Universidad Nacional de Entre Ríos

Facultad de Ingeniería

Trabajo práctico N° 3

Computación de Alto Rendimiento

Autor:

Justo Garcia

Octubre 2023

Tabla de contenidos

1 Introducción

En este informe desarrollaré las actividades propuestas para el trabajo práctico número 3 de Computación de Alto Rendimiento. Iré adjuntando los códigos necesarios, sin embargo recomiendo revisar el repositorio de Github de este trabajo para obtener una mejor visualización de las soluciones propuestas.

2 Ejercicio 1

2.1 Consigna

Implementar un código con OpenMP que realice la suma de todos los elementos de una matriz en paralelo.

1. Probar con diferentes formas de acceso a las componentes de la matriz (por columna y por fila).
2. Comparar tiempos absolutos y speedup en ambos casos.
3. Discutir los resultados obtenidos.

2.2 Resolución

2.2.1 Implementación con OpenMP

Para la resolución de este ejercicio decidí implementar un ciclo que vaya aumentando el número de hilos para así obtener el dato del tiempo para distintos valores, obteniendo así también una ejecución secuencial cuando se corre con tan solo un hilo.
Luego implementé la suma (y registros de tiempos pertinentes) de todos los elementos de la matriz accediendo de dos formas:

- Por filas.
- Por columnas.

Por otro lado, como el número de acceso será dependiente de la cantidad de elementos implementé el código de forma que se corra con diversos tamaños de matrices cuadradas.
El código es el siguiente:

```cpp
#include <stdio.h>
#include <iostream>
#include <vector>
#include <omp.h>
#include <ctime>

using namespace std;

int main(int argc, char **argv)
{
    freopen("salida.csv", "w", stdout);
    int num_iteraciones = 5;
    vector<int> lista_de_filas(num_iteraciones, 500);
    vector<int> lista_de_columnas(num_iteraciones, 500);
    for(int i = 0 ; i < lista_de_filas.size(); i++)
    {
        lista_de_filas[i] = lista_de_filas[i]+(i*2000);
        lista_de_columnas[i] = lista_de_columnas[i]+(i*2000);
    }
    int maxThreads = omp_get_max_threads()/2;
    printf("hilos,f,c,acceso,tiempo\n");

    omp_set_num_threads(2);

    for(int it = 0 ; it < num_iteraciones ; it ++)
    {
        for(int hilos = 1 ; hilos <= maxThreads ; hilos++)
        {
            omp_set_num_threads(hilos);
            for(int iteracion = 0 ; iteracion < num_iteraciones ;
   iteracion++)
            {
                int filas = lista_de_filas[iteracion];
                int columnas = lista_de_columnas[iteracion];

                double wt0, wt1;


```

```
39              vector<vector<int>> matriz(filas, vector<int>(columnas, 1)
    );
40
41              // Accediendo a filas
42              int i = 0, j = 0, suma = 0;
43              wt0 = omp_get_wtime();
44              #pragma omp parallel default(none) shared(matriz) private(
    i, j) reduction(+:suma)
45              {
46                  // printf("%d\n", omp_get_num_threads());
47                  #pragma omp for
48                  for(i = 0 ; i < matriz.size(); i++)
49                  {
50                      for(j = 0 ; j < matriz[0].size(); j++)
51                      {
52                          suma += matriz[i][j];
53                      }
54                  }
55              }
56              wt1 = omp_get_wtime();
57              printf("%d,%d,%d,Filas,%f\n",  hilos,filas, columnas, wt1-
    wt0);
58
59              suma = 0;
60
61              //Accediendo a columnas
62              i = 0, j=0;
63              wt0 = omp_get_wtime();
64              #pragma omp parallel default(none) shared(matriz) private(
    i, j) reduction(+:suma)
65              {
66                  // printf("%d\n", omp_get_num_threads());
67                  #pragma omp for
68                  for(i = 0 ; i < matriz[0].size(); i++)
69                  {
70                      for(j = 0 ; j < matriz.size(); j++)
71                      {
72                          suma += matriz[j][i];
73                      }
```

```
74                      }
75                  }
76                  wt1 = omp_get_wtime();
77
78                  // printf("%d,%d,Columnas,%f\n", filas, columnas, double(
    t1-t0)/CLOCKS_PER_SEC);
79                  printf("%d,%d,%d,Columnas,%f\n", hilos, filas, columnas,
    wt1-wt0);
80
81                  // printf("%d,%d,ColumnasSec,%f\n", filas, columnas,
    double(t1-t0)/CLOCKS_PER_SEC);
82              }
83          }
84      }
85
86
87
88  }
```

2.2.2 Ejecución

La ejecución se llevo a cabo en un Intel® Core™ i5-10400F, que cuenta con 6 núcleos y 12 hilos con hyperthreading. A medida que se iba llevando a cabo las operaciones se iban informando los datos descriptos previamente.

2.2.3 Análisis

Lo primero que hice fue promediar las distintas ejecuciones que había llevado a cabo para así tener una mejor representación.

Teniendo promediados las diferentes ejecuciones calculé ciertos estadísticos para ver si, independientemente del número de hilos, había diferencias significativas entre ambos tipos de acceso.

Acceso	Tiempo medio (s)
Columnas	0.133533
Filas	0.080173

Tabla 1: Tiempo medio de cómputo para los diferentes tipos de acceso

Acceso	Desvio estándar (s)
Columnas	0.182093
Filas	0.107352

Tabla 2: Desvío estándar para los diferentes tipos de acceso

De esta forma, pude observar que los tiempos medios de acceso cuando se realizaban por filas eran mejores y que el desvío era menor.

Luego decidí realizar gráficas del tiempo en función del número de hilos para los distintos número de componentes con los cuáles se habían llevado a cabo las operaciones [1, 2].

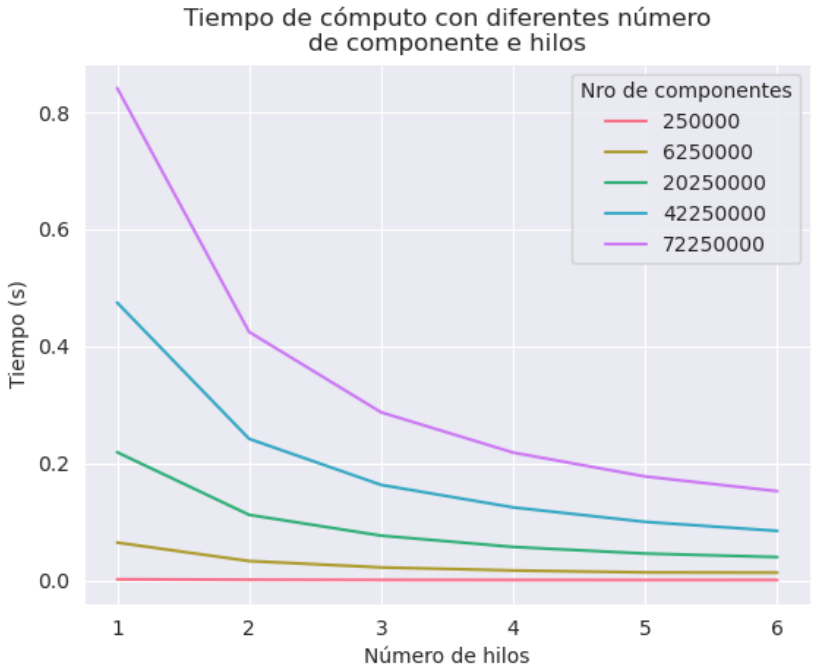

Fig. 1: Tiempos para diferentes nros de hilos

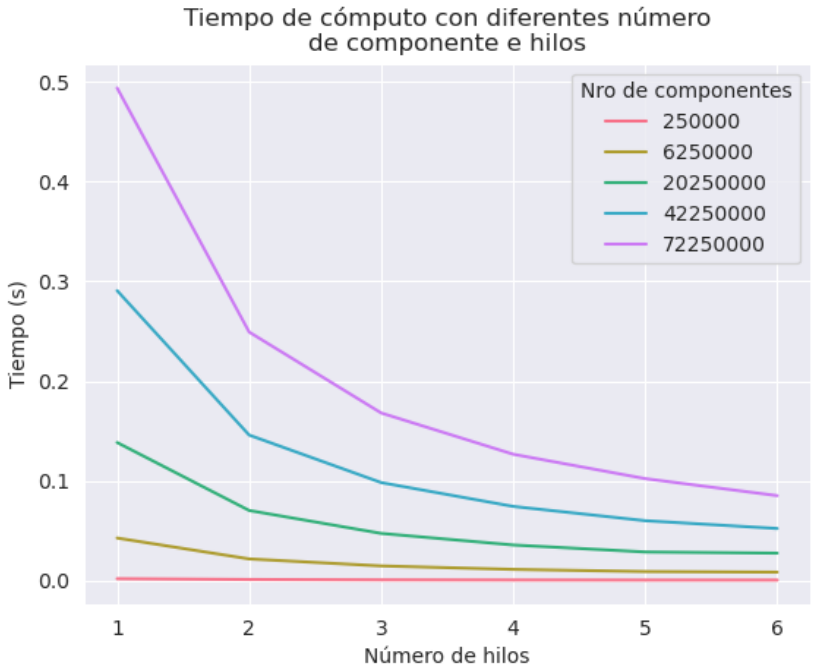

Fig. 2: Tiempos para diferentes nros de hilos

Como era de esperarse los tiempos de cómputo van decayendo con el número de hilos. Sin embargo a medida que aumenta el número de hilos deja de ser tan significativa la diferencia, sobre todo en matrices de menor tamaño. Si bien ya había quedad claro con los estadísticos, aca también se puede observar que se obtienen mejores tiempos accediendo por filas.

Por otro lado, como sabemos, el speedup es un parámetro de alta relevancia a la hora de evaluar la performance de códigos en paralelo. Por ello, realicé el cálculo de la siguiente manera:

$$S = \frac{t_s}{t_p}$$

Habiéndolo calculado procedí a graficarlo de forma similar a cómo realicé para los tiempos

[3, 4].

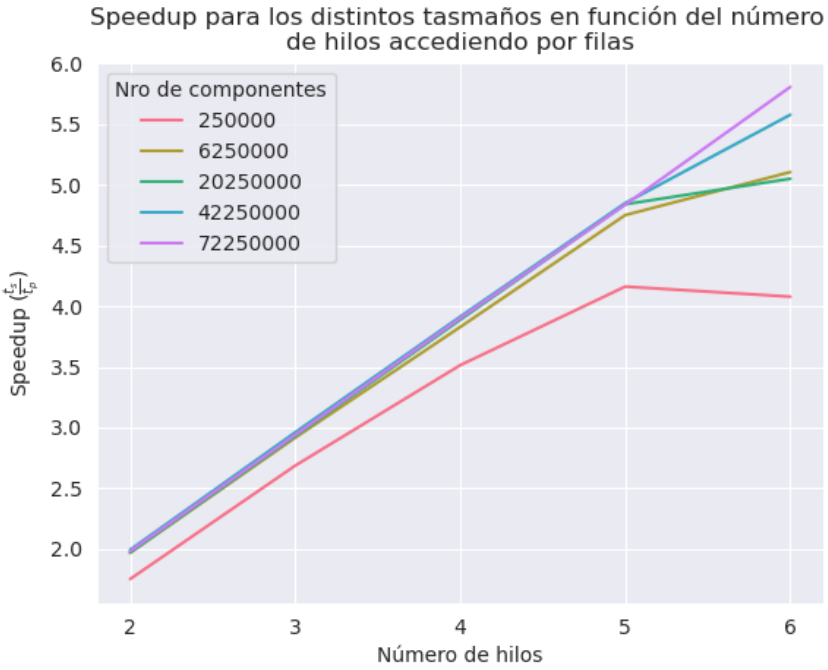

Fig. 3: Speedup para diferentes nros de hilos

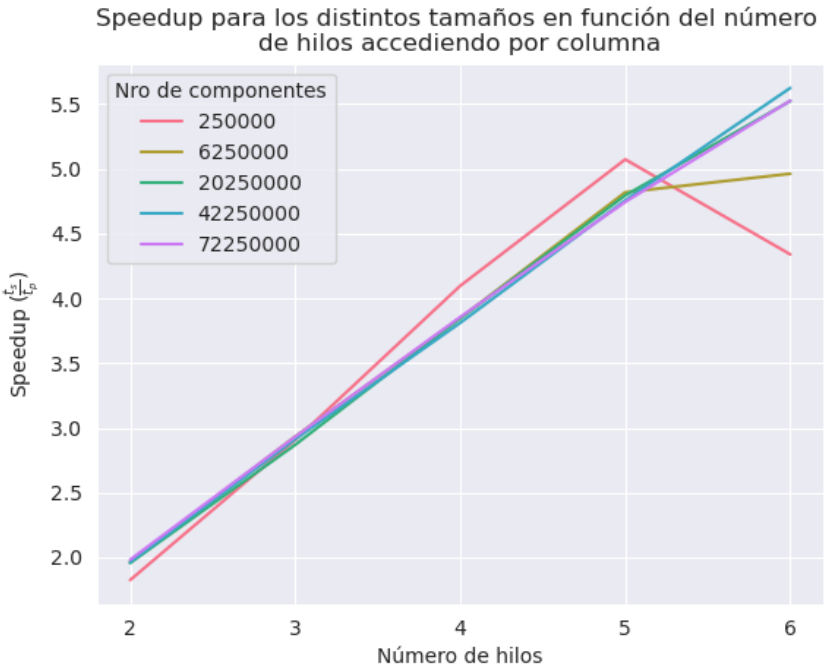

Fig. 4: Speedup para diferentes nros de hilos

En ambas gráficas se puede apreciar que el speedup aumenta a medida que se agregan hilos a la ejecución, y que en los valores más bajos de tamaños de componentes tiende a decaer primero.

3 Ejercicio 2

3.1 Consigna

Implementar una versión paralela del Teorema de los Números Primos para arquitecturas de memoria compartida empleando OpenMP.

1. Describir cuales variables debern ser compartidas y cuales privadas. ¿Por qué?
2. Determinar el speedup para diferentes números de threads.
3. Emplear diferentes esquemas de distribución de carga. Comparar rendimiento y escalabilidad.

3.2 Resolución

3.2.1 Implementación con OpenMP

Para poder paralelizar este problema decidí insertar el ciclo con el cual se hacen las llamadas a la función para checkear si un número es primo o no. Para ello debía estar una región paralela donde decidí que el tratamiento por defecto sea none, tanto el límite hasta donde se checkea como el tamaño del chunk sean compartidas, el iterador n sea privado y se haga la reduction sobre el número de primos.

Este ciclo lo hice dos veces para poder correrlo una vez con un schedule static y en la otra dynamic. Además, definí un vector de chunks para probar con diferentes tamaños de chunk.

```
1 #include <stdio.h>
2 #include <iostream>
3 #include <vector>
4 #include <omp.h>
5 #include <cmath>
6 #include <ctime>
```

```
7
8  using namespace std;
9
10 int esPrimo(int n)
11 {
12     if(n <= 1) return 0;
13
14     int m = int(sqrt(n));
15
16     for (int j = 2 ; j <= m ; j++)
17         if (!(n%j)) return 0;
18     return 1;
19 }
20
21
22 int main(int argc, char **argv)
23 {
24     freopen("salida.csv", "w", stdout);
25     int limite = pow(10, 7), n=2, primes = 0, maxThreads =
       omp_get_max_threads()/2;
26     double wt0, wt1;
27
28     int nroChunks = 6;
29     vector<int> chunks(nroChunks, 10);
30     for(int i = 0 ; i < nroChunks ; i++) chunks[i] = pow(chunks[i], (i+2))
       ;
31
32     printf("Hilos,Schedule,Chunk,Limite,Primes,Tiempo\n");
33
34     int nroIteraciones = 5;
35
36     for(int it = 0 ; it < 3 ; it++)
37     {
38         for(int hilos = 1 ; hilos <= maxThreads ; hilos++)
39         {
40             omp_set_num_threads(hilos);
41             for(int i = 0 ; i < nroChunks ; i++)
42             {
43                 primes = 0;
```

```
44                int chunk = chunks[i];
45                wt0 = omp_get_wtime();
46                #pragma omp parallel default(none) shared(limite, chunk)
   private(n) reduction(+:primes)
47                {
48                    #pragma omp for schedule(static,chunk)
49                    for(n = 2; n <= limite ; n++)
50                    {
51                        if(esPrimo(n)) primes++;
52                    }
53                }
54                wt1 = omp_get_wtime();
55                printf("%d,Static,%d,%d,%d,%f\n",hilos,chunk,limite,primes
   ,wt1-wt0);
56
57
58                primes = 0;
59                wt0 = omp_get_wtime();
60                #pragma omp parallel default(none) shared(limite, chunk)
   private(n) reduction(+:primes)
61                {
62                    #pragma omp for schedule(dynamic,chunk)
63                    for(n = 2; n <= limite ; n++)
64                    {
65                        if(esPrimo(n)) primes++;
66                    }
67                }
68                wt1 = omp_get_wtime();
69                printf("%d,Dynamic,%d,%d,%d,%f\n",hilos,chunk,limite,
   primes,wt1-wt0);
70            }
71        }
72     }
73
74 }
```

3.2.2 Ejecución y análisis

La compilación y ejecución se realizo en mi computadora personal y la salida se almaceno en un archivo csv.

Con los datos de interés registrados, procedí a analizarlos en una notebook de jupyter.

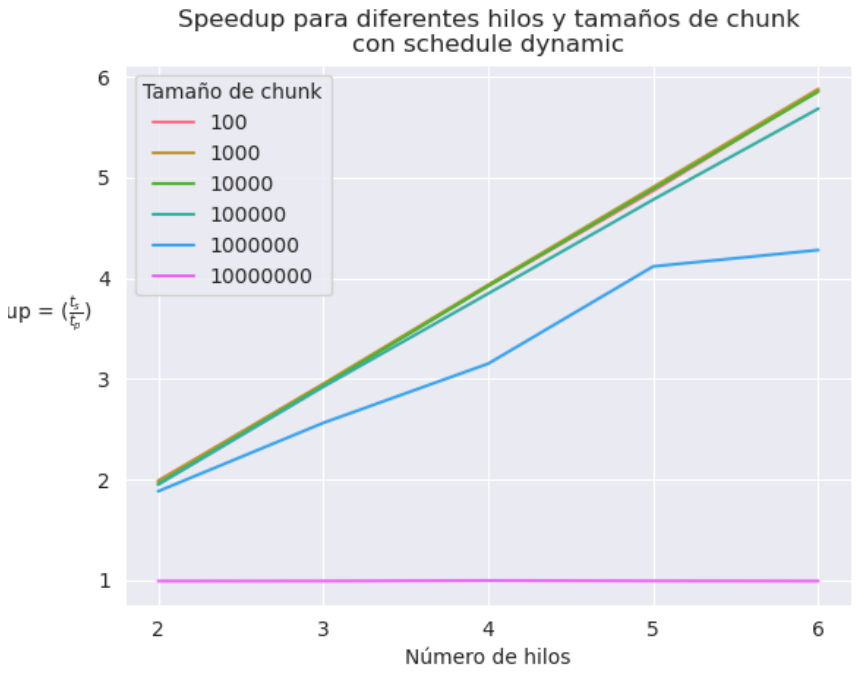

Fig. 5: Speedups para diferentes hilos y tamaños de chunk.

Como se puede apreciar en la figura 5 el speedup para tamaños de chunk más chicos aumenta considerablemente, mientras que se mantiene aproximadamente en uno para el tamaño más grande de chunk.

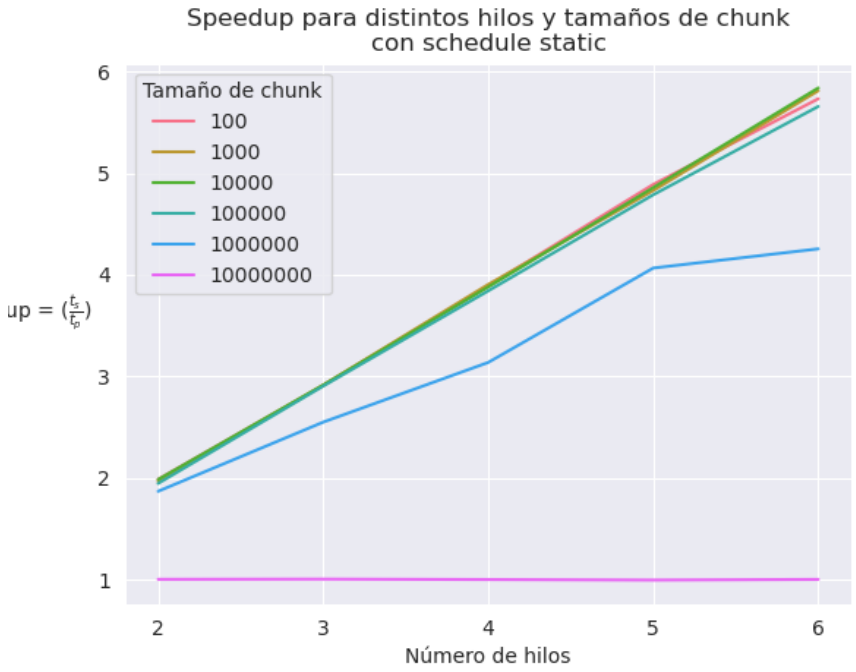

Fig. 6: Eficiencia para diferentes hilos y tamaños de chunk.

De la imagen 6 podemos observar que con el algoritmo static se obtiene un comportamiento similar a con dynamic.

4 Ejercicio 3

4.1 Consigna

Implementar un código utilizando OpenMP que efectúe el producto de dos matrices densas en paralelo.

1. Describir cuales variables deben ser compartidas y cuales privadas. ¿Por qué?

2. Determinar el speedup para diferentes número de threads y para diferentes tamaños de matrices.

4.2 Resolución

4.2.1 Implementación con OpenMP

Para realizar el producto entre dos matrices densas decido primero rellanarla y este será el primer ciclo que decido paralelizar con OpenMP. En él decidí que sean compartidas ambas matrices y los iteradores privados, como omití la cláusila default los tamaños de las matrices también serán compartidos.

Luego decidí paralelizar el ciclo que recorre ambas matrices y la resultante realizando el producto. En este caso agregué la clausula default para tener un mejor control de la privacidad de los datos. Por ello tuve que hacer compartidas las tres matrices, sin estrategias de conciliación porque van a estar accediendo a porciones diferentes y privados los tres iteradores para tener mejor control de que los ciclos for se paralelicen correctamente.

```
1 #include <stdio.h>
2 #include <iostream>
3 #include <vector>
4 #include <omp.h>
5 #include <ctime>
```

```
6
7  using namespace std;
8
9  int main(int argc, char **argv)
10 {
11     freopen("salida.csv", "w", stdout);
12     int num_iteraciones = 5, maxThreads = omp_get_max_threads()/2;
13     vector<int> lista_de_filas(num_iteraciones, 500);
14     vector<int> lista_de_columnas(num_iteraciones, 500);
15
16     for(int i = 0 ; i < lista_de_filas.size(); i++)
17     {
18         lista_de_filas[i] = lista_de_filas[i]+(i*200);
19         lista_de_columnas[i] = lista_de_columnas[i]+(i*200);
20     }
21
22     double t0, t1;
23
24     printf("NroThreads,Filas,Columnas,Tiempo\n");
25     for(int it = 0 ; it < 5 ; it ++)
26     {
27         for(int numThreads = 1 ; numThreads <= maxThreads ; numThreads++)
28         {
29             omp_set_num_threads(numThreads);
30             for(int tamanio = 0 ; tamanio < lista_de_filas.size() ;
   tamanio++)
31             {
32                 int filas = lista_de_filas[tamanio];
33                 int columnas = lista_de_columnas[tamanio];
34
35
36                 //Declaro la matriz
37                 vector<vector<int>> matrizA(filas, vector<int>(columnas,
   0));
38                 vector<vector<int>> matrizB(filas, vector<int>(columnas,
   0));
39                 vector<vector<int>> matrizResultante(filas, vector<int>(
   columnas, 0));
40
```

```
41
42              //Relleno la matriz
43              int f, c;
44              #pragma omp parallel shared(matrizA, matrizB) private(f, c
    )
45              {
46                  #pragma omp for
47                  for(f = 0 ; f < filas ; f++)
48                  {
49                      for(c = 0 ; c < columnas ; c++)
50                      {
51                          matrizA[f][c] = f+c+1;
52                          matrizB[f][c] = (filas-f)+(columnas-c)+1;
53                      }
54                  }
55              }
56
57
58              int i, j, k;
59              t0 = omp_get_wtime();
60              #pragma omp parallel default(none) shared(matrizA, matrizB
    , matrizResultante, filas, columnas) private(i, j, k)
61              {
62                  #pragma omp for
63                  for (i = 0; i < filas; i++)
64                  {
65                      for (j = 0; j < columnas; j++)
66                      {
67                          for (k = 0; k < columnas; k++)
68                          {
69                              matrizResultante[i][j] += matrizA[i][k] *
    matrizB[k][j];
70                          }
71                      }
72                  }
73              }
74              t1 = omp_get_wtime();
75              printf("%d,%d,%d,%f\n",numThreads,filas, columnas, t1-t0);
76
```

```
77
78                  // printf("Matriz A:\n");
79                  // for(int i = 0 ; i < filas ; i++)
80                  // {
81                  //     for(int j = 0 ; j < columnas ; j++)
82                  //     {
83                  //         cout<<matrizA[i][j]<<" ";
84                  //     }
85                  //     cout<<endl;
86                  // }
87
88                  // printf("Matriz B:\n");
89                  // for(int i = 0 ; i < filas ; i++)
90                  // {
91                  //     for(int j = 0 ; j < columnas ; j++)
92                  //     {
93                  //         cout<<matrizB[i][j]<<" ";
94                  //     }
95                  //     cout<<endl;
96                  // }
97
98                  // printf("Matriz Resultante:\n");
99                  // for(int i = 0 ; i < filas ; i++)
100                 // {
101                 //     for(int j = 0 ; j < columnas ; j++)
102                 //     {
103                 //         cout<<matrizResultante[i][j]<<" ";
104                 //     }
105                 //     cout<<endl;
106                 // }
107             }
108         }
109     }
110
111
112
113
114 }
```

4.2.2 Ejecución y Análisis

Compilé y ejecuté el código en mi computadora, llevando a cabo el registro en un archivo csv de el número de hilos y tiempos.

Luego procesé los datos obtenidos, calculé el speedup y realicé gráficas.

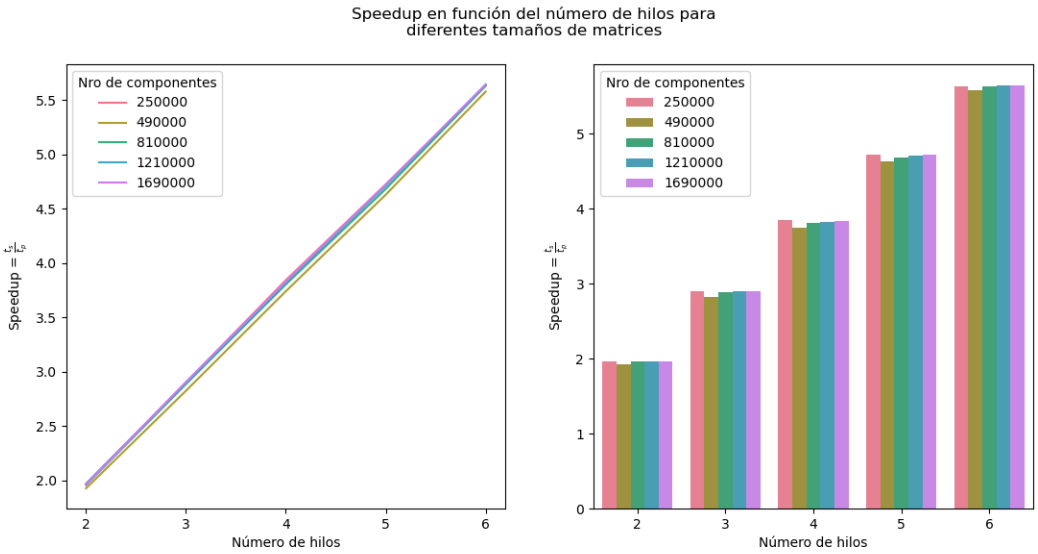

Fig. 7: Speedup para diferentes números de hilos y tamaños matriciales.

La figura 7 permite visualizar que es considerable el speedup obtenido con el aumento del número de hilos, casi independientemente del tamaño que tendrán las matrices.

5 Ejercicio 4

5.1 Consigna

Implementar un código en OpenMP que realice la integración numérica de la función:

$$f(x) = \frac{4}{1+x^2} = \pi$$

en el intervalo $0 \leq x \leq 1$. Emplee diferentes esquemas (reduction, critical, ordered, etc) para sumar las contribuciones parciales de cada thread (hilo) y comparar los speed up y eficiencias obtenidos.

5.2 Resolución

5.2.1 Implementación con OpenMP

Decidí paralelizar el cálculo de la integración con diferentes esquemas:

- Usando la cláusula *reduction* para el área.
- Usando la directiva *atomic*.
- Cláusula *ordered*.

Para todos estos casos fui realizando las mediciones de tiempo correspondientes.

```
#include <stdio.h>
#include <iostream>
#include <vector>
#include <omp.h>
#include <ctime>

using namespace std;

int main(int argc, char **argv)
```

```
{
    freopen("salida.csv", "w", stdout);
    printf("NumThreads,Estrategia,Tiempo\n");
    float n1 = 0.0, n2 = 1.0, particiones = 900000, paso = n2/particiones;
    int maxThreads = omp_get_max_threads()/2;
    float area = 0.0;
    float n = 0.0;

    double t0, t1;

    /*
    Declaro un vector de centros para que OpenMP pueda
    trabajar con el iterador entero.
    */

    int i;
    vector<float> centros(particiones);
    #pragma omp parrallel default(none) shared(centros) private(i)
    {
        #pragma omp for
        for(i = 0 ; i <= centros.size() ; i++)
        {
            centros[i] = n1 + paso*float(i);
        }
    }

    int num_iteraciones = 100;

    for(int hilos = 1 ; hilos <= maxThreads ; ++hilos)
    {
        omp_set_num_threads(hilos);
        for(int it = 0 ; it < num_iteraciones ; it++)
        {

            t0 = omp_get_wtime();
            #pragma omp parallel default(none) shared(centros, paso)
   private(i) reduction(+:area)
            {
                #pragma omp for
```

```
48                for(i = 0 ; i <= centros.size() ; i++)
49                {
50                    area += (1.0/(1.0+(centros[i]*centros[i])))*paso;
51                }
52            }
53            t1 = omp_get_wtime();
54            // printf(",Reduction,%f\n", double(t1-t0)/CLOCKS_PER_SEC);
55            printf("%d,Reduction,%f\n", hilos, t1-t0);
56
57            area = 0.0;
58            t0 = omp_get_wtime();
59            #pragma omp parallel default(none) shared(area,centros, paso)
    private(i)
60            {
61                #pragma omp for
62                for(i = 0 ; i <= centros.size() ; i++)
63                {
64                    #pragma omp atomic
65                    area += (1.0/(1.0+(centros[i]*centros[i])))*paso;
66                }
67            }
68            t1 = omp_get_wtime();
69            printf("%d,Critical,%f\n",hilos, t1-t0);
70
71
72            area = 0.0;
73
74            t0 = omp_get_wtime();
75            #pragma omp parallel default(none) shared(area,centros, paso)
    private(i)
76            {
77                #pragma omp for ordered
78                for(i = 0 ; i <= centros.size() ; i++)
79                {
80                    area += (1.0/(1.0+(centros[i]*centros[i])))*paso;
81                }
82            }
83            t1 = omp_get_wtime();
84            printf("%d,Ordered,%f\n", hilos, t1-t0);
```

```
85
86              }
87      }
88
89
90  }
```

5.2.2 Ejecución

El código fue compilado y ejecutado en la misma computadora descripta previamente. La salida fue capturada en un archivo *.csv* para su posterior manipulación.

5.2.3 Análisis

Con el archivo generado a partir de su ejecución lo analicé desde una notebook de jupyter, realicé los cálculos de speedup y eficiencia y realicé las gráficas correspondientes.

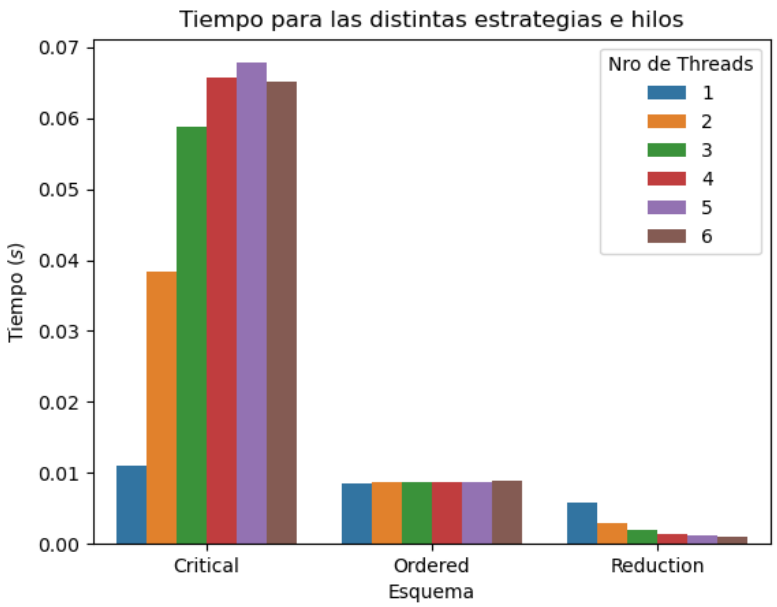

Fig. 8: Tiempos para los diferentes esquemas.

Como se puede apreciar en la figura 8 los tiempos son muy superiores cuando se ejecuta con la directiva critical. Con ordered se mantiene casi constante, lo que tiene sentido porque estamos serializando el código y finalmente con reduction se fueron reduciendo aprovechando los beneficios de la paralelización.

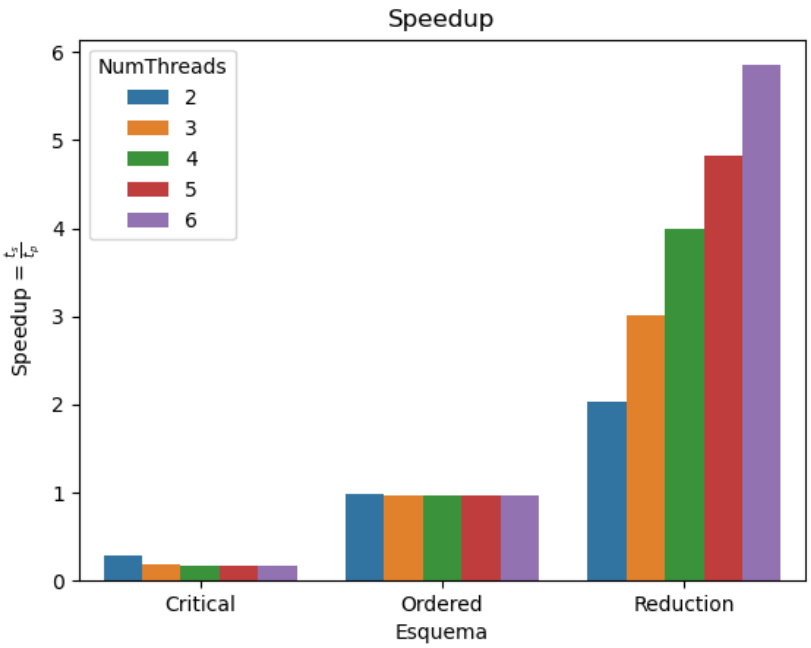

Fig. 9: Speedup para los diferentes esquemas e hilos.

De la figura 9 podemos observar que en critical el speedup es inferior a 1, en ordered se mantiene aproximadamente en 1 y finalmente en reduction sí obtenemos buenos valores de speedup, incrementándose con el número de hilos.

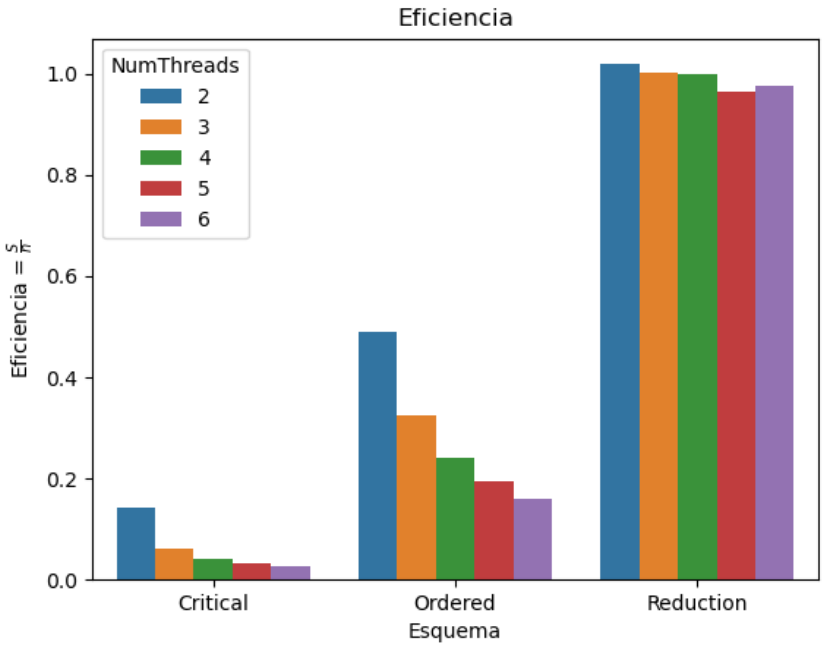

Fig. 10: Eficiencia para los diferentes esquemas e hilos.

Finalmente, en la figura 10 podemos observar la eficiencia para los diferentes esquemas y números de threads, se puede ver que se en los tres se alcanza el máximo con 2 threads y luego empieza a decaer. En los dos primeros es más evidente porque estamos aumentando el número de hilos sin obtener un aumento significativo del speedup [9]