Paralelismo - 23GIIN

Actividad 4 – Portafolio

Ejercicios de OpenMPI y OpenMP

Gagliardo Miguel Angel

30 de Enero de 2024

1) Con el siguente programa **piparallel.c**

- Revise y explique qué hace el programa (puede hacer diagramas o dibujos)
- Compile y ejecute
- Explique la utilidad de las primitivas de comunicación colectiva.

Explicacion corta

El programa utiliza la biblioteca MPI (Message Passing Interface) para calcular una aproximación del valor de **pi** utilizando el método del trapecio. Este método consiste en dividir el intervalo total en intervalos pequeños y aproximar la curva y = F(x) en los diversos intervalos pequeños mediante alguna curva más simple cuya integral puede calcularse utilizando solamente las ordenadas de los puntos extremos de los intervalos.

Explicación extensiva

El programa distribuye el trabajo entre varios procesos para mejorar el rendimiento en un entorno de computación paralela, importando la lib MPI, ademas de math y Standard I/O.

- Se define una constante **STEPS** con un valor de 5. Esta representa el número de veces que se realizará la estimación de **pi**.
- Se define una **función f(a)** que representa la función a integrar. En este caso y como comente antes, la función que utiliza en el método es la de trapecio para aproximar la integral.
- El proceso 0 o primer proceso, establece el valor de n (número de trapecios) y registra el tiempo de inicio (**startwtime**).

- Se utiliza una barrera de sincronización (**MPI_Barrier**) para asegurarse de que todos los procesos estén listos antes de comenzar el cálculo.
- El bucle principal realiza el cálculo de la integral mediante el método del trapecio distribuyendo el trabajo entre los procesos.
- Se utiliza la función **MPI_Reduce** que suma los resultados parciales obtenidos por cada procesador y asi obtener el resultado total.
- Nuevamente, el proceso 0 imprime los resultados finales y registra el tiempo de finalización.
- Para finalizar, se cierra MPI con **MPI_Finalize()** y se devuelve 0.

Compilacion y Ejecucion

Demostracion para 2 y para 4 procesadores:

```
[/tmp]
gattes$ mpicc piparallel.c -o piparallel

[/tmp]
gattes$ mpirun -np 2 piparallel
Process 0 running on lagias
Process 1 running on lagias
pi is approximately 3.1415926535899388, Error is 0.0000000000001457
wall clock time = 0.012489

[/tmp]
gattes$ mpirun -np 4 piparallel
Process 2 running on lagias
Process 1 running on lagias
Process 0 running on lagias
Process 3 running on lagias
pi is approximately 3.1415926535899028, Error is 0.0000000000001097
wall clock time = 0.008506
```

Primitivas de Comunicación Colectiva

Recordemos que son operaciones que implican la participación de todos los procesos en un comunicador MPI específico y son fundamentales en la programación paralela, ya que permiten la sincronización y la coordinación eficiente entre los procesos para realizar tareas colaborativas. Para el caso del programa **piparallel.c,** se utilizan las siguientes primitivas:

- **MPI_Init:** Inicializa el entorno MPI
- **MPI_Comm_size:**Determina el numero total de procesos en el comunicador MPI
- **MPI_Comm_rank:** Determina el rango del proceso que llama en el comunicador MPI
- **MPI_Get_processor_name:** Obtiene el nombre del procesador en el que se está ejecutando el proceso que llama
- **MPI_Bcast:** Transmite un mensaje desde el proceso con rango 0 (el primer proceso) a todos los demas procesos
- **MPI_Reduce:** Como se explica anteriormente, realizan una operacion de reduccion entre todos los procesos y devuelve un resultado. En este caso es la suma parcial de todos los procesos para obtener el valor final.
- **MPI_Wtime:** Devuelve el tiempo de reloj transcurrido, en segundos
- **MPI_Finalize:** Finaliza el entorno de MPI, permitiendo que los procesos realicen una limpieza de su entorno antes de terminar.

Por ultimo y para este caso, se utiliza el comunicador **MPI_COMM_WORLD,** que incluye todos los procesos en la ejecución del programa.

2. El objetivo de esta ejercicio es comparar el comportamiento de un algoritmo paralelo versus una versión secuencial. El problema que nos interesa es un algoritmo secuencial que cuenta el número de apariciones de un número en un arreglo muy grande.

Para simular una búsqueda en un arreglo mayor, lo que haremos es buscar N veces en la misma estructura de datos.

Version Secuencial

- Generar el ejecutable de CuentaSec.c
- Ejecutar el programa variando la cardinalidad desde 100.000 hasta 1.000.000.Anotar en la Tabla 1 el tiempo de ejecución de cada corrida (use gprof y compare con el tiempo retornado por el programa). Es conveniente ejecutarlo más de una vez y tomar el promedio de los tiempos.

```
[/tmp/cuentasec]
gattes$ gcc -Wall CuentaSec.c -o CuentaSec

[/tmp/cuentasec]
gattes$ for cardinalidad in 100000 200000 400000 600000 800000 1000000; do echo -e "Ejecutando para cardinalidad ${cardinalidad}"; ./CuentaSec ${cardinalidad}; done
Ejecutando para cardinalidad 100000
Veces que aparece el 8 = 1080000
Tiempo de proceso: 1:426(seg:mseg)
Ejecutando para cardinalidad 200000
Veces que aparece el 8 = 2050000
Tiempo de proceso: 2:790(seg:mseg)
Ejecutando para cardinalidad 400000
Veces que aparece el 8 = 4010000
Tiempo de proceso: 8:655(seg:mseg)
Ejecutando para cardinalidad 600000
Veces que aparece el 8 = 5990000
Tiempo de proceso: 13:869(seg:mseg)
Ejecutando para cardinalidad 800000
Veces que aparece el 8 = 7940000
Tiempo de proceso: 17:118(seg:mseg)
Ejecutando para cardinalidad 1000000
Veces que aparece el 8 = 10010000
Tiempo de proceso: 21:823(seg:mseg)
```

Version Gprof:

```
[/tmp/cuentasec]
gattes$ for cardinalidad in 100000 200000 400000 600000 800000 1000000; do echo -e "Ejecutando para cardinalidad ${cardinalidad}"; ./CuentaSecGProf ${cardinalidad}; gprof -b CuentaSecGProf gmon.out; done
Ejecutando para cardinalidad 100000
Veces que aparece el 8 = 1080000
Tiempo de proceso: 1:345(seg:mseg)
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.00      1.34     1.34                             main
Ejecutando para cardinalidad 200000
Veces que aparece el 8 = 2050000
Tiempo de proceso: 2:752(seg:mseg)
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.00      2.75     2.75                             main
Ejecutando para cardinalidad 400000
Veces que aparece el 8 = 4010000
Tiempo de proceso: 5:615(seg:mseg)
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.00      5.61     5.61                             main
Ejecutando para cardinalidad 600000
Veces que aparece el 8 = 5990000
Tiempo de proceso: 15:111(seg:mseg)
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.00     15.11    15.11                             main
Ejecutando para cardinalidad 800000
Veces que aparece el 8 = 7940000
Tiempo de proceso: 18: 89(seg:mseg)
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.00     18.10    18.10                             main
Ejecutando para cardinalidad 1000000
Veces que aparece el 8 = 10010000
Tiempo de proceso: 21:348(seg:mseg)
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.00     21.35    21.35                             main
```

Version optimizacion (-O3):

```
[/tmp/cuentasec]
gattes$ gcc -Wall -O3 CuentaSec.c -o CuentaSec

[/tmp/cuentasec]
gattes$ for cardinalidad in 100000 200000 400000 600000 800000 1000000; do echo -e "Ejecutando para cardinalidad ${cardinalidad}"; ./CuentaSec ${cardinalidad}; done
Ejecutando para cardinalidad 100000
Veces que aparece el 8 = 1080000
Tiempo de proceso: 0:107(seg:mseg)
Ejecutando para cardinalidad 200000
Veces que aparece el 8 = 2050000
Tiempo de proceso: 0:214(seg:mseg)
Ejecutando para cardinalidad 400000
Veces que aparece el 8 = 4010000
Tiempo de proceso: 0:438(seg:mseg)
Ejecutando para cardinalidad 600000
Veces que aparece el 8 = 5990000
Tiempo de proceso: 0:645(seg:mseg)
Ejecutando para cardinalidad 800000
Veces que aparece el 8 = 7940000
Tiempo de proceso: 0:888(seg:mseg)
Ejecutando para cardinalidad 1000000
Veces que aparece el 8 = 10010000
Tiempo de proceso: 1: 81(seg:mseg)
```

Tabla 1

Cardinalidad	Sin Optimización (seg:mseg)	Sin Optimización con gprof(seg:mseg)	Con Optimización O3 (seg:mseg)
100000	1:426	1:345	0:107
200000	12:790	2:752	0:214
400000	8:655	5:615	0:438
600000	13:869	15:111	0:645
800000	17:118	18:89	0:888
1000000	21:823	21:348	01:81

Como se puede observar en la Tabla 1, la ejecucion con optimizacion es claramente mas rapida que las demas. La razon de esto es que al compilar el código **con gcc usando la opción -O3**, estamos activando el **nivel de optimización 3** del compilador. Este nivel de optimización incluye un conjunto de transformaciones y mejoras que están diseñadas para maximizar el rendimiento del código generado. Para nuestro caso hay algunas mejoras que pudieron haber ayudado:

- Desenrollado de bucles (**loop unrolling**): Es una técnica que consiste en duplicar o triplicar el cuerpo del bucle para reducir la sobrecarga de las instrucciones de salto y permitir una ejecución más eficiente.
- **Code Motion**: Que se utiliza para mover instrucciones fuera de bucles reduciendo así la cantidad de operaciones que se realizan en cada iteración del bucle
- Vectorizacion (**Vectorization**): Es la capacidad de procesar varios elementos de datos simultáneamente en registros SIMD. Esto puede mejorar significativamente el rendimiento, especialmente en operaciones repetitivas.

Dado que este programa secuencial mayormente ejecuta 2 bucles estas optimizaciones hacen que la performance mejore significativamente.

Version Paralela

- Generar una versión paralela sencilla de este programa en la que el maestro reparte el trabajo entre sí mismo y el resto de los procesos. El reparto consistirá en dividir el arreglo en tantos trozos como procesos. Un esqueleto de este programa, al que denominaremos **CuentaPar.c**

Output

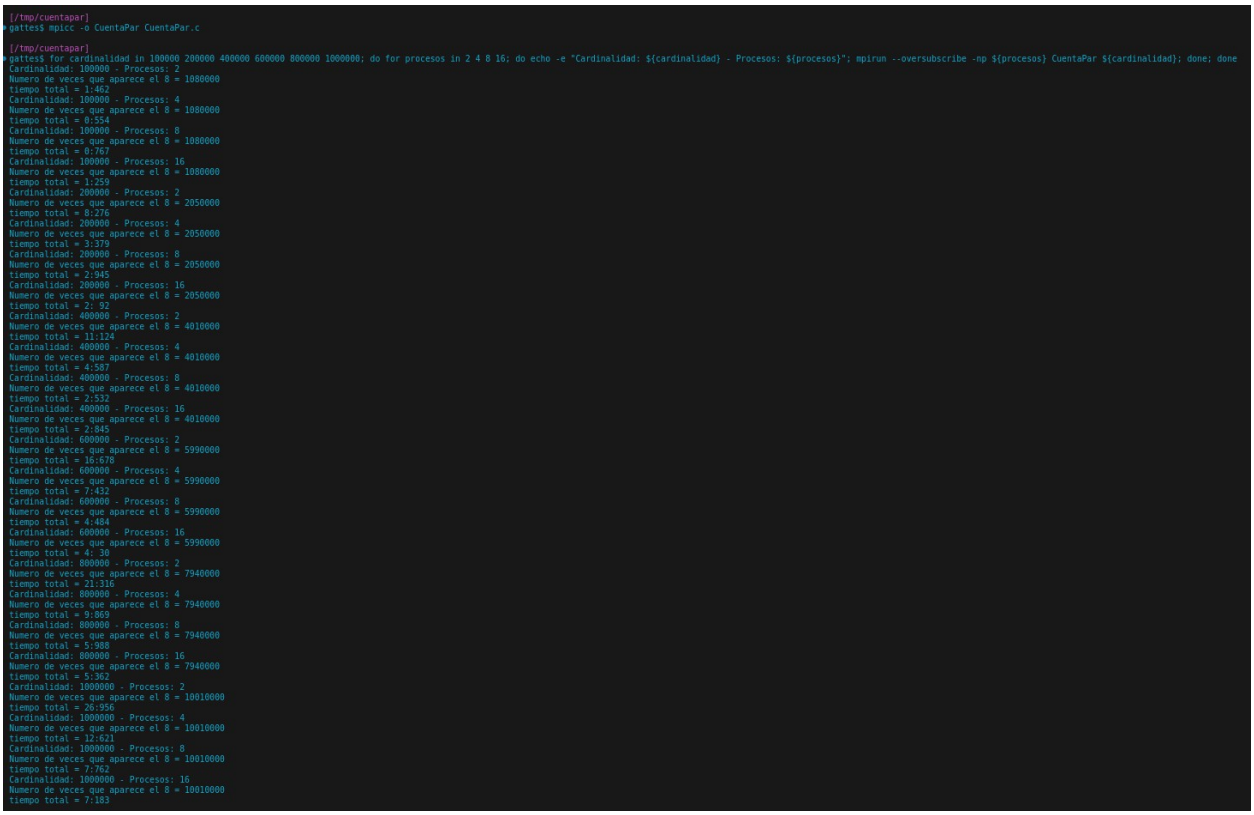


Tabla 2. Version Paralela

Cardinalidad	P=2 (seg:mseg)	P=4 (seg:mseg)	P=8 (seg:mseg)	P=16 (seg:mseg)
100000	1:462	0:554	0:767	1:259
200000	8:276	3:379	2:945	2:92
400000	11:124	4:587	2:532	2:845
600000	16:678	7:432	4:484	4:30
800000	21:316	9:869	5:988	5:362
1000000	29:956	12:621	7:762	7:183

Tabla 3. Calcular la **aceleración** obtenida con respecto a la versión secuencial y analizar el resultado (graficar resultados)

Lo que se hizo en este caso es, para cada P=N (Con N = cantidad de procesos) compararlo contra **la version secuencial sin optimizacion, sin optimizacion con gprof y con optimizacion O3.** Se muestran las tablas para cada uno de estos casos (P=2, P=4, P=8, P=16) y los graficos.

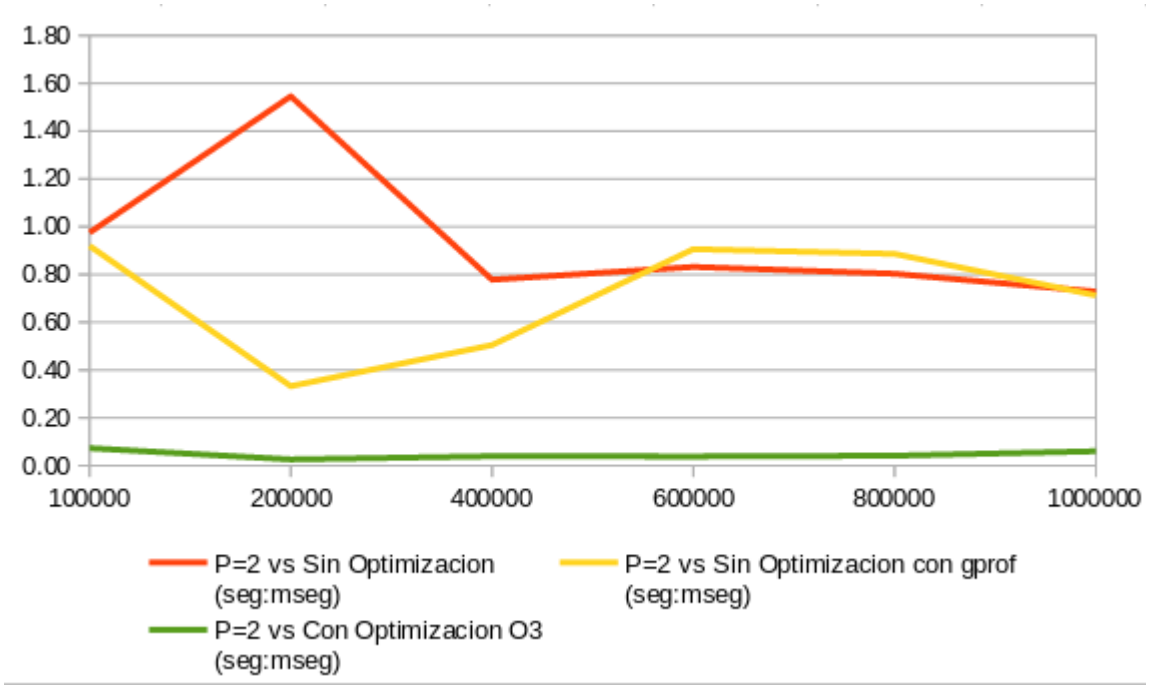
Para **todos los graficos**, **el eje Y representa la aceleracion y el eje X la cardinalidad.**

Recordamos que la formula de aceleracion es $A = \frac{ts}{tp}$

Nota del alumno: Asi lo hice dado que no esta especificado contra que ejecucion en particular de la version secuencial.

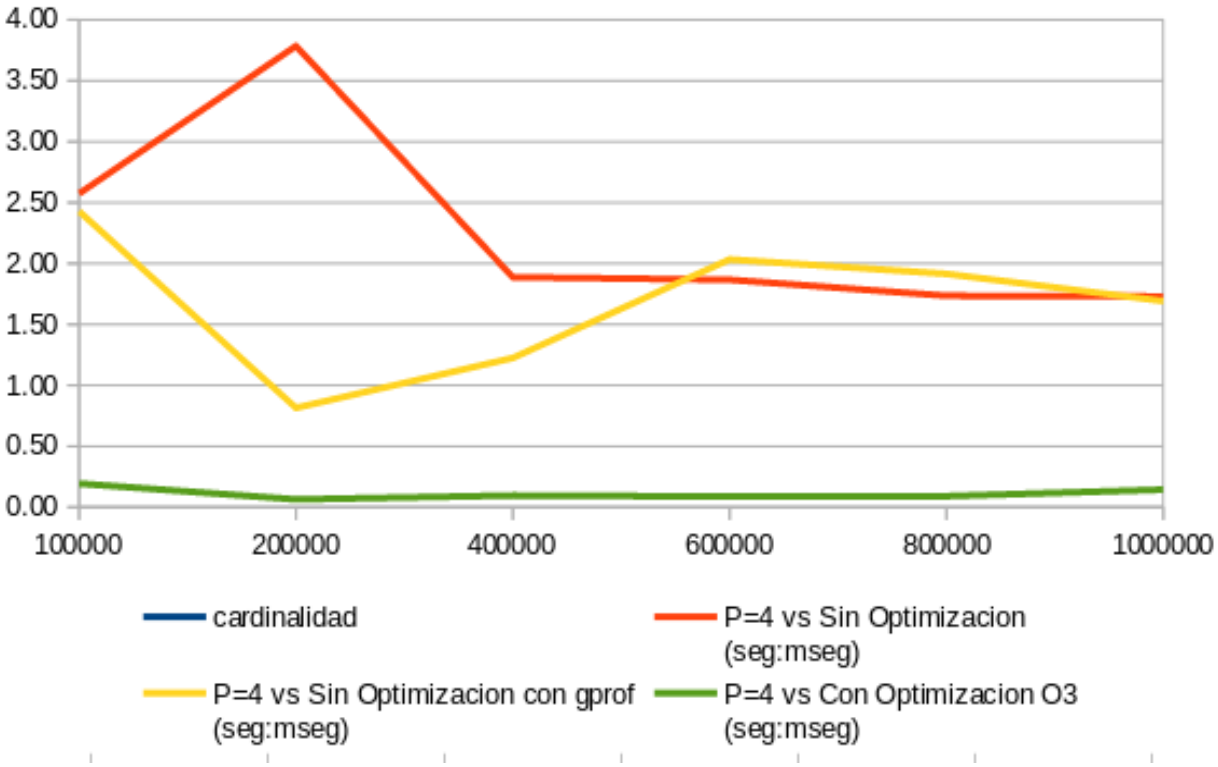
P=2

Cardinalidad	P=2 vs Sin Optimizacion (seg:mseg)	P=2 vs Sin Optimizacion con gprof (seg:mseg)	P=2 vs Con Optimizacion O3 (seg:mseg)
100000	0.98	0.92	0.07
200000	1.55	0.33	0.03
400000	0.78	0.50	0.04
600000	0.83	0.91	0.04
800000	0.80	0.89	0.04
1000000	0.73	0.71	0.06



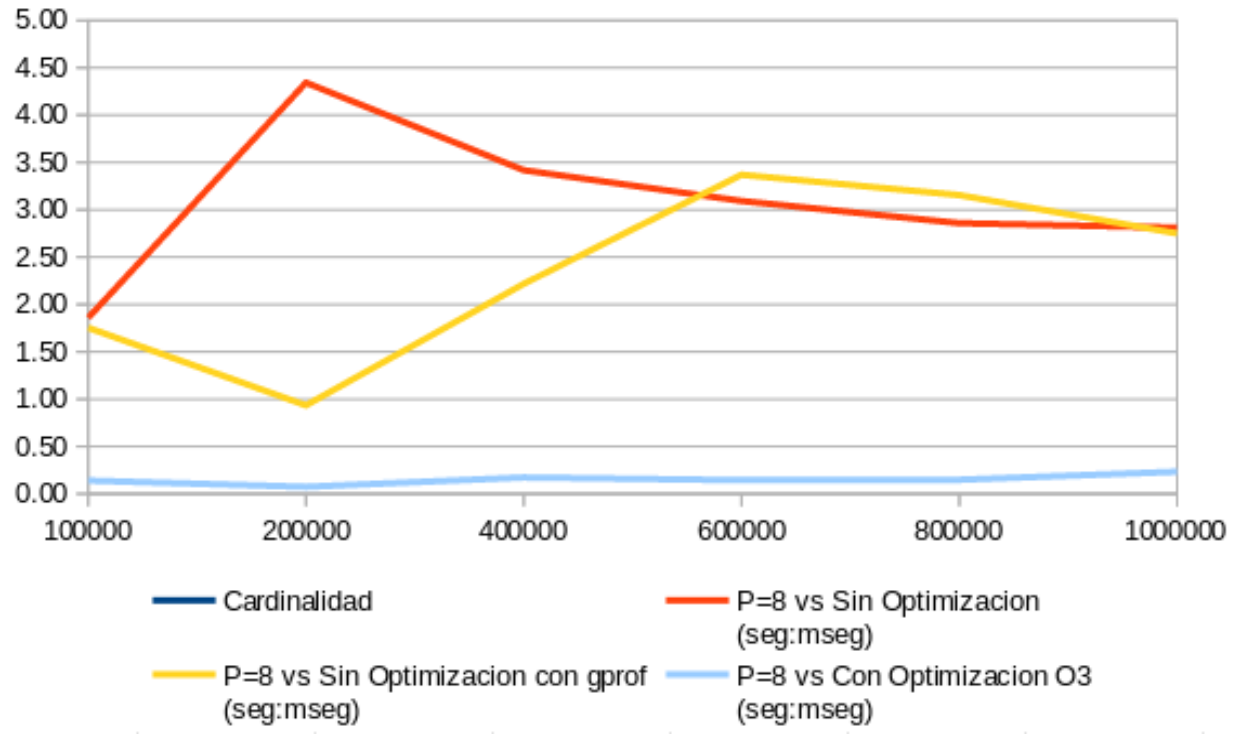
P=4

Cardinalidad	P=4 vs Sin Optimizacion (seg:mseg)	P=4 vs Sin Optimizacion con gprof (seg:mseg)	P=4 vs Con Optimizacion O3 (seg:mseg)
100000	2.57	2.43	0.19
200000	3.79	0.81	0.06
400000	1.89	1.22	0.10
600000	1.87	2.03	0.09
800000	1.73	1.91	0.09
1000000	1.73	1.69	0.14



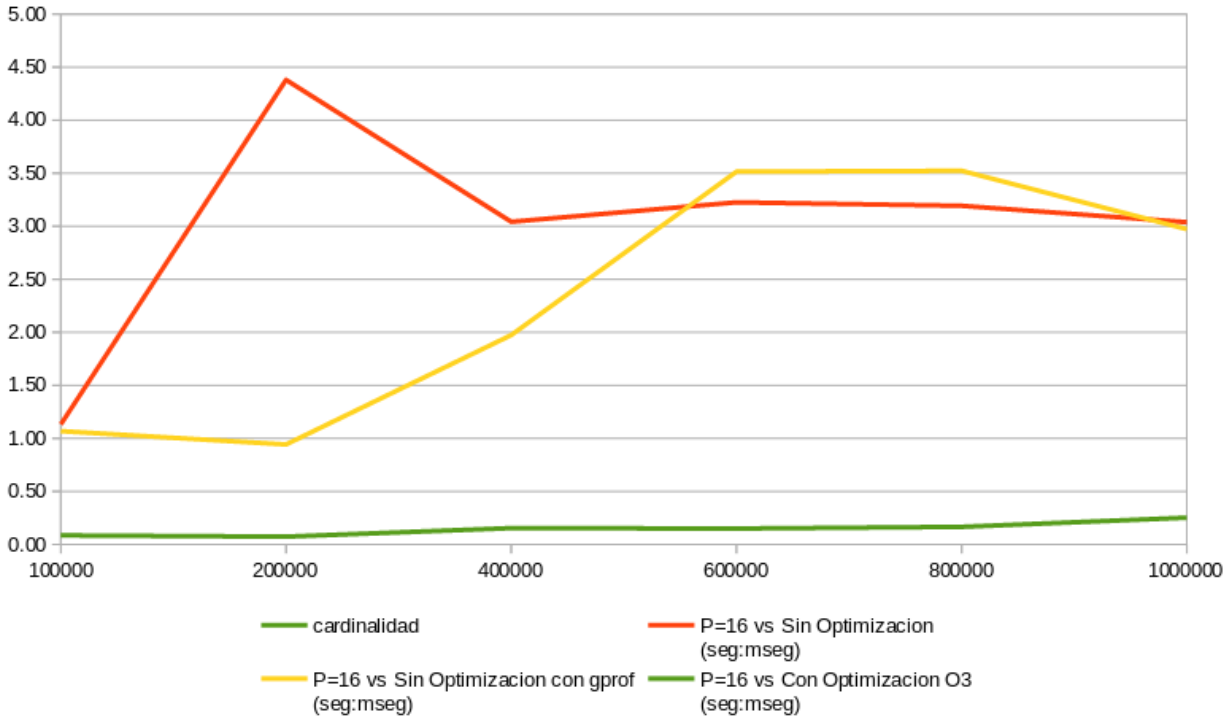
P=8

Cardinalidad	P=8 vs Sin Optimizacion (seg:mseg)	P=8 vs Sin Optimizacion con gprof (seg:mseg)	P=8 vs Con Optimizacion O3 (seg:mseg)
100000	1.86	1.75	0.14
200000	4.34	0.93	0.07
400000	3.42	2.22	0.17
600000	3.09	3.37	0.14
800000	2.86	3.15	0.15
1000000	2.81	2.75	0.23



P=16

Cardinalidad	P=16 vs Sin Optimizacion (seg:mseg)	P=16 vs Sin Optimizacion con gprof (seg:mseg)	P=16 vs Con Optimizacion O3 (seg:mseg)
100000	1.13	1.07	0.08
200000	4.38	0.94	0.07
400000	3.04	1.97	0.15
600000	3.23	3.51	0.15
800000	3.19	3.52	0.17
1000000	3.04	2.97	0.25



Analisis de los resultados de Aceleracion

1. Al analizar todos los graficos de las operaciones paralelas versus la operacion secuencial **sin optimizacion** podemos visualizar que, sobre todo al utilizar 8 y 16 procesos, la mayor aceleracion es alcanzada entre las cardinalidades de 200-400 mil (3 a 4.5). Luego la aceleracion se mantiene estable ~3.

2. Al analizar las operaciones paralelas versus secuencial sin optimizacion **con gprof** vemos que el caso es distinto, para todos los procesos 2, 4, 8 y 16, entre cardinalidades de 100 a 400mil, la aceleracion baja a 1, mientras que al sobrepasar esta cota alcanza niveles similares de aceleracion de ~3, tal como mencionado en el punto anterior.

3. Por ultimo, podemos observar que, **para todas las operaciones en paralelo vs la optimizacion O3**, **la aceleracion es minima (siempre <0.5)**.

A traves de estos 3 analisis vemos como si bien el paralelismo nos provee para este caso en particular con pocos cambios en el codigo (importando las librerias e implementando las funciones) una aceleracion mucho mayor a la secuencial y sin optimizacion, es importante entender que es mas complejo de escribir el codigo para procesamiento paralelo que para secuencial.

Por otro lado, vemos como una buena implementacion del codigo y una compilacion optimizada mejora notablemente la performance de nuestro codigo **aun** habiendo utilizado un solo hilo. A pesar de eso, tambien es importante entender que las optimizaciones del compilador se centran principalmente en mejorar el rendimiento de **un solo hilo o proceso**. No introducen inherentemente paralelismo ni distribuyen el trabajo entre varios procesadores, asi como que

las optimizaciones del compilador **pueden depender de la arquitectura de hardware específica**, **y las optimizaciones que funcionan bien en un tipo de procesador pueden no ser tan efectivas en otro.**

Tabla 4. Calcular la **eficiencia** obtenida con respecto a la versión secuencial y analizar el resultado (graficar resultados)

Lo que se hizo en este caso es tomar la tabla 3 para cada P=N (Con N = cantidad de procesos) calcular su eficiencia en base a su aceleracion contra **la version secuencial sin optimizacion, sin optimizacion con gprof y con optimizacion O3.** Se muestran las tablas para cada uno de estos casos (P=2, P=4, P=8, P=16) y los graficos.

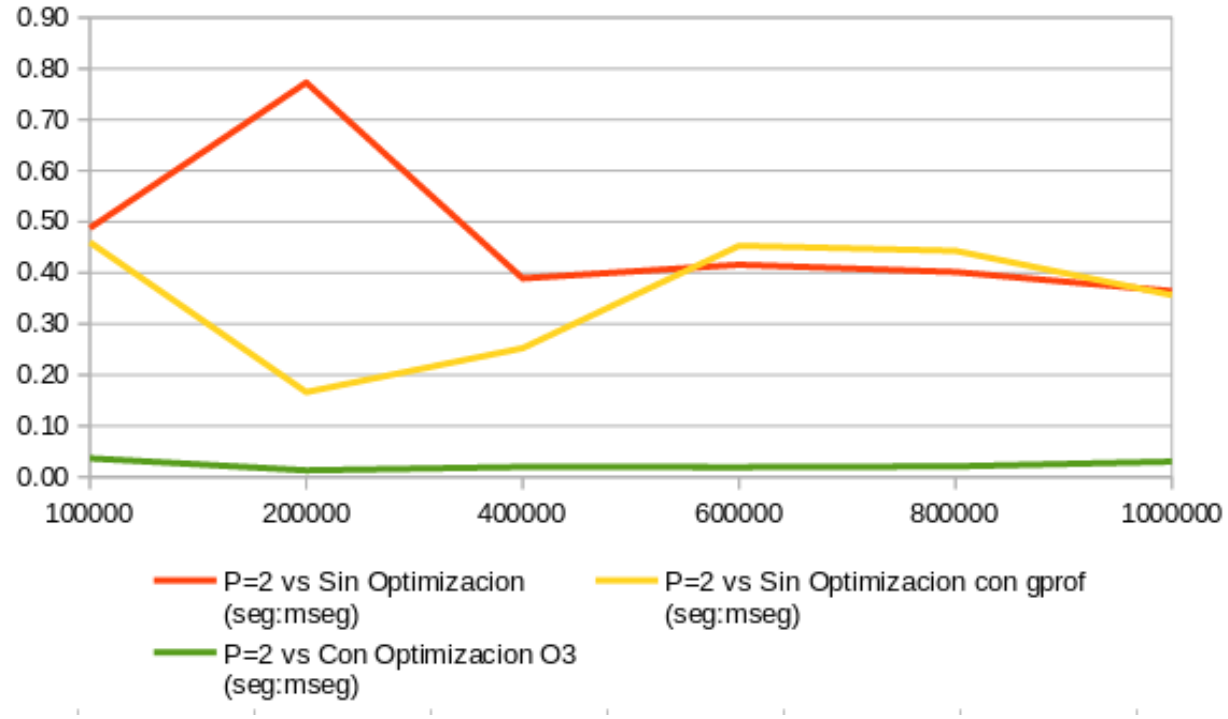
Para **todos los graficos**, **el eje Y representa la eficiencia y el eje X la cardinalidad.**

Recordamos que la formula de eficiencia es $E = \frac{Ac}{P}$

Nota del alumno: Nuevamente, lo hice asi dado que no esta especificado contra que ejecucion en particular de la version secuencial.

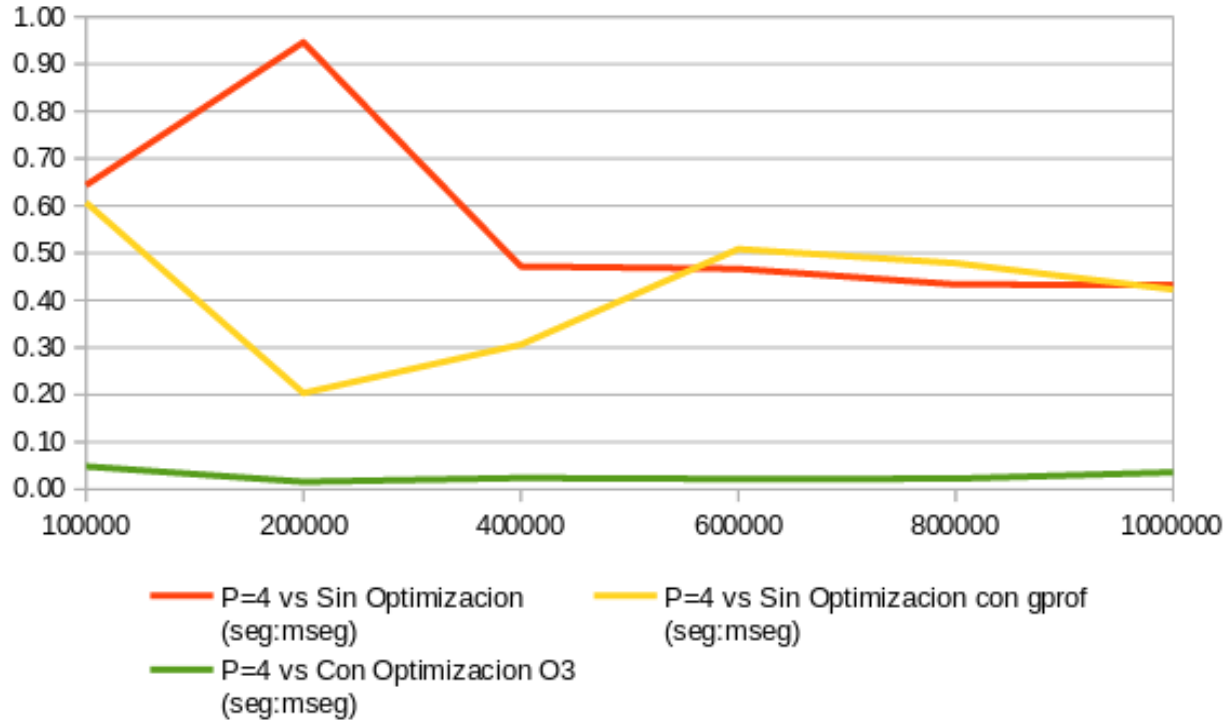
P=2

Cardinalidad	P=2 vs Sin Optimizacion (seg:mseg)	P=2 vs Sin Optimizacion con gprof (seg:mseg)	P=2 vs Con Optimizacion O3 (seg:mseg)
100000	0.49	0.46	0.04
200000	0.77	0.17	0.01
400000	0.39	0.25	0.02
600000	0.42	0.45	0.02
800000	0.40	0.44	0.02
1000000	0.36	0.36	0.03



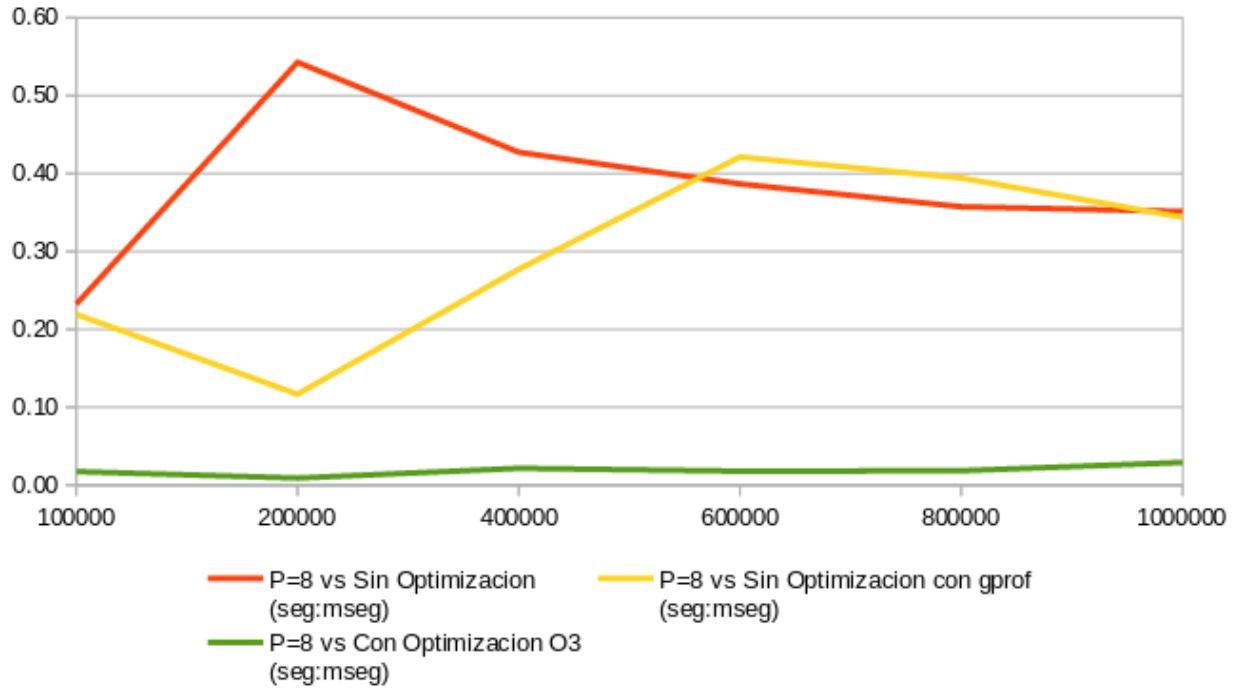
P=4

Cardinalidad	P=4 vs Sin Optimizacion (seg:mseg)	P=4 vs Sin Optimizacion con gprof (seg:mseg)	P=4 vs Con Optimizacion O3 (seg:mseg)
100000	0.64	0.61	0.05
200000	0.95	0.20	0.02
400000	0.47	0.31	0.02
600000	0.47	0.51	0.02
800000	0.43	0.48	0.02
1000000	0.43	0.42	0.04



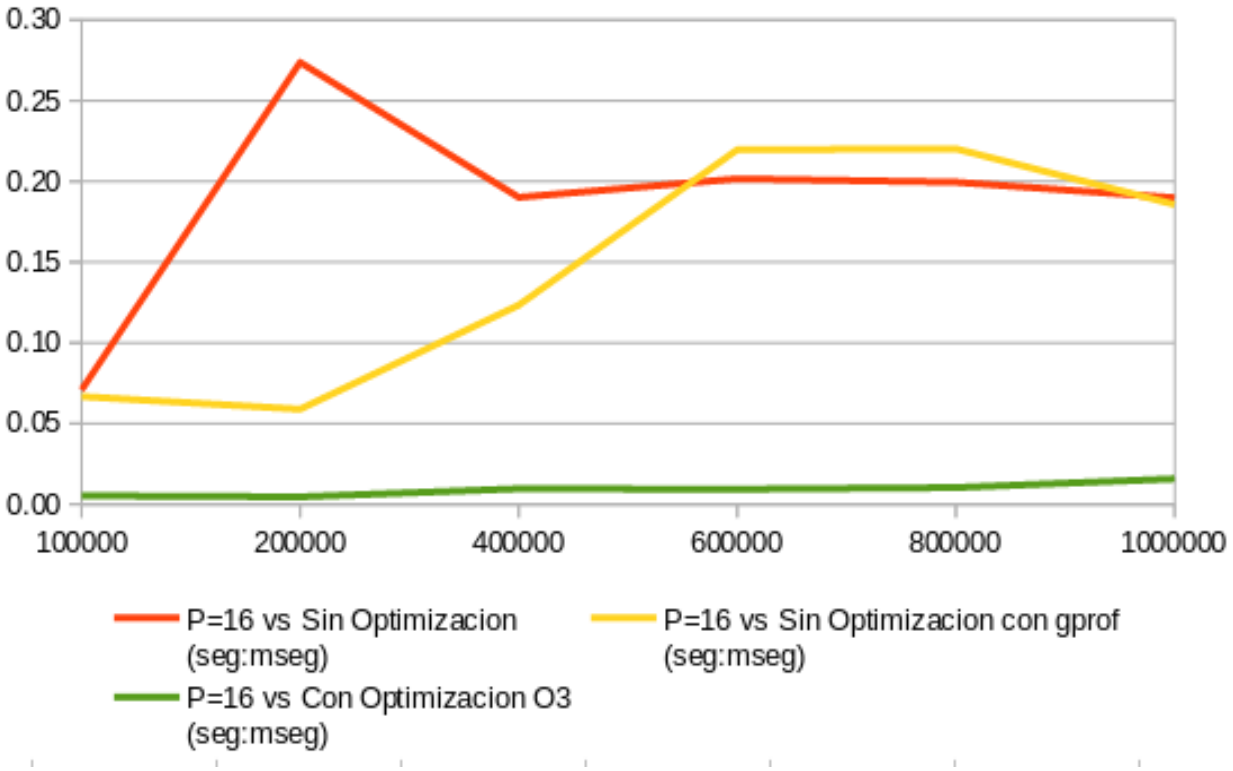
P=8

Cardinalidad	P=8 vs Sin Optimizacion (seg:mseg)	P=8 vs Sin Optimizacion con gprof (seg:mseg)	P=8 vs Con Optimizacion O3 (seg:mseg)
100000	0.23	0.22	0.02
200000	0.54	0.12	0.01
400000	0.43	0.28	0.02
600000	0.39	0.42	0.02
800000	0.36	0.39	0.02
1000000	0.35	0.34	0.03



P=16

Cardinalidad	P=16 vs Sin Optimizacion (seg:mseg)	P=16 vs Sin Optimizacion con gprof (seg:mseg)	P=16 vs Con Optimizacion O3 (seg:mseg)
100000	0.07	0.07	0.01
200000	0.27	0.06	0.00
400000	0.19	0.12	0.01
600000	0.20	0.22	0.01
800000	0.20	0.22	0.01
1000000	0.19	0.19	0.02



Analisis de los resultados de Eficiencia

1. Al analizar todos los graficos de las operaciones paralelas versus la operacion secuencial **sin optimizacion** podemos visualizar que la mayor eficiencia se encuentra al utilizar 4 procesos en con cardinalidad 200mil. Esto es muy probable dado que el procesador utilizado para esta ejecucion es de 4 nucleos y todos ellos se aprovechan. Aun asi vemos que al aumentar la cardinalidad, si bien la eficiencia disminuye, se mantiene en un valor positivo de ~0.5

2. Al analizar las operaciones paralelas versus secuencial sin optimizacion **con gprof** vemos que el caso es distinto, para todos los casos la eficiencia primero decae y aumenta a medida que lo hace la cardinalidad.

3. Por ultimo, podemos observar que, **para todas las operaciones en paralelo vs la optimizacion O3**, **la eficiencia es imperceptible.** Lo cual apunta a un completo desperdicio de recursos entre la version paralela contra la secuencial.

3. MPI: Comunicaciones Punto a Punto

3.1. Este ejercicio ayuda a visualizar el desempeño relativo de los cuatro modos de comunicación (synchronous, ready, buffered, y standard). El programa blocksends.c indica el tiempo (wallclock) empleado en llamadas bloqueantes para los cuatro modos de comunicación. Todos los receives son colocados antes que cualquier mensaje sea enviado, nótese que se obtendrían tiempos diferentes si los receives no se colocan así.

- Lea el programa blocksends.c
- Compílelo y ejecútelo
- Note el tiempo empleado por el send bloqueante en los cuatro modos.

```
gattes@lagias:/tmp$ mpicc -o blocksends blocksends.c
gattes@lagias:/tmp$ mpirun blocksends
 Usage: blocksends <message_length_in_number_of_floats>
 Usage: blocksends <message_length_in_number_of_floats>
 Usage: blocksends <message_length_in_number_of_floats>
--------------------------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been aborted.
--------------------------------------------------------------------------
 Usage: blocksends <message_length_in_number_of_floats>
gattes@lagias:/tmp$ mpirun blocksends 10000
 Message size = 10000 floats
 Task 0 initialized
 Message size = 10000 floats
 Task 1 initialized
 Message size = 10000 floats
 Task 3 initialized
 Message size = 10000 floats
 Task 2 initialized
 Ready to send messages
 Elapsed time for synchronous send =       29 uSec
 Elapsed time for ready send =       17 uSec
 Elapsed time for buffer allocation =        3 uSec
 Elapsed time for buffered send =       22 uSec
 Elapsed time for standard send =       14 uSec
```

3.2. Este ejercicio permite constatar que las rutinas no bloqueantes son más seguras que las bloqueantes. Use el programa deadlock.c.

- Compile el programa **deadlock.c**
- Ejecútelo con dos procesos. ¿Qué sucede?
- Varíe el tamaño del mensaje dentro del programa. Dependiendo del tamaño del mensaje, el programa escribirá unas líneas y luego se detendrá. Aborte el programa con <ctrl. C>.
- ¿Por qué el programa entra en interbloqueo?
- Corrija el programa de manera de evitar el interbloqueo
- Compare su solución con la original dada.

```
gattes@lagias:/tmp$ mpicc -o deadlock deadlock.c
gattes@lagias:/tmp$ mpirun -np 2 deadlock
 Task 0 initialized
 Task 1 initialized
 Task 0 has sent the message
 Task 1 has sent the message
```

He tenido que abortar la ejecucion utilizando **Ctrl + C** dado que ambos procesos se bloquean nunca finalizan, que es lo que se supone que hace: Un deadlock o abrazo de la muerte.
En todos los casos y a pesar de variar el tamaño del mensaje dentro del programa (linea 13: **#define MSGLEN 102400**) el mismo ingresara en un deadlock. El mismo ocurre porque los 2 procesos intentan enviarse mensajes entre si al mismo tiempo (ergo, de manera simultanea) utilizando la operacion **MPI_Send** que es de **envio bloqueante**, y **MPI_Recv** que es de **recepcion bloqueante.**

El problema radica finalmente en que ambas tareas están bloqueadas esperando un mensaje la una de la otra y dado que ambas están en estado de espera, el programa entra en bloqueo (deadlock) y no avanza más.

Para **solucionar** este problema, y tal como se comenta en la linea **4: “To fix the code, replace blocking send with non-blocking send”**.

Por tanto procedo a reemplazar las operaciones **MPI_Send** con **MPI_Isend, las ultimas lineas del programa quedaran de la siguiente manera:**

```
57  /* ------------------------------------------------------------
58   * send and receive messages <Using non-blocking comm>
59   * ------------------------------------------------------------ */
60  MPI_Request request;
61  printf ( " Task %d has initiated non-blocking send\n", rank );
62  MPI_Isend ( message1, MSGLEN, MPI_FLOAT, dest, send_tag, MPI_COMM_WORLD, &request );
63  printf ( " Task %d has sent the message\n", rank );
64
65  MPI_Recv ( message2, MSGLEN, MPI_FLOAT, source, recv_tag, MPI_COMM_WORLD, &status );
66  printf ( " Task %d has received the message\n", rank );
67  MPI_Wait(&request, &status);
68
69  MPI_Finalize();
```

En este caso he agregado la función **MPI_Isend** para realizar el envío **no bloqueante**. La función **MPI_Wait** se utiliza después de la recepción (**MPI_Recv**) **para esperar a que la operación de envío no bloqueante se complete**.

Una vez mas, compilado y ejecutado vemos como el deadlock esta resuelto:

```
gattes@lagias:/tmp$ mpicc -o deadlock deadlock.c
gattes@lagias:/tmp$ mpirun -np 2 deadlock
 Task 1 initialized
 Task 0 initialized
 Task 1 has initiated non-blocking send
 Task 1 has sent the message
 Task 0 has initiated non-blocking send
 Task 0 has sent the message
 Task 0 has received the message
 Task 1 has received the message
```

4. MPI: Comunicaciones Colectivas

Se quiere que implemente un programa similar al presentado en **avg.c** y **all_avg.c**, pero en este caso el objetivo será encontrar el valor mínimo, máximo y promedio de los elementos que están contenidos en una matriz de números reales. Tomen en cuenta las siguientes especificaciones:

- El programa debe crear una matriz de dimensión N×N, donde N es el número de procesos a ejecutar. La matriz debe ser llenada con números reales generados aleatoriamente.
- El proceso maestro (o raíz) debe repartir una fila para cada proceso, usando la primitiva **MPI_Scatter** para que cada proceso, incluyéndolo.
- Cada proceso debe procesar la fila que le corresponde para conseguir el valor mínimo, máximo y el promedio de su fila.
- Cada proceso debe mostrar su identificador, junto con el valor mínimo, máximo y promedio de su fila en pantalla, con:

  printf(“Process %d with row %d – min: %f; max: %f; avg: %f”, myrank, myrow, mymin, mymax, myavg);
- Luego del cómputo, el proceso raíz o maestro colectará todos los valores de todos los procesos (que vendrán en un arreglo de tres elementos, con el mínimo, el máximo y el promedio respectivamente) mediante la primitiva **MPI_Gather**.
- Para terminar, el proceso raíz toma el contenido de todos los arreglos y encuentra los valores mínimo, máximo y promedio, entre los valores mínimos, máximos y promedios recibidos. Luego de lo cual los mostrará por pantalla.

Codigo

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <assert.h>

// Crea la matriz cuadrada N x N de numeros reales aleatorios
float *create_rand_matrix(int rows, int cols) {
  float *matrix = (float *)malloc(sizeof(float) * rows * cols);
  assert(matrix != NULL);

  int i, j;
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      matrix[i * cols + j] = (rand() / (float)RAND_MAX);
    }
  }
  return matrix;
}

// Realiza el computo de minimo, maximo y promedio (avg)
void compute_stats(float *row, int num_elements, float *min, float *max, float *avg) {
  *min = row[0];
  *max = row[0];
  *avg = 0.0;

  int i;
  for (i = 0; i < num_elements; i++) {
    *avg += row[i];
    if (row[i] < *min) {
      *min = row[i];
    }
    if (row[i] > *max) {
      *max = row[i];
    }
  }

  *avg /= num_elements;
}
```

```c
int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);

  int myrank;
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  if (argc != 2) {
    fprintf(stderr, "Uso: %s <número_de_procesos>\n", argv[0]);
    MPI_Abort(MPI_COMM_WORLD, 1);
  }

  int num_processes = atoi(argv[1]);
  int rows_per_process = num_processes;
  int cols = num_processes;

  float *matrix = NULL;
  float *sub_row = (float *)malloc(sizeof(float) * cols);
  float min, max, avg;

  if (myrank == 0) {
    srand(time(NULL));
    matrix = create_rand_matrix(num_processes, num_processes);
    // Imprime la matriz generada
    printf("Matriz generada:\n");
    for (int i = 0; i < num_processes; i++) {
      for (int j = 0; j < num_processes; j++) {
        printf("%f ", matrix[i * num_processes + j]);
      }
      printf("\n");
    }
  }

  // Scatter filas de la matriz a cada proceso
  MPI_Scatter(matrix, cols, MPI_FLOAT, sub_row, cols, MPI_FLOAT, 0, MPI_COMM_WORLD);

  // Calcula las estadísticas para la fila asignada a cada proceso
  compute_stats(sub_row, cols, &min, &max, &avg);

  // Imprime las estadísticas para cada proceso
  printf("Process %d with row %d - min: %f; max: %f; avg: %f\n", myrank, myrank, min, max, avg);

  // Gather las estadísticas parciales para cada proceso en el proceso raíz
  float *stats_array = (float *)malloc(sizeof(float) * num_processes * 3); // 3 elementos por proceso (min, max, avg)
  assert(stats_array != NULL);
  MPI_Gather(&min, 1, MPI_FLOAT, stats_array, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Gather(&max, 1, MPI_FLOAT, stats_array + num_processes, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Gather(&avg, 1, MPI_FLOAT, stats_array + 2 * num_processes, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

  // Proceso raíz calcula el mínimo, máximo y promedio global
  if (myrank == 0) {
    float global_min, global_max, global_avg;
    compute_stats(stats_array, num_processes, &global_min, &global_max, &global_avg);
    printf("Global minimum: %f; Global maximum: %f; Global average: %f\n", global_min, global_max, global_avg);
  }

  // Libera memoria
  if (myrank == 0) {
    free(matrix);
  }
  free(sub_row);
  free(stats_array);

  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();

  return 0;
}
```

Compilacion y ejecucion

No pude resolver como el programa tomaria el numero de procesos a traves del flag **-np**, por lo cual hay que pasarle el mismo numero de procesos que de tamaño de matriz, por ejemplo **-np 4 all_matrix 4**, o bien **-np 8 all_matrix 8**, como en los siguientes ejemplos:

```
gattes@lagias:/tmp$ mpicc -o all_matrix all_matrix.c
gattes@lagias:/tmp$ mpirun --oversubscribe -np 4 all_matrix 4
Matriz generada:
0.430983 0.344666 0.078760 0.202011
0.646518 0.510708 0.363272 0.217167
0.269520 0.578685 0.736671 0.945394
0.941187 0.453604 0.950264 0.571822
Process 1 with row 1 - min: 0.217167; max: 0.646518; avg: 0.434416
Process 2 with row 2 - min: 0.269520; max: 0.945394; avg: 0.632567
Process 0 with row 0 - min: 0.078760; max: 0.430983; avg: 0.264105
Process 3 with row 3 - min: 0.453604; max: 0.950264; avg: 0.729219
Global minimum: 0.078760; Global maximum: 0.453604; Global average: 0.254763
gattes@lagias:/tmp$ mpirun --oversubscribe -np 8 all_matrix 8
Matriz generada:
0.723518 0.907922 0.090302 0.303331 0.314367 0.612556 0.169283 0.689850
0.794026 0.503059 0.781477 0.751121 0.948814 0.209924 0.903570 0.871073
0.159494 0.437138 0.441892 0.830887 0.338759 0.039054 0.092697 0.217973
0.397485 0.364748 0.933294 0.868491 0.609095 0.800482 0.513114 0.332613
0.708404 0.603416 0.635944 0.022771 0.215972 0.805227 0.712621 0.009998
0.308285 0.494098 0.761119 0.257099 0.704022 0.664689 0.128172 0.863515
0.101828 0.570064 0.694402 0.440587 0.609118 0.787099 0.658560 0.006603
0.151846 0.591854 0.875094 0.760941 0.392336 0.388208 0.093553 0.100740
Process 4 with row 4 - min: 0.009998; max: 0.805227; avg: 0.464294
Process 5 with row 5 - min: 0.128172; max: 0.863515; avg: 0.522625
Process 6 with row 6 - min: 0.006603; max: 0.787099; avg: 0.483532
Process 0 with row 0 - min: 0.090302; max: 0.907922; avg: 0.476391
Process 1 with row 1 - min: 0.209924; max: 0.948814; avg: 0.720383
Process 7 with row 7 - min: 0.093553; max: 0.875094; avg: 0.419322
Process 2 with row 2 - min: 0.039054; max: 0.830887; avg: 0.319737
Process 3 with row 3 - min: 0.332613; max: 0.933294; avg: 0.602415
Global minimum: 0.006603; Global maximum: 0.332613; Global average: 0.113777
```

5. OpenMP: Esta actividad consta de nueve ejercicios sobre el uso de la librería OpenMP para la creación de programas paralelos en memoria compartida.

6.1. Ejecute el programa **e1.c**: ¿Cuáles de las instrucciones del tipo **#include** se pueden eliminar en este programa sin impedir que se ejecute correctamente y por qué pueden ser eliminadas?

El único **#include** necesario para que el programa ejecute correctamente es **#include <omp.h>.** La razon es que sin esta inclusión, el compilador no reconocerá la instrucciones de OpenMP **#pragma omp parallel** y se generará un error durante la compilación.

Para el caso de las instrucciones **printf** y **exit**, normalmente el compilador proporciona estas definiciones por defecto **aunque no es recomendable** y es buena practica incluirlos por defecto.

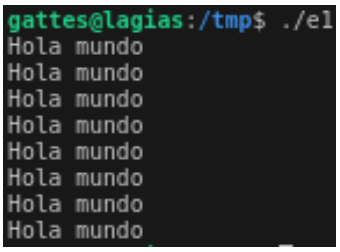
6.2. Ejecute el programa e2.c

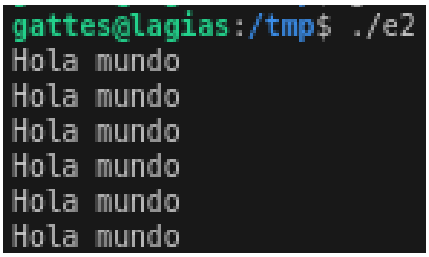
a) Compare la salida con el programa e2.c

b) Ejecute el programa e1.c con la opción export OMP_NUM_THREADS=8 y explique.

a) Ambos programas se ejecutan con diferentes cantidad de hilos, en el caso de **e2.c** se ejecuta con 6 hilos definidos en:

```
int cantidad_hilos = 6;

omp_set_num_threads(cantidad_hilos);
```





b) Dado que el programa **e1.c** se ejecutara por default tantas veces como hilos en el equipo, y que el equipo donde realizo esta prueba tiene 8 hilos, no tiene mucho sentido setear **OMP_NUM_THREADS=8** (que da el mismo resultado que sin setearlo). En este ejemplo lo realizo con **OMP_NUM_THREADS=4**.

La variable de entorno **OMP_NUM_THREADS** se utiliza para controlar el número de hilos que OpenMP utilizará en un programa paralelo. Al ejecutar el programa de OpenMP con la opción **OMP_NUM_THREADS=4** como en este ejemplo, se establece la cantidad de hilos en 4 antes de ejecutar el programa:

6.3. Las variables que hayan sido declaradas antes del inicio de la sección paralela, serán compartidas entre todos los hilos de ejecución. Observe el programa **e3.c**
La variable **comp** es compartida entre los hilos durante la sección paralela, mientras que la variable **priv** es privada para cada hilo. Así, invariablemente la impresión de los hilos dará como resultado 1 para todas las variables **priv**, mientras que el resultado de las variables **comp** podría ser imprevisible. Explique por qué sucede esto.

Dado que la variable **comp** es compartida todos los hilos incrementan la misma, sin embargo **no hay control de concurrencia** lo cual podria incurrir en una **race condition** o **condición de carrera**. O sea, varios hilos podrían intentar incrementar **comp** simultáneamente lo cual la hace **imprevisible**, finalmente, debido a la falta de sincronización entre los hilos o control de concurrencia (como se menciono antes).

6.4. Ejecute el programa **e4.c:**
a) Qué función cumple la función **shared(sum)** de la directiva **pragma**
La funcion **shared(sum)** se utiliza para especificar que la variable **sum** es compartida entre los hilos que ejecutan el bucle for dentro de la region paralela declarada bajo la directiva **pragma.**

b) Ejecute varias veces el programa ¿observa algún error en alguna ejecución? ¿a qué se debe el error?

El valor de la variable **r** varia en cada ejecucion, esto es asi porque, tal como se explico en el ejercicio anterior pero en este caso con la variable **sum**, la misma está siendo compartida entre los hilos y modificada por los mismos sin un mecanismo de sincronización adecuado, lo que podría dar lugar a una condición de carrera, generando asi resultados indeseados o inospitos en el valor de **sum**.

6.5. Ejecute el programa **e5.c:**

a) Qué función cumple la directiva **#pragma omp critical**

La directiva **#pragma omp critical** en OpenMP se utiliza para especificar una **región crítica** del código, donde **un solo hilo puede ejecutar esa región a la vez**. Es una forma de evitar condiciones de carrera. En este caso la región crítica se utiliza para proteger la operación de acumulación en la variable **sum** dentro del bucle paralelo.

b) Ejecute varias veces el programa ¿observa algún error en alguna ejecución? ¿a qué se debe el error?

No, no hay errores dado que la directiva **#pragma omp critical** ahora resuelve la condicion de carrera, no generando resultados indeseados.

6.6. Ejecute el programa **e6.c:**

¿Qué ventaja representa el uso de **reduction(+:sum)** con respecto a los códigos e4.c y e5.c ?

La cláusula **reduction(+:sum)** indica a OpenMP que realice la operación de suma de manera **segura** para cada hilo y **luego combine los resultados al final del bucle**, evitando así condiciones de carrera y garantizando la consistencia del resultado.

En cuanto al ejercicio **e4.c** resolveria el problema de **race condition,** en cuanto al **e5.c** es una mejora dado que **reduction(+:sum)** puede realizar la reduccion de manera mas optimizada y paralela, mientras **#pragma omp critical** introduce cierto grado de serializacion dado que cada hilo tiene que esperar su turno para ejecutar la region critica

6.7. Ejecute el programa **e7.c:**

Explique el funcionamiento de este programa. Puede usar dibujos ilustrativos.

Es un programa que calcula el producto escalar de dos vectores **a** y **b,** ambos de longitud **N** y ambos se llenan con valores, donde **a[i] = i** y **b[i] = i** para cada índice i desde **0 hasta N-1.**

En el bucle paralelo de la funcion **producto, cada hilo calcula su contribución al producto escalar**, que es la suma de los productos de los elementos correspondientes de los vectores**.**

Como se comento en el punto anterior, se utiliza **reduction(+:sum)** para realizar la reducción de manera segura y eficiente, y cada hilo tiene su propia variable privada sum.

Finalmente en el bucle de la funcion **main** se imprime cada elemento de los vectores a, b, y el resultado del producto individualmente.

6.8. Ejecute y analice el programa **e8-serial.c:**

Paralelice el código usando OpenMP y responda las siguientes preguntas:

1. ¿Cuáles variables deben ser compartidas y cuáles privadas?

La unica variable compartida es **sum**. La variable **x** e **i** (del bucle) son privadas.

2. ¿Debe haber secciones críticas?

No hace falta una seccion critica (aunque podria implementarse) ya que en este caso decidi usar **reduction(+:sum)** que es mas seguro y confiable que una region critica.

3. De no haberlas, ¿puede resolverse usando otra técnica?

Tal como en el punto anterior, preferi usar la tecnica de operacion de suma de manera segura utilizando **reduction(+:sum)**

```c
/* Programa e8.c */
#include "stdio.h"
#include "stdlib.h"

static long num_steps = 100000;
double step, pi;

int main() {
    int i;
    double x, sum = 0.0;

    step = 1.0 / (double)num_steps;

    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < num_steps; i++) {
        x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }

    pi = step * sum;
    printf("Pi = %f\n", pi);

    return 0;
}
```

6.9. Ejecute y analice el programa **e9-serial.c:**

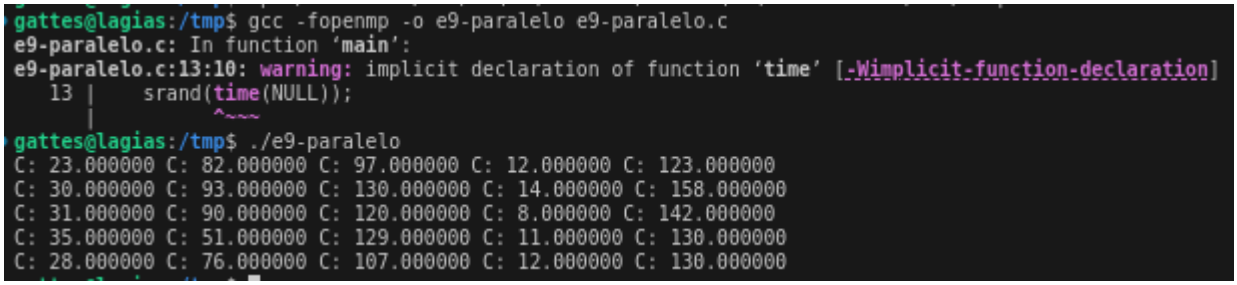
Implemente una versión paralela usando OpenMP**.**

El programa realiza la multiplicación de dos matrices cuadradas N x N.

```
gattes@lagias:/tmp$ gcc -fopenmp -o e9-serial e9-serial.c
e9-serial.c: In function 'main':
e9-serial.c:10:12: warning: implicit declaration of function 'time' [-Wimplicit-function-declaration]
   10 |     srand ( time(NULL) );
      |             ^~~~
gattes@lagias:/tmp$ ./e9-serial
C: 92.000000 C: 120.000000 C: 36.000000 C: 80.000000 C: 97.000000
C: 28.000000 C: 68.000000 C: 43.000000 C: 87.000000 C: 57.000000
C: 59.000000 C: 36.000000 C: 70.000000 C: 81.000000 C: 63.000000
C: 92.000000 C: 109.000000 C: 69.000000 C: 131.000000 C: 98.000000
C: 26.000000 C: 70.000000 C: 34.000000 C: 81.000000 C: 53.000000
```

Una version paralela podria ser:

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 5

int main() {
   /* DECLARACION DE VARIABLES */
   float A[N][N], B[N][N], C[N][N]; // declaracion de matrices de tamaño NxN
   int i, j, m; // indices para la multiplicacion de matrices

   /* LLENAR LAS MATRICES CON NUMEROS ALEATORIOS */
   srand(time(NULL));
   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
         A[i][j] = (rand() % 10);
         B[i][j] = (rand() % 10);
      }
   }

   /* MULTIPLICACION DE LAS MATRICES */
   #pragma omp parallel for collapse(2) private(m) shared(A, B, C)
   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
         C[i][j] = 0.0; // colocar el valor inicial para el componente C[i][j] = 0
         for (m = 0; m < N; m++) {
            C[i][j] += A[i][m] * B[m][j];
         }
      }
   }

   /* IMPRIMIR LA MATRIZ RESULTANTE */
   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
         printf("C: %f ", C[i][j]);
      }
      printf("\n");
   }

   /* FINALIZAR EL PROGRAMA */
   return 0;
}
```



REFERENCIAS

- Manual de la asignatura 23GIIN – Paralelismo: https://shorturl.at/coFLW