en la comunicación bloqueante el master se debe quedar esperando a que el worker genere el mensaje y se lo envíe, en cambio, la comunicación no bloqueante no espera a que el worker envíe el mensaje para devolver el control, por lo que el master puede continuar con su ejecución y el control se retorna mucho más rápido, aunque el tiempo final de las operaciones termine siendo similar.

Si se elimina la operación MPI_Wait, el proceso master va a continuar ejecutando a pesar de no asegurar la recepción del mensaje del worker. Tenga o no el mensaje, va a realizar el printf que le sigue al Wait (línea 54), por lo que si la recepción todavía no se completó, es decir, no se escribió el buffer, no va a imprimirse el mensaje correcto, va a imprimir lo que haya en el buffer en ese momento.

En el código de ejemplo, al retirar la línea, se imprime "No debería estar leyendo esta frase.", que es el contenido que tenía el buffer por default, lo que indica que el pasaje de mensajes entre los procesos no se realizó.

## Ejercicio 3 – Practica 4

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 | N | Blocking-ring.c | Non-blocking-ring-c |
|---|---|---|---|
| P = 4 | N = 10000000 | 0.247760 | 0.221724 |
| | N = 20000000 | 0.494233 | 0.440828 |
| | N = 40000000 | 0.981933 | 0.882902 |
| P = 8 | N = 10000000 | 1.743007 | 0.897193 |
| | N = 20000000 | 3.470623 | 1.776804 |
| | N = 40000000 | 6.917952 | 3.467731 |
| P = 16 | N = 10000000 | 3.478514 | 0.815676 |
| | N = 20000000 | 6.930753 | 1.687486 |
| | N = 40000000 | 13.827173 | 3.439898 |

El que requiere menos tiempo de comunicación es el non-blocking-ring.

En este ejemplo de programa, los datos a enviar de un proceso a su próximo no dependen de los datos a recibir de su proceso anterior. Cuando se hace el receive y posteriormente el send, pasa lo siguiente:

Con comunicaciones bloqueantes, primero se espera a que la transferencia del receive se termine, es decir, cada proceso antes de enviar sus datos tiene que esperar a recibir los datos del proceso anterior, por lo que esa espera es una pérdida de tiempo en comunicación.

Con comunicaciones no bloqueantes, cuando se hace el receive, se prepara el buffer para recibir los datos, y se retorna el control al proceso antes de que la transferencia sea efectiva, por lo que el proceso comienza a realizar el send. Esto hace que cada proceso no tenga que esperar a recibir la información de su proceso anterior antes de comenzar a enviar la propia.

# Punto 2

## Algoritmo paralelo empleado MPI

### Explicación de la solución

Vamos directamente con el desarrollo, empezamos con la reserva de memoria.
Para las matrices r1, r2, r1a y r2b, cada proceso aloca una sección o strip de la matriz (stripSize*n), ya que no necesitan utilizar la matriz completa (n*n).
Lo mismo ocurre con las matrices c, t y m, con la diferencia que el proceso coordinador (en este caso el de rank = 0), va a alocar el tamaño completo. En el caso de t y m las aloca completas ya que es el encargado de inicializar cada uno de sus valores, y luego serán repartidas entre los procesos (por esta razón los demás procesos solo alocan un strip). En el caso de c, se aloca completa ya que es el encargado de "recolectar" los strips de los demás procesos cuando el programa termine, es quien va a tener la matriz c completa con los valores del resultado.
Para el caso de la matriz a y b, se van a alocar de manera completa en todos los procesos, ya que cada proceso las utiliza para la multiplicación de matrices, y como son el segundo operando de la multiplicación (r1*a y r2*b) necesitamos recorrer todas sus columnas.

Luego, el coordinador inicializa los datos, y los procesos se encuentran en una barrera antes de comenzar con la ejecución.

El primer paso en la ejecución es la distribución de los datos inicializados de las matrices m, t, a y b.
m y t se distribuyen mediante un scatter, ya que el coordinador tiene la matriz completa y reparte un strip a cada proceso.
a y b se distribuyen mediante broadcast, ya que es necesario que todos tengan la matriz completa.

Se continúa con la inicialización de las matrices r1 y r2, en donde cada proceso se va a encargar de calcular los valores de su strip, y para eso dispone de los strips necesarios de las matrices involucradas, t y m.
Además, cada proceso suma los valores calculados, lo que resulta en que cada proceso tendrá la sumatoria de los valores de su strip de r1 y r2.

Para obtener el valor del promedio de r1 y de r2, utilizado posteriormente para el cálculo de c, se utiliza la herramienta de comunicación Allreduce. De esta manera, cada proceso aportará la sumatoria de su strip, y como resultado todos los procesos obtienen el resultado de la sumatoria total de r1 y r2.

Decidimos que el valor de la sumatoria total sea comunicado a cada proceso para que cada proceso calcule el promedio por su cuenta, si bien el valor será el mismo en todos los procesos, es más eficiente que cada uno realice los cálculos a que el proceso coordinador tenga que hacer los cálculos y realizar otra comunicación para comunicarles el valor resultante a los demás.

Luego de esto, se realizan las multiplicaciones de r1 con a y r2 con b.
Para este punto, cada proceso dispone de su strip de r1 o r2 (calculado anteriormente) y con las matrices completas de a y b, por lo que tiene todos los valores necesarios para calcular su strip en r1a o r2b.
El algoritmo de multiplicación será el mismo que en entregas anteriores, pero en este caso solo se recorre una fila, el strip que le corresponde a cada proceso.

Finalizada la multiplicación, solo queda el cálculo de c.
Como en ocasiones anteriores, cada proceso será el encargado de calcular su strip. Para esto debe tener los valores de r1a y r2b (calculados recién), el promedio de r1 y r2 (el cual cada proceso lo calcula, por lo que lo tienen) y el strip de la matriz t que fue distribuido al comienzo.

Luego del cálculo de c, todos los strips son unificados en la matriz resultante. Esta acción se realiza mediante la comunicación gather, y el proceso que la recibe es el coordinador, quien reservó suficiente espacio para contenerla.

A lo largo del programa se utilizó un vector, en donde cada proceso marcó sus tiempos cada vez que realizaba una acción de comunicación o computación. Al finalizar el programa, mediante un reduce, con la función min y max, estos valores son comunicados al coordinador, el cual realiza las operaciones necesarias para calcular el tiempo total de ejecución y el tiempo de comunicación del programa.

## Análisis de rendimiento

Tiempos de ejecución y comunicación:

| n | P = 8 (1 nodo) | | P = 16 (2 nodos) | | P = 32 (4 nodos) | |
|---|---|---|---|---|---|---|
| | Tiempo de ejecución | Tiempo de comunicación | Tiempo de ejecución | Tiempo de comunicación | Tiempo de ejecución | Tiempo de comunicación |
| **512** | 0.102289 | 0.030445 | 0.110702 | 0.078513 | 0.119856 | 0.113892 |
| **1024** | 0.646052 | 0.085031 | 0.600887 | 0.332304 | 0.482665 | 0.373728 |
| **2048** | 4.789172 | 0.400957 | 3.371191 | 1.190826 | 2.525798 | 1.462144 |
| **4096** | 36.636862 | 1.433410 | 22.251770 | 4.632734 | 14.724527 | 5.950580 |

*nota: el bs es el mismo que en las demás entregas, 64, sin embargo, en algunos casos se adaptó por la cantidad de procesos: cuando P = 32 y n = 512, bs = 16, cuando P = 32 y n = 1024, bs = 32 y cuando P = 16 y n = 512, bs = 32.

SpeedUp:

| n | 8 | 16 | 32 |
|---|---|---|---|
| **512** | 5.49321041 | 5.07574389 | 4.68808403 |
| **1024** | 6.73693139 | 7.24330531 | 9.01745103 |
| **2048** | 7.27379702 | 10.3332813 | 13.791865 |
| **4096** | 7.34493126 | 12.0932058 | 18.2753058 |

Eficiencia:

| n | 8 | 16 | 32 |
|---|---|---|---|
| **512** | 0.6866513 | 0.31723399 | 0.14650263 |
| **1024** | 0.84211642 | 0.45270658 | 0.28179534 |
| **2048** | 0.90922463 | 0.64583008 | 0.43099578 |
| **4096** | 0.91811641 | 0.75582536 | 0.5711033 |

Overhead de comunicación:

| n | 8 | 16 | 32 |
|---|---|---|---|
| **512** | 29.7637087 | 70.9228379 | 95.0240288 |
| **1024** | 13.161634 | 55.3022448 | 77.4301016 |
| **2048** | 8.37215702 | 35.3235993 | 57.888398 |
| **4096** | 3.91248028 | 20.8196202 | 40.4127073 |

## Conclusión del análisis de rendimiento individual

En cuanto a la solución con solo MPI, los resultados no son muy alentadores. El speedup de la solución y, por consecuente, su eficiencia, son relativamente bajas. Si bien no es muy notorio con P = 8, podemos apreciar que con P = 16 los valores son bajos. Con un n = 512 la eficiencia es solamente de un 31%, recién con el tamaño de matriz de 4096 el valor puede ser más aceptable, llegando al 75% de eficiencia. Pero, como es de esperarse, esta situación se ve agravada con P = 32, donde incluso con el mayor tamaño de matriz, la eficiencia no llega al 60%, y con una matriz chica llega al 14%.

Este 14% de eficiencia al incrementar la cantidad de procesos, nos indica que el algoritmo presenta una escalabilidad débil. Se podría decir utilizar 32 procesos para una matriz de tamaño 4096 no es muy eficiente tampoco, porque no llega ni a 60% de eficiencia, pero depende de cada situación evaluar si vale la pena incrementar la cantidad de procesadores, aunque no lo haga tan bien su eficiencia.

Por último, vamos a comparar la eficiencia de esta implementación y la de la entrega anterior, hecha con Pthreads. Vamos a utilizar 8 hilos en el caso de Pthreads (1 solo proceso) y 8 procesos con MPI.

| n | Versión Pthreads | Versión MPI |
|---|---|---|
| **512** | 0.927134 | 0.686651 |
| **1024** | 0.940305 | 0.842116 |
| **2048** | 0.963031 | 0.909225 |
| **4096** | 0.964978 | 0.918116 |

Como se puede apreciar, la versión Pthreads tiene una mayor eficiencia. Por ejemplo, con la matriz más pequeña, la diferencia de eficiencia llega a ser casi de un 25%.

Esta diferencia se debe principalmente a que la comunicación realizada por Pthreads, al ser mediante memoria compartida, es más veloz que la comunicación por mensajes de MPI.

A medida que escala el tamaño de n, la diferencia en la eficiencia de ambas soluciones se reduce a un 5%, ya que el overhead de comunicación disminuye, y las comunicaciones no tienen tanto impacto en el tiempo total de resolución, evidenciando que la diferencia en la eficiencia de cada solución reside en la capacidad de comunicación de cada tecnología.

## Algoritmo paralelo empleando MPI+OpenMP

### Explicación de la solución

En este caso el código es muy parecido, pero agregando las herramientas de OMP.
Por esta razón, antes de comenzar con la ejecución, se realiza la creación de hilos.

En este caso, las comunicaciones están precedidas de la sentencia omp single, ya que solo requerimos que la comunicación sea entre procesos, por lo que alcanza con un solo hilo encargándose de la comunicación.

Una vez finalizada la comunicación, cada proceso dispone de su strip, y se comienza con la inicialización de r1 y r2.
Para esto, el proceso divide su strip en varios segmentos, y cada hilo recibirá uno para comenzar la realización de la tarea.
De esta manera, una vez se termine la sentencia quedará calculado el strip del proceso, y el valor de la sumatoria también será correcto, ya que se utiliza la herramienta reduction y la operación de suma, para que se unan las sumatorias que cada hilo calculó en su segmento.

Uno de los hilos va a ser el encargado de comunicar (como lo hace el coordinator en el código MPI), representando su proceso, el valor de la sumatoria de su strip mediante un Allreduce, y posteriormente también de calcular el promedio de r1 y r2.

Luego, cada hilo realiza la multiplicación de matrices de r1*a y r2*b, donde se encarga exclusivamente del segmento que le asigna omp del strip que tiene el proceso. Y con esos resultados, hace lo mismo en el cálculo de c.

Para finalizar, uno de los hilos se encarga de comunicarse con los demás procesos para unir los strips de c y almacenar la matriz resultante en el proceso coordinator.

Como aclaración nos gustaría indicar que como la distribución de carga es equitativa, utilizamos el schedule static para omp, no hay diferencia entre el poder de procesamiento de cada hilo ni en los datos asignados entre ellos.

### Análisis de rendimiento

Tiempo de ejecución y comunicación:

| n | P = 16 (2 nodo) | | P = 32 (4 nodos) | |
|---|---|---|---|---|
| | Tiempo de ejecución | Tiempo de comunicación | Tiempo de ejecución | Tiempo de comunicación |
| **512** | 0.115139 | 0.074477 | 0.115868 | 0.095737 |
| **1024** | 0.577998 | 0.285436 | 0.500231 | 0.351015 |
| **2048** | 3.370430 | 1.117912 | 2.488862 | 1.387795 |
| **4096** | 21.894036 | 4.470355 | 14.136399 | 5.454058 |

*nota: el bs es el mismo que en las demás entregas, 64, sin embargo, en algunos casos se adaptó por la cantidad de procesos: cuando P = 32 y n = 512, bs = 16, cuando P = 32 y n = 1024, bs = 32 y cuando P = 16 y n = 512, bs = 32.

Speedup:

| n | 16 | 32 |
|---|---|---|
| **512** | 4.88014487 | 4.84944074 |
| **1024** | 7.5301437 | 8.70079623 |
| **2048** | 10.3356144 | 13.9965434 |
| **4096** | 12.2908007 | 19.035628 |

Eficiencia:

| n | 16 | 32 |
|---|---|---|
| **512** | 0.30500905 | 0.15154502 |
| **1024** | 0.47063398 | 0.27189988 |
| **2048** | 0.6459759 | 0.43739198 |
| **4096** | 0.76817504 | 0.59486338 |

Overhead de comunicación:

| n | 16 | 32 |
|---|---|---|
| **512** | 64.6844249 | 82.6259191 |
| **1024** | 49.3835619 | 70.1705812 |
| **2048** | 33.1682308 | 55.7602229 |
| **4096** | 20.4181404 | 38.5816643 |

### Conclusión del análisis de rendimiento individual

El rendimiento de esta solución es muy similar al de MPI. Los valores de optimización no son muy altos. Para valores chicos de n, la eficiencia de la solución es muy pobre, en especial en P = 32, como es de esperarse. Sin embargo, a medida que el tamaño de n incrementa, la eficiencia mejora notablemente, en P = 16 se pasa el 75% por ejemplo.

Esta situación evidencia que la solución tiene una escalabilidad débil, si no se aumenta el tamaño del problema la eficiencia decae enormemente.

Para el mayor tamaño del problema, con P = 32, la eficiencia es del 60% aproximadamente, quedará en manos de quien implemente el sistema decidir si vale la pena la mejora de tiempo a pesar de la caída de la eficiencia. Otra opción conveniente, si se desea utilizar esta cantidad de procesos, sería seguir aumentando el tamaño del problema.

## Comparación de ambos modelos

### Análisis de eficiencia comparativo (con P = 16 y P = 32)

Como se puede observar en los resultados plasmados anteriormente, el rendimiento de ambas soluciones es bastante parecido, ambas incrementan el tiempo de ejecución ampliamente respecto a las soluciones secuenciales, y ambas soluciones sufren el problema de escalabilidad débil.

Si bien ambas soluciones utilizan paralelismo, la solución de MPI+OMP es más eficiente. Esto se debe a que MPI va a tener que realizar todas sus comunicaciones mediante pasaje de mensajes, mientras que MPI+OMP va a aprovechar la arquitectura del clúster, haciendo uso de la memoria compartida en algunas comunicaciones, y dejando menos resueltas por pasaje de mensajes. Por ejemplo, en el caso de P = 32, que es la situación más notoria, MPI va a tener que comunicar por pasaje de mensajes a 32 procesos, mientras que MPI+OMP va a comunicar solo 4 procesos por pasaje de mensajes, y cada proceso comunica sus 8 hilos por memoria compartida.

Esta situación de ventaja en el tiempo de comunicación se puede apreciar observando los overhead de comunicación de cada solución, en donde se aprecia que es menor en el caso de MPI+OMP, ya que las resuelve más rápido. Generalmente, la diferencia entre el tiempo de ejecución de MPI+OMP y MPI es esta diferencia en el tiempo de comunicación.

Si bien los tiempos de MPI+OMP son mejores, la mejora no es muy grande, la mayor es con P = 32 y n = 4096, que es el procesamiento más pesado, en donde el tiempo de ejecución llega a ser medio segundo más rápido. Asumimos que esta situación se da porque la cantidad de mensajes no es tan grande, pero la diferencia sería mucho más notoria en un contexto con mayor cantidad de datos y comunicación.

Se puede concluir entonces, que a la hora de resolver un problema, hay que tener en cuenta varias cosas, como pueden ser la arquitectura en donde se ejecuta, ya que si hay que comunicar nodos separados, vamos a tener que usar MPI, pero siempre que se puede utilizar memoria compartida podemos aprovechar OMP, por eso la solución híbrida fue la más eficiente. También hay que tener en cuenta el balance entre tamaño de problema y cantidad de procesadores, analizando el overhead de comunicación, que si nos pasamos con los procesadores puede ser alto en soluciones a problemas pequeños, y, de la mano, la escalabilidad de la solución.

Si se desea realizar la validación de los resultados en ambas soluciones, ir al anexo 1.

# ANEXO 1

En cuanto a la validación del programa, para no hacerlo muy engorroso, se puede optar por inicializar las matrices con el valor del índice i (en lugar de con un valor random entre 1 y 100).

Para simplificar la comprobación de los resultados, mostramos en pantalla el primer valor de las últimas 15 filas del algoritmo con n = 512:

c[497][0] = 3309427128499.299316

c[498][0] = -485003495395.333435

c[499][0] = -11519389993707.042969

c[500][0] = -19672809425427.105469

c[501][0] = -17437772656851.357422

c[502][0] = -6847102439133.143555

c[503][0] = 2360589895623.279785

c[504][0] = 1681845022838.849365

c[505][0] = -8312875462389.689453

c[506][0] = -18468576679652.671875

c[507][0] = -19445597448209.148438

c[508][0] = -10322964900057.054688

c[509][0] = 519829989596.999756

c[510][0] = 3085549463769.812500

c[511][0] = -5037408472615.143555