
Universidad Nacional de La Plata

Sistemas Paralelos

Programación en pasaje de mensajes / Programación híbrida

ENTREGA 3 - TRABAJO FINAL

***Grupo 8***

***Alumnos:***
Castro, Lautaro Germán
Torrico, Fabrizio Benjamín


Facultad de Informática

Año 2024

# Índice

# 1. Ejercicios de entrega

## 1.1. Ejercicio 2

**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?**

Como puede observarse en Figura 2 y Figura 1, la versión no-bloqueante retorna antes el control al proceso (para el resto de casos sucede lo mismo). Esto se debe a que las operaciones bloqueantes no liberan al proceso hasta que la operación haya sido completada, en este caso entonces MPI_Recv() va a retornar el control cuando esta se complete y el mensaje haya sido recibido, mientras que las operaciones no bloqueantes retornan el control de forma inmediata, aunque la operación no haya sido completada, por lo cual en MPI_IRecv() aunque el mensaje no se recibió , se retorna el control inmediatamente.

```
Tiempo transcurrido 0.000000 (s):        proceso 0, llamando a MPI_Recv()
↪  [bloqueante] (fuente rank 1)
Tiempo transcurrido 2.000088 (s):        proceso 0, MPI_Recv() devolvio control con
↪  mensaje: Hola Mundo! Soy el proceso 1
Tiempo transcurrido 2.000099 (s):        proceso 0, llamando a MPI_Recv()
↪  [bloqueante] (fuente rank 2)
Tiempo transcurrido 4.000147 (s):        proceso 0, MPI_Recv() devolvio control con
↪  mensaje: Hola Mundo! Soy el proceso 2
Tiempo transcurrido 4.000159 (s):        proceso 0, llamando a MPI_Recv()
↪  [bloqueante] (fuente rank 3)
Tiempo transcurrido 6.000176 (s):        proceso 0, MPI_Recv() devolvio control con
↪  mensaje: Hola Mundo! Soy el proceso 3

Tiempo total = 0.000000 (s)
```

Figura 1: blocking con P = 4

```
Tiempo transcurrido 0.000000 (s):        proceso 0, llamando a MPI_IRecv() [no bloqueante]
↪  (fuente rank 1)
Tiempo transcurrido 0.000017 (s):        proceso 0, MPI_IRecv() devolvio el control..
                                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 2.000176 (s):        proceso 0, operacion receive completa con
↪  mensaje: Hola Mundo! Soy el proceso 1
Tiempo transcurrido 2.000185 (s):        proceso 0, llamando a MPI_IRecv() [no bloqueante]
↪  (fuente rank 2)
Tiempo transcurrido 2.000189 (s):        proceso 0, MPI_IRecv() devolvio el control..
                                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 4.000110 (s):        proceso 0, operacion receive completa con
↪  mensaje: Hola Mundo! Soy el proceso 2
Tiempo transcurrido 4.000121 (s):        proceso 0, llamando a MPI_IRecv() [no bloqueante]
↪  (fuente rank 3)
Tiempo transcurrido 4.000126 (s):        proceso 0, MPI_IRecv() devolvio el control..
                                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 6.000072 (s):        proceso 0, operacion receive completa con
↪  mensaje: Hola Mundo! Soy el proceso 3

Tiempo total = 0.000000 (s)
```

Figura 2: Non-blocking con P = 4

**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?**

No, los valores no se imprimen correctamente. Esto sucede porque cuando se utiliza comunicación no bloqueante, la semántica de la operación no se garantiza, es decir, que aunque se retorna el control esta puede no haberse completado, entonces cuando se utilizan este tipo de comunicaciones, el programador debe asegurarse de que la operación se haya completado, la operación MPI_Wait() es una operación que permite validar el cumplimiento de una operación, en este caso, esta operación bloquea al proceso hasta que la operación que se indica en MPI_WAIT sea completada, entonces al remover MPI_Wait(), no se garantiza que la operación se haya completado a partir de ese punto, y se imprimen mensajes incorrectos. Puede visualizarse la ejecución sin la línea 52 en Figura 3

```
Tiempo transcurrido 0.000000 (s):        proceso 0, llamando a MPI_IRecv() [no bloqueante]
↪  (fuente rank 1)
Tiempo transcurrido 0.000016 (s):        proceso 0, MPI_IRecv() devolvio el control..
                                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 0.000021 (s):        proceso 0, operacion receive completa con
↪  mensaje: No deberia estar leyendo esta frase.
Tiempo transcurrido 0.000023 (s):        proceso 0, llamando a MPI_IRecv() [no bloqueante]
↪  (fuente rank 2)
Tiempo transcurrido 0.000026 (s):        proceso 0, MPI_IRecv() devolvio el control..
                                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 0.000030 (s):        proceso 0, operacion receive completa con
↪  mensaje: No deberia estar leyendo esta frase.
Tiempo transcurrido 0.000033 (s):        proceso 0, llamando a MPI_IRecv() [no bloqueante]
↪  (fuente rank 3)
Tiempo transcurrido 0.000035 (s):        proceso 0, MPI_IRecv() devolvio el control..
                                        ..pero el mensaje no fue aun recibido..
Tiempo transcurrido 0.000039 (s):        proceso 0, operacion receive completa con
↪  mensaje: No deberia estar leyendo esta frase.

Tiempo total = 0.000000 (s)
```

Figura 3: Non-blocking sin MPI_Wait

## 1.2. 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?**

Como puede observarse en Tabla 1 y Tabla 2 la implementación con pasaje de mensajes **no bloqueante** es la más óptima. Esto sucede porque la comunicación bloqueante para realizar una comunicación en anillo, impone una secuencialidad en los procesos para evitar deadlock, mientras que en la solución no bloqueante esta secuencialidad desaparece (debido al no bloqueo del send, que en pasaje bloqueante debe esperar el correspondiente receive para continuar) explotando de mejor manera el paralelismo.

*Nota: Como el equipo utilizado solo dispone de 4 nucleos, las ejecuciones con 8 o 16 demoran más por la sobrecarga de procesos.*

| N | 4 | 8 | 16 |
|---|---|---|---|
| 10.000.000 | 0.11 | 0.25 | 0.88 |
| 20.000.000 | 0.20 | 0.52 | 2.09 |
| 40.000.000 | 0.47 | 1.03 | 5.39 |

Cuadro 1: Tiempo de comunicación con operación bloqueante

| N | 4 | 8 | 16 |
|---|---|---|---|
| 10.000.000 | 0.04 | 0.07 | 0.10 |
| 20.000.000 | 0.09 | 0.13 | 0.24 |
| 40.000.000 | 0.17 | 0.34 | 0.55 |

Cuadro 2: Tiempo de comunicación con operación no-bloqueante

# 2. Introducción

Este trabajo práctico tiene como objetivo el desarrollo de dos algoritmos paralelos para calcular la expresion

$$R = \frac{maxA * maxB - minA * minB}{PromA * PromB} \times [A \times B] + [C \times D]$$

Los algoritmos desarrollados son:

1. Algoritmo paralelo utilizando MPI
2. Algoritmo paralelo hibrido utilizando MPI+OpenMp

Se mide el rendimiento de los algoritmos variando el tamaño del problema y la cantidad de núcleos utilizados. Además, se compara el rendimiento de los algoritmos paralelos con los resultados obtenidos en el trabajo práctico anterior.

# 3. Detalles de la experimentación

## 3.1. Hardware utilizado

Este trabajo se realizó sobre la **Partición Blade (CLuster Multicore)** del cluster proporcionado por la cátedra. El equipo cuenta con las siguientes características:

- Cluster conformado por 16 nodos.
- Por nodo 8GB de RAM y 2 procesadores Intel Xeon con 4 cores que operan a 2.0GHz.
- caché L1 con tamaño de 32KB para datos y 32KB para instrucciones.

Debido a que para las experimentaciones son necesarios múltiples nodos y hasta 8 cores, la partición es adecuada para la experimentación.

### 3.2. Pruebas realizadas

Las pruebas se realizaron considerando las siguiente variaciones:

- Tamaño de problema: N = { 512, 1024, 2048, 4096 }
- Cantidad de núcleos para MPI; P = { 8, 16, 32 } (equivalente a 1, 2 y 4 nodos)
- Cantidad de núcleos para MPI+OpenMP: P = { 16, 32 } (equivalente a 2 y 4 nodos)

## 4. Soluciones planteadas

### 4.1. Optimizaciones generales

Dado que todos los algoritmos fueron escritos a partir de la versión secuencial implementada en el primer trabajo, en todos se encontrarán las siguientes optimizaciones:

- **Localidad de los datos**
  - Las matrices fueron organizadas internamente por filas o columnas de acuerdo a como debe procesarse respecto a la operación aplicada. También se utilizó un algoritmo de multiplicación por bloques para el producto de matrices. Haciendo un uso óptimo de la localidad espacial.
  - Cada recorrido de matriz fue aprovechado incluyendo la mayor cantidad de operatoria posible dentro del recorrido (localidad temporal). Por ejemplo el mínimo, máximo y promedio se obtienen en un único recorrido por cada matriz.
- **Optimización de caché:** Para el tamaño de bloque en la multiplicación se buscó un número donde el bloque entrara en la caché L1 del hardware utilizado. Para ello se utilizó el siguiente cálculo:

$$2 * (\text{tamañoDeBloque})^2 * \text{tamañoPalabra} = \text{tamañoDeCache}$$

  Siendo que tamaño de palabra es 8 bytes porque se utilizan matrices con datos de tipo double, y el tamaño de caché L1 es 32KB, el cálculo da un resultado de 45 aproximadamente. Por lo cual lo indicado sería un tamaño que no exceda ese tamaño, por ejemplo: 32. De igual manera, se realizaron pruebas para la elección del tamaño ideal en la Tabla 5.2, denotaron que el tamaño óptimo es de 64.
- **Expresiones de cómputo repetido:** Se evitaron utilizar expresiones de cómputo que se repitan en cada iteración de un loop. Por ejemplo, expresiones como i*N o j*N para acceder a una fila o columna de una matriz para una operatoria, son realizadas de manera previa a entrar al loop que las utiliza.
- **Optimización del compilador:** El compilador gcc ofrece varios niveles de optimización. Todos los algoritmos en esta ocasión fueron compilados con un nivel -O3.

### 4.2. Algoritmos Paralelo con MPI

#### 4.2.1. Idea general

Como se nombró anteriormente, este trabajo presenta una operatoria con matrices, donde se puede articular muy facilmente una **descomposición de datos**, donde cada proceso realiza el mismo cómputo sobre un subconjunto de datos y se sincronizan las operaciones necesarias.

#### 4.2.2. Implementacion

**1)** El proceso coordinador inicializa las matrices A, B, C, D y R con valores específicos para pruebas.

**2)** Las matrices A y C se dividen en bloques y se distribuyen entre los procesos mediante `MPI_Scatter`.

**3)** Las matrices BBB y DDD se transmiten completamente a todos los procesos utilizando `MPI_Bcast`.

**4)** Cada proceso calcula el máximo, mínimo y promedio de su porción de la matriz A y de su correspondiente segmento de B. Luego se globaliza cada valor

**5)** Se realizan las operaciones de matrices. A x B, multiplicacion del factor y C x D

**6)** Los resultados parciales se recolectan en el proceso coordinador utilizando `MPI_Gather`. y se verifican

#### 4.2.3. Decisiones de diseño

- `MPI_Scatter` se utiliza para distribuir las filas de las matrices A y C entre los procesos.
- `MPI_Bcast` se utiliza para transmitir las matrices completas B y D a todos los procesos, dado que estas son necesarias para la multiplicación en cada proceso.
- Se utilizan barreras de sincronización (`MPI_Barrier`) antes de medir los tiempos de comunicación para asegurar que todos los procesos estén listos.
- Los tiempos de comunicación y ejecución se registran y se reducen utilizando `MPI_Reduce` o `MPI_Allreduce` para obtener los tiempos mínimos y máximos.

### 4.3. Algoritmo Paralelo Híbrido con MPI+OpenMP

#### 4.3.1. Idea general

Debido a que este algoritmo se realizó a partir del algoritmo MPI puro (Subsección 4.2), presenta la misma idea general, es decir un modelo **SPMD (Single Program Multiple Data)** donde cada proceso realiza el mismo cómputo en una porción diferente de datos, pudiendo diferir algún cómputo o instrucción a partir de condicionales.

#### 4.3.2. Implementacion

Respecto a la implementación debido a las facilidades de OpenMP para paralelizar código secuencial, la implementación simplemente consistió en articular las directivas de openMP en aquellas secciones del código donde podía explotarse el paralelismo.

#### 4.3.3. Decisiones de diseño

- Se utilizo un nivel de soporte **MPI_THREAD_SERIALIZED (Nivel 2)**, donde los procesos pueden ser multi-hilados y los diferentes hilos pueden ejecutar rutinas MPI pero sólo una a la vez; los llamados a MPI no pueden ser realizados en simultáneo por 2 hilos.
- Sólo se realiza una única vez la creación de hilos, con el objetivo de evitar la sobrecarga que genera la creación/destrucción de hilos.
- Se utiliza una planificación static para evitar un desbalance en la carga de trabajo, dado que contamos con procesadores homogéneos.

# 5. Pruebas de tiempo

Para medir los tiempos debido a las posible desincronización de clocks entre nodos se aplicaron dos estrategias:

- **Tiempo total:** Se calcula como tiempo final - tiempo inicial. Ambos proporcionados por el hilo master, ya que este es el último en terminar (porque ejecuta un Scatter), y tomar su tiempo inicial es coherente porque antes de la medición se antepone una barrera.
- **Tiempo de comunicación**: Este se calcula como el promedio de las diferencias entre los tiempos maximos y minimos en zonas de comunicacion del programa.

## 5.1. Pruebas de tiempo (En segundos)

### 5.1.1. MPI

| MPI (Tiempo de ejecucion) | | | |
|---|---|---|---|
| N | 8 procesos | 16 procesos | 32 procesos |
| 512 | 0.17 | 0.14 | 0.15 |
| 1024 | 1.22 | 0.88 | 0.7 |
| 2048 | 9.36 | 5.66 | 3.76 |
| 4096 | 73.930 | 40.79 | 24.04 |

| MPI (Tiempo de comunicación) | | | |
|---|---|---|---|
| N | 8 procesos | 16 procesos | 32 procesos |
| 512 | 0.02 | 0.06 | 0.11 |
| 1024 | 0.07 | 0.29 | 0.38 |
| 2048 | 0.34 | 1.06 | 1.38 |
| 4096 | 1.83 | 4.48 | 5.51 |

| MPI (Speedup) | | | |
|---|---|---|---|
| N | 8 procesos | 16 procesos | 32 procesos |
| 512 | 3.84 | 4.41 | 4.19 |
| 1024 | 4.39 | 6.02 | 7.61 |
| 2048 | 4.68 | 7.73 | 11.63 |
| 4096 | 5.06 | 9.16 | 15.54 |

| MPI (Eficiencia) | | | |
|---|---|---|---|
| N | 8 procesos | 16 procesos | 32 procesos |
| 512 | 0.48 | 0.28 | 0.13 |
| 1024 | 0.55 | 0.38 | 0.24 |
| 2048 | 0.59 | 0.48 | 0.36 |
| 4096 | 0.63 | 0.57 | 0.49 |

| MPI (Overhead) | | | |
|---|---|---|---|
| N | 8 procesos | 16 procesos | 32 procesos |
| 512 | 16.47 % | 43.75 % | 70.88 % |
| 1024 | 6.52 % | 33.31 % | 54.61 % |
| 2048 | 3.67 % | 18.71 % | 36.65 % |
| 4096 | 2.48 % | 10.98 % | 22.91 % |

### 5.1.2. MPI+OpenMP

| MPI + OpenMP (Tiempo de ejecución) | | |
|---|---|---|
| N | 2 Nodos (16 hilos) | 4 Nodos (32 hilos) |
| 512 | 0.240048 | 0.255078 |
| 1024 | 0.948539 | 1.006955 |
| 2048 | 6.377802 | 3.962076 |
| 4096 | 46.455126 | 26.198110 |

| MPI + OpenMP (Tiempo de Comunicacion) | | |
|---|---|---|
| N | 2 Nodos (16 hilos) | 4 Nodos (32 hilos) |
| 512 | 0.071 | 0.084 |
| 1024 | 0.28 | 0.340 |
| 2048 | 1.15 | 1.336 |
| 4096 | 4.80 | 5.329 |

| MPI + OpenMP (Speedup) | | |
|---|---|---|
| N | 2 Nodos (16 hilos) | 4 Nodos (32 hilos) |
| 512 | 1.43 | 1.34 |
| 1024 | 2.95 | 2.78 |
| 2048 | 3.72 | 5.59 |
| 4096 | 4.65 | 8.25 |

| MPI + OpenMP (Eficiencia) | | |
|---|---|---|
| N | 2 Nodos (16 hilos) | 4 Nodos (32 hilos) |
| 512 | 0.08 | 0.04 |
| 1024 | 0.18 | 0.08 |
| 2048 | 0.23 | 0.17 |
| 4096 | 0.29 | 0.25 |

| MPI + OpenMP (Overhead) | | |
|---|---|---|
| N | 2 Nodos (16 hilos) | 4 Nodos (32 hilos) |
| 512 | 29.90 % | 33.15 % |
| 1024 | 30 % | 33.77 % |
| 2048 | 18.19 % | 33.73 % |
| 4096 | 10.33 % | 20.34 % |

### 5.2. Tamaño de bloque óptimo

| N | Secuencial (bloque 16) | Secuencial (bloque 32) | Secuencial (bloque 64) |
|---|---|---|---|
| | Tiempo | Tiempo | Tiempo |
| 512 | 0.6573 | 0.4185 | 0.3426 |
| 1024 | 5.3341 | 3.8149 | 2.8003 |
| 2048 | 43.8128 | 30.8074 | 23.7526 |
| 4096 | 373.6544 | 264.2447 | 216.0979 |

## 6. Conclusiones

### 6.1. MPI vs MPI+OpenMP

#### Rendimiento

En términos de rendimiento, se observaron resultados similares entre ambas implementaciones en la mayoría de los casos de prueba. Ambos enfoques muestran tiempos de ejecución comparables para tamaños de problema simliares sin embargo:

- **MPI**: El tiempo de ejecución disminuye con el aumento del número de nodos. Pero el overhead de comunicación es más alto y aumenta mas rapido en comparación con OpenMP, lo que indica una deficiencia en la comunicación entre nodos.
- **MPI+OpenMP**: Al igual que con MPI, se aprecia un buen rendimiento general pues con el aumento del número de nodos o hilos, el tiempo de ejecución disminuye. De igual manera, el aumento en el overhead de comunicación a medida que aumenta el número de nodos intuye un negativo rendimiento en tareas más grandes.

El speedup de OpenMP alcanza un speedup casi lineal, sin embargo a medida que se agregan mas nucleos, la eficiencia disminuye con el overhead y la sincronizacion. Por otro lado MPI, muestra un speedup que sigue aumentando con el incremento del numero de procesadores, aunque no linealmente por la sobrecarga de comunicacion

#### Escalabilidad

- **MPI**: La escalabilidad también parece ser buena, pero el overhead de comunicación es más alto en comparación con OpenMP. Esto podría limitar la escalabilidad en sistemas con un gran número de nodos o en tareas con una carga de comunicación significativa.
- **MPI+OpenMP**: La escalabilidad parece ser buena hasta cierto punto, ya que el tiempo de ejecución disminuye significativamente al aumentar el número de nodos o hilos. Sin embargo, el aumento en el overhead de comunicación a medida que se agregan más nodos podría limitar la escalabilidad en tareas más grandes.

OpenMP muestra una mejor eficiencia en la comunicacion en comparacion con MPI, lo que lo hace mas adecuado para sistemas con un numero limitado de nodos o hilos. Mientras que MPi, a pesar de tener un mayor overhead de comunicacion, puede escalar mejor en sistemas con mayor numero de nodos. Los tiempo resultaron muy similares por lo que la eleccion de OpenMP y MPI debe basarse en las caracteristicas especificas del sistema y la aplicacion, teniendo en cuento el numero de procesadores, requisitos de comunicacion, etc.

## 6.2. MPI vs OpenMP/Pthreads

### Rendimiento

En cuanto a rendimiento, los algoritmos hilos puros presentaron una amplia ventaja, y es que la comunicación por memoria compartida es más ágil que el paso de mensajes, además de que este se acrecenta a medida que el tamaño del problema aumenta. También esta diferencia de rendimiento se puede deber a que en el paso de mensajes, cuando la comunicación es entre nodos, los datos deben replicarse aumentando posiblemente los fallos de caché, cosa que es probable que se reduzca en memoria compartida porque los datos no deben replicarse.

### Escalabilidad

La ventaja de memoria compartida usando OpenMP o Pthreads radica en su velocidad y facilidad de uso en dominios de problema de un tamaño considerable, Si se necesita escalar a magnitudes mucho más altas nos encontramos con un límite en memoria compartida para seguir ampliando la capacidad de cómputo (hardware), en este caso MPI resulta ideal por el gran numero de procesadores que puede manejar de manera eficiente dando una solucion robusta para sistemas de memoria distribuida