


Practical Session 2

Juan Diego Collazos Mejia

August 23, 2025

Ejercicio 1: Lectura


- Producer and Consumer

 [producer-consumer.c in GitHub](#)

```
[jucollas][fedora][±][main ?:5 X][~/../scripts/exercise-1]
└─ gcc -pthread -o producer-consumer.out producer-consumer.c
[jucollas][fedora][±][main ?:5 X][~/../scripts/exercise-1]
└─ ./producer-consumer.out producer(1000 100) {
  ● Productor produjo: 1 for (int i = 0; i < PRODUCE_COUNT;
  ● Consumidor consumió: 1 int item = i + 1;
  ● Productor produjo: 2
  ● Productor produjo: 3 sem_wait(&empty);
  ● Consumidor consumió: 2 pthread_mutex_lock(&mutex);
  ● Productor produjo: 4
  ● Consumidor consumió: 3 buffer[in] = item;
  ● Productor produjo: 5 printf("● Productor produjo: %d\n",
  ● Consumidor consumió: 4 in = (in + 1) % BUFFER_SIZE;
  ● Consumidor consumió: 5
```

Figure 1: Execution Producer and Consumer


- Matrix x Vector

 [matrix-vector-multiplication.c in GitHub](#)

```
[jucollas][fedora][±][main ?:5 X][~/../scripts/exercise-1]
└─ gcc -pthread -o matrix-vector-multiplication.out matrix-vector-multiplication.c
[jucollas][fedora][±][main ?:5 X][~/../scripts/exercise-1]
└─ ./matrix-vector-multiplication.out 4
Input m y n: 4 4 15 int my_last_1
Input A (4 x 4): 16 for (int i = 0; i < m; i++)
1 2 3 4 17 y[i] = 0;
5 6 7 8 18 for (int j = 0; j < n; j++)
9 10 11 12 19 y[j] = 0;
13 14 15 16 20
Input x (4): 21 return 0;
1 2 3 4 22
Answer: y = A * x: 23
30.000000 24 void mult_matrix(int **A, int m, int n, int *x, int *y)
70.000000 25 pthread_t *th;
110.000000 26 for (long i = 0; i < m; i++)
150.000000 27
```

Figure 2: Execution matrix x vector


- Trapezoidal Rule

 [trapezoidal-rule.c in GitHub](#)

```
[jucollas][fedora][±][main ? :5 X][~/.../scripts/exercise-1]
└─ gcc -pthread -o trapezoidal-rule.out trapezoidal-rule.c
[jucollas][fedora][±][main ? :5 X][~/.../scripts/exercise-1]
└─ ./trapezoidal-rule.out 7
Approximate integral in [0.000000, 12.000000] = 575.998272002016392
```

Figure 3: Execution Trapezoidal Rule

- Count Sort


 [count-sort.c in GitHub](#)

```
[jucollas][fedora][±][main ? :5 X][~/.../scripts/exercise-1]
└─ gcc -pthread -o count-sort.out count-sort.c
[jucollas][fedora][±][main ? :5 X][~/.../scripts/exercise-1]
└─ ./count-sort.out 4
7 8 7 5 6 7 4 5
8 7 5 6 7 4 5
4 5 5 6 7 7 8
```

Figure 4: Execution Count Sort

Ejercicio 2: Suma de un Arreglo Grande

Pthread implementacion


 [adder_array_with_pthread.cpp in GitHub](#)

```
[jucollas][fedora][±][main U :3 ? :8 X][~/.../scripts/exercise-2]
└─ g++ -pthread -o adder_pthread.out adder_array_with_pthread.cpp
[jucollas][fedora][±][main U :3 ? :8 X][~/.../scripts/exercise-2]
└─ ./randomizer.out 1000000 | ./adder_pthread.out 2
ANS: 4496433
CPU time: 0.01113 seconds
```

Figure 5: Execution SumArray Pthread

Size of array (N)	Threads	CPU Time (s)
1e6	2	0.000948
1e6	4	0.001389
1e6	8	0.001574
1e7	2	0.004705
1e7	4	0.007255
1e7	8	0.013487
1e8	2	0.039386
1e8	4	0.061709
1e8	8	0.100795

OpenMP implementacion

 `adder_array_with_openMP.cpp` in GitHub

```
[jucollas][fedora][±][main U:3 ?:8 X][~/.../scripts/exercise-2]
└─ g++ -fopenmp -o adder_openMP.out adder_array_with_openMP.cpp
[jucollas][fedora][±][main U:3 ?:8 X][~/.../scripts/exercise-2]
└─ ./randomizer.out 1000000 | ./adder_openMP.out 2
4497395
CPU time: 0.008287 seconds
```

Figure 6: Execution SumArray Openmp

Size of array (N)	Threads	CPU Time (s)
1e6	2	0.007395
1e6	4	0.013883
1e6	8	0.019693
1e7	2	0.070266
1e7	4	0.106531
1e7	8	0.14082
1e8	2	0.668011
1e8	4	0.721224
1e8	8	1.31289

Comparacion Implementaciones

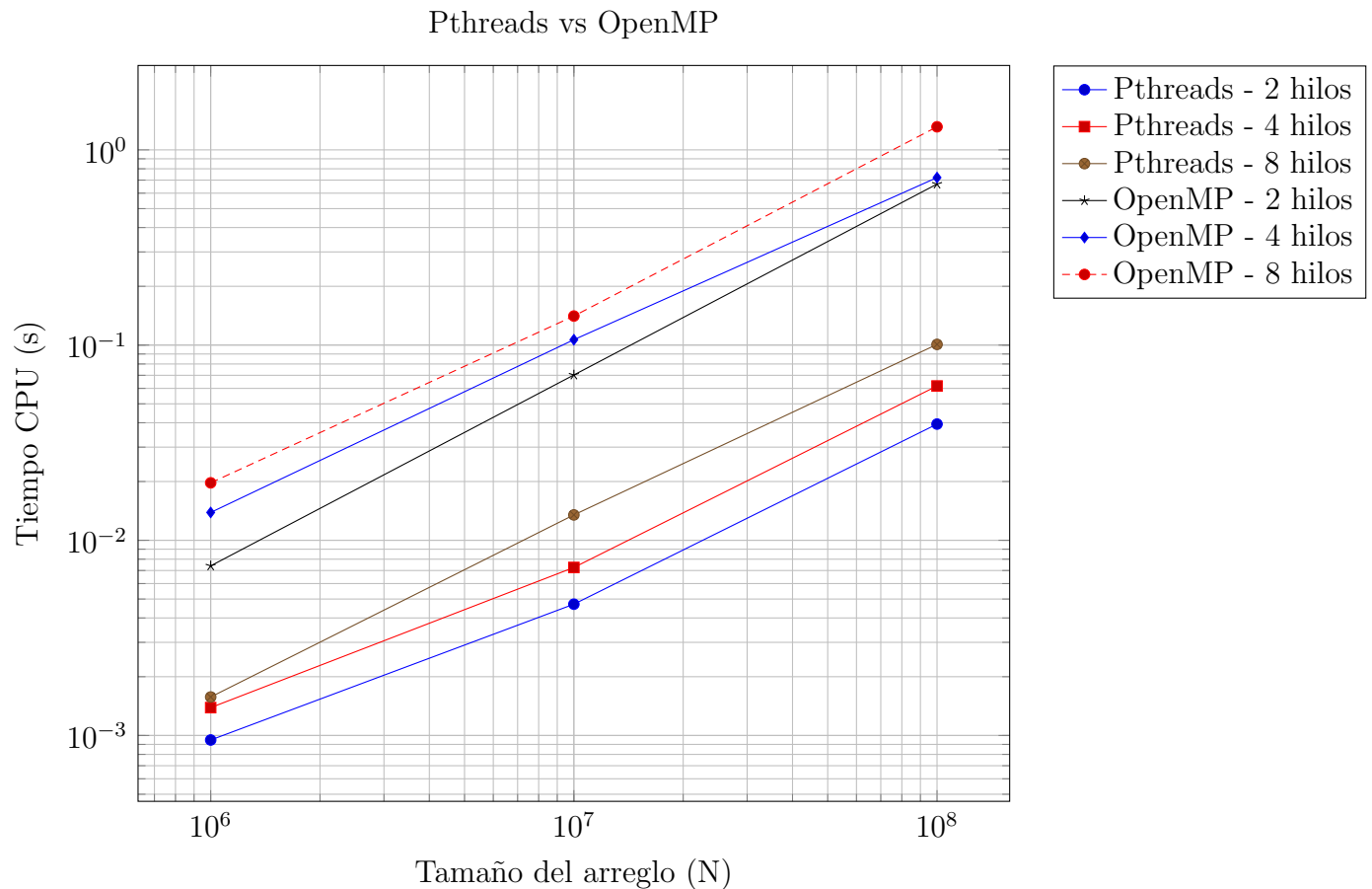


Figure 7: Comparación del tiempo de ejecución entre Pthreads y OpenMP con distintos hilos

Conclusión

A partir de los resultados obtenidos se observa que la implementación con **Pthreads** presenta tiempos de ejecución considerablemente menores frente a la implementación con **OpenMP**, para los tres tamaños de entrada analizados.


En particular, para un arreglo de 10^8 elementos, **Pthreads** logra tiempos del orden de 0.04 a 0.10 segundos dependiendo de la cantidad de hilos, mientras que **OpenMP** requiere entre 0.66 y 1.31 segundos. Esto implica que, en este escenario, la eficiencia de **Pthreads** es entre 6 y 15 veces superior.

Además, se aprecia que al incrementar el número de hilos no siempre se obtiene una reducción proporcional del tiempo de ejecución. De hecho, tanto en **Pthreads** como en **OpenMP**, el incremento de hilos de 2 a 8 tiende a aumentar el tiempo de ejecución en lugar de disminuirlo, lo cual puede atribuirse a la *sobrecarga de sincronización* y a la gestión de

hilos.

En conclusión, la implementación con **Pthreads** resulta más eficiente para este problema específico de suma de arreglos, mientras que **OpenMP** muestra un mayor costo en términos de tiempo de CPU conforme aumenta el tamaño de entrada y el número de hilos.

Ejercicio 3: Multiplicación de Matrices

 `mult_matrix_parallel.cpp` in GitHub

```
[jucollas][fedora][±][main U:2 ? :8 X][~/.../scripts/exercise-3]
└─ g++ matrix_randomizer.cpp -o randomizer
[jucollas][fedora][±][main U:2 ? :8 X][~/.../scripts/exercise-3]
└─ g++ -pthread mult_matrix_parallel.cpp -o mult
[jucollas][fedora][±][main U:2 ? :8 X][~/.../scripts/exercise-3]
└─ (./randomizer 3 3; ./randomizer 3 3) | ./mult
3 3
73803 218901 122924
443639 1155376 1026136
263413 636008 666344
CPU time: 0.000515 seconds
```

Figure 8: Execution MatrixMult