Reporte técnico: Taller 03

Taller de Sistemas Operativos
Escuela de Ingeniería Informática

Alejandro Rudolphy Fernández

alejandro.rudolphy@alumnos.uv.cl

Resumen. *En el presente informe se explicará el desarrollo de un problema que será resuelto usando programación paralela, con el fin de dividir el problema en partes independientes y, si es posible, lograr una ejecución más eficiente que su contraparte secuencial. Además, se mostrará el diseño de la solución utilizando diseños de alto nivel. Por otro lado, se explicará la metodología utilizada, y se plantearán hipótesis en base al diseño que luego será comprobada en la sección de resultados.*

1 Introducción

Se necesita la implementación de un programa que llene un arreglo de números aleatorios y luego los sume de manera paralela, usando OpenMP. OpenMP, abreviación de Open Multi-Processing, es una interfaz de programación de aplicaciones para la programación multiproceso de memoria compartida en múltiples plataformas, OpenMP se basa en el modelo **fork-join**, paradigma que proviene de los sistemas Unix, donde una tarea muy pesada se divide en K hilos (fork) con menor peso, para luego "recolectar" sus resultados al final y unirlos en un solo resultado (join)[1].

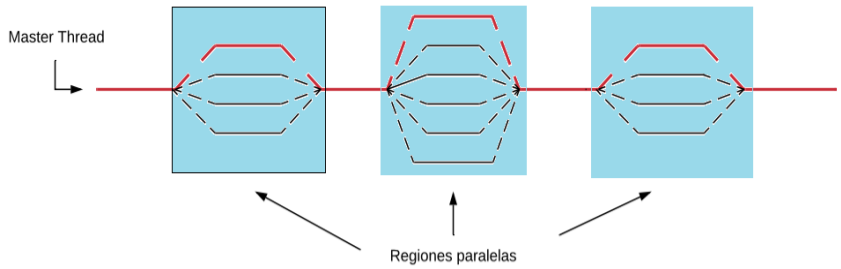

Figura 1 – Modelo de programación en OpenMP.

Para esta experiencia se utilizó una máquina virtual creada utilizando el software de virtualización VirtualBox, con sistema operativo Ubuntu versión 18.04. La programación paralela utiliza simultáneamente múltiples elementos de procesamiento para resolver un problema [2], por eso, para esta experiencia es necesario que la máquina virtual posea múltiples núcleos, esto se puede revisar usando el comando **nproc –all**. En la Figura 2 se muestra la cantidad de procesadores usados para el desarrollo de este informe.

```
arudolphy@tso2020:~$ nproc --all
4
```

Figura 2 – Procesadores de la máquina virtual usada.

1.1 Descripción del problema

El objetivo de este proyecto fue la implementación de un programa que llene un arreglo de números enteros y luego los sume. Ambas tareas deben realizarse de forma paralela, implementadas con OpenMP.

Posterior a esto, se realizarán pruebas de desempeño sobre este programa, para estudiar su funcionamiento en base a los parámetros ingresados. El programa funcionará en base a parámetros ingresados por el usuario vía consola. En la Figura 3 se puede observar la estructura de la forma de uso del comando utilizado para el funcionamiento del programa, y en la Tabla 1, el significado de cada uno de los parámetros.

```
./sumArray -N <num> -t <num> -l <num> -L <num> [-h]
```

Figura 3 – Forma de uso del comando.

PARÁMETRO	SIGNIFICADO	EJEMPLO
-N	Tamaño del arreglo.	1000000
-t	Número de threads.	4
-l	Límite inferior del rango aleatorio de los números.	1
-L	Límite superior del rango aleatorio de los números.	100
-h	Muestra la ayuda de uso y termina.	30

Tabla 1 – Descripción de la forma de uso.

1.1.1 Ejemplo de uso

```
./sumArray -N 10000000 -t 4 -l 1 -L 100 [-h]
```

Figura 4 – Ejemplo de uso del comando.

1.2 Objetivo

El primer objetivo es lograr la implementación correcta de OpenMP, seguido de esto, la realización de un programa que cumpla los requerimientos. El programa se dividirá en dos módulos:

- Módulo de llenado: El programa recibirá un archivo vacío, de tamaño previamente ingresado por el usuario, donde dividirá el trabajo en múltiples threads, y los llenará de manera paralela. La salida de este módulo es un archivo lleno de números enteros aleatorios del tipo **unit32_t,** formado por la unión de los archivos creados en los threads usados anteriormente.

- Módulo de suma: Recibe el arreglo llenado por el módulo anterior, y lo vuelve a separar en múltiples arreglos dependiendo de la cantidad predeterminada de threads. Luego, suma paralelamente estos arreglos, para finalmente juntar todos los valores de esas sumas parciales en una suma total.

Además, el trabajo incluye pruebas de desempeño de la solución para comprobar si efectivamente es más eficiente hacer el programa paralelamente.

1.3 Estructura del documento

Este documento se compone, en primera instancia, de una introducción que incluye una descripción detallada de las tecnologías a utilizar, además del problema, junto a los objetivos de este. Seguido a esto, está la sección de diseño, dónde se explica el problema de una manera general para luego descomponerlo y explicarlo separadamente. En la sección tres se encontrarán los resultados de la experiencia, junto a sus pruebas de desempeño correspondiente, y, finalmente, en la sección cuatro y cinco se encontrarán las conclusiones de la experiencia y las referencias usadas en el documento.

2 Diseño.

2.1 Metodología

La metodología utilizada para esta experiencia consiste en el análisis preciso de los requerimientos, lograr un diseño óptimo del funcionamiento del programa, donde se puedan observar los procesos a realizar para lograr los resultados esperados. Una vez entendido el diseño general de la solución, se pueden observar dos tareas que sobresalen del diseño general, que servirán para la resolución del trabajo. Lo óptimo es dividir el diseño en dos módulos, el módulo de llenado y el módulo de suma que serán explicados más adelante.

2.2 Diseño general de la solución

El programa será compuesto por dos módulos, como se observa en la Figura 5, el primer módulo será la parte de llenado, donde el programa recibirá un arreglo de tamaño predeterminado por el usuario y lo dividirá en múltiples threads, configurados previamente por el usuario. Cada thread recibirá un "trozo" de arreglo y lo llenará de números aleatorios del tipo **unit32_t,** para finalmente juntarlo en uno solo y dar fin al primer módulo**.** El segundo módulo, consistirá en la suma de los valores del arreglo llenado previamente, para esto, después de recibir el arreglo, lo volverá a dividir en múltiples threads que se encargarán de hacer sumas parciales en cada "trozo" de arreglo, para finalmente sumar esas sumas parciales y obtener el resultado.

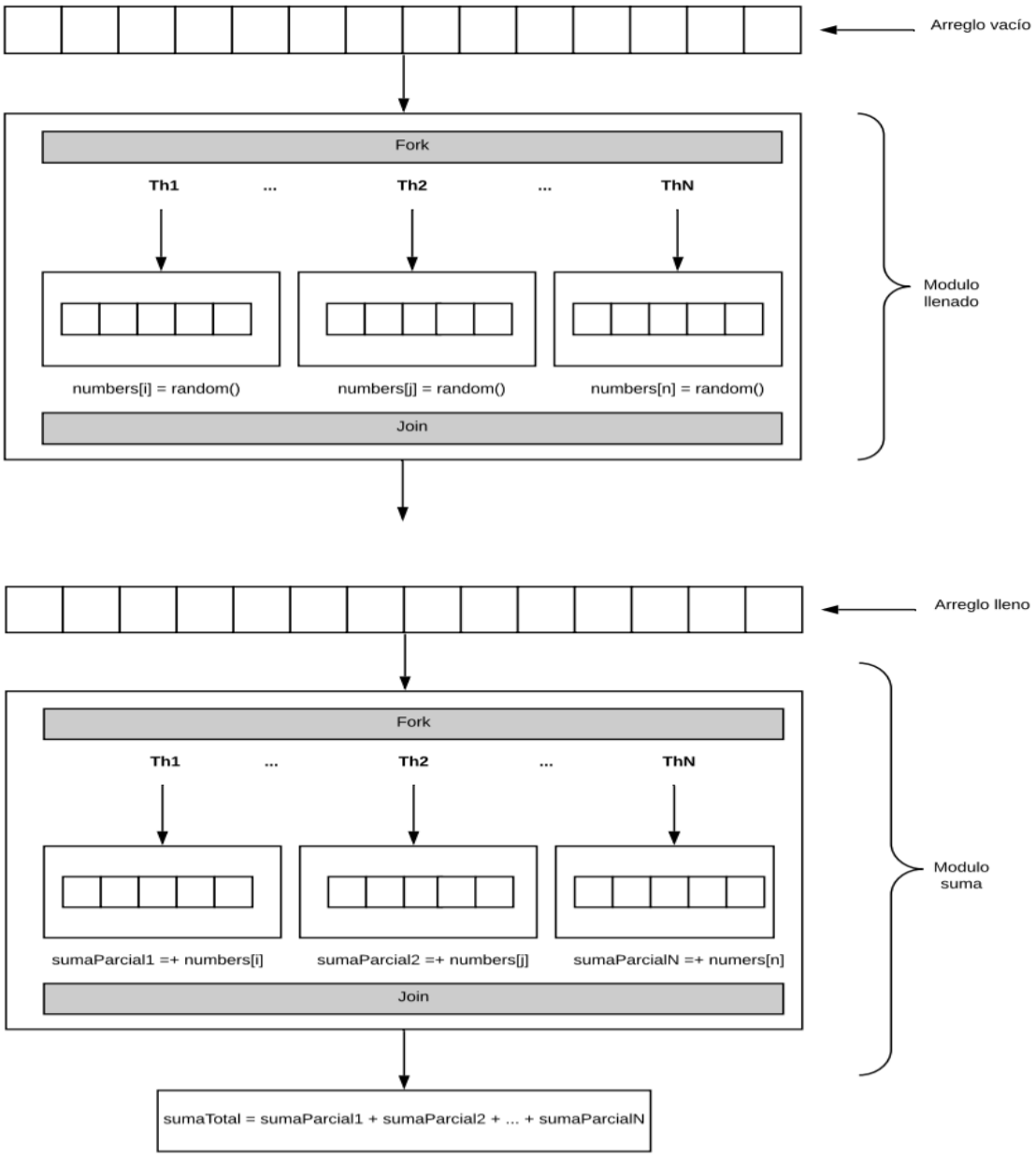

Figura 5 – Diseño general de la solución.

2.3 Módulo de llenado

El módulo de llenado será realizado por una función **fillArray**, esta debe poder recibir los siguientes parámetros:

- Número de elementos
- Cantidad de threads
- Límite inferior del rango de números aleatorios
- Límite superior del rango de números aleatorios

Con esto completado, la función debe poder llenar el arreglo de manera secuencial y paralela al ejecutarse. En la Figura 6 se puede observar el diagrama de componentes de la función y en la Figura 7 el módulo en detalle.

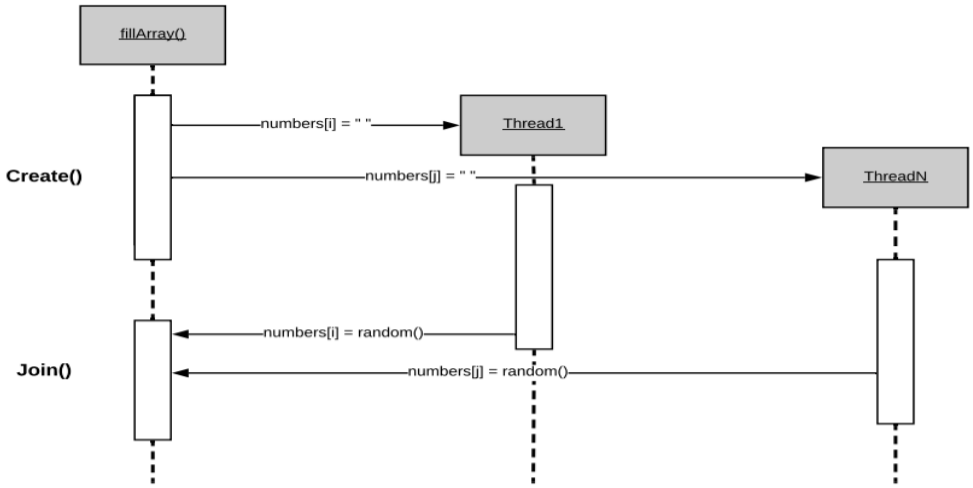

Figura 6 – Diagrama de secuencia módulo de llenado.

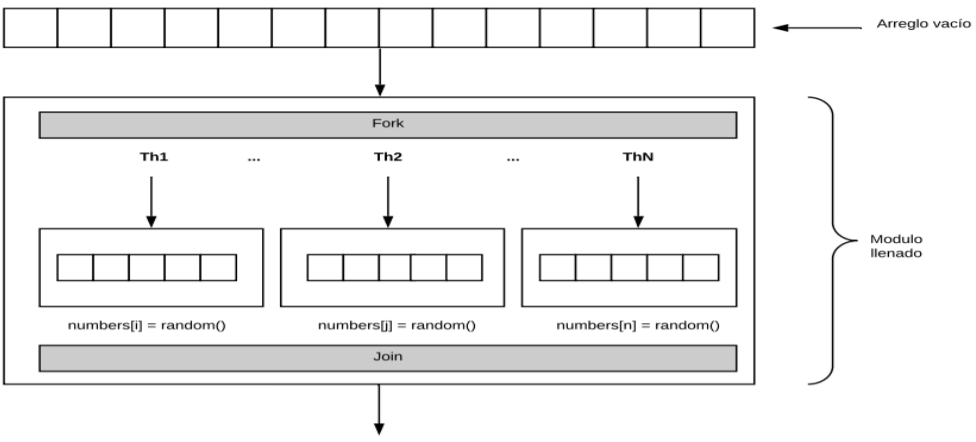

Figura 7 – Diagrama del módulo llenado.

2.4 Módulo de suma

El módulo suma será realizado por una función **sumaParcial**, esta debe ser capaz de recibir lo siguiente:

- Arreglo llenado previamente en el módulo llenado
- Cantidad de threads ingresada previamente para el módulo llenado

Con esto completado, la función debe ser capaz de realizar una suma del arreglo, de manera secuencial y paralela, usando sumas parciales divididas en n threads en esta última. En las Figuras 8 y 9 se pueden observar diagramas de esta función.

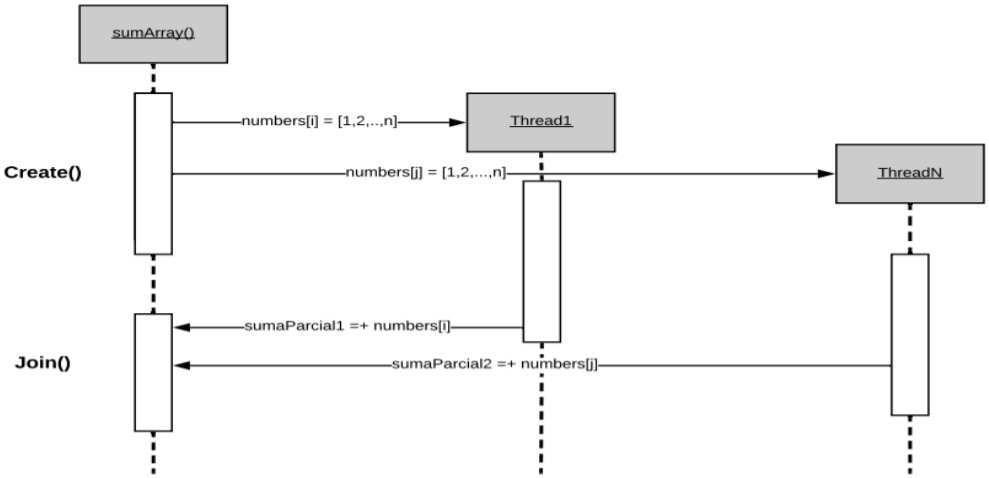

Figura 8 – Diagrama secuencial módulo de suma.

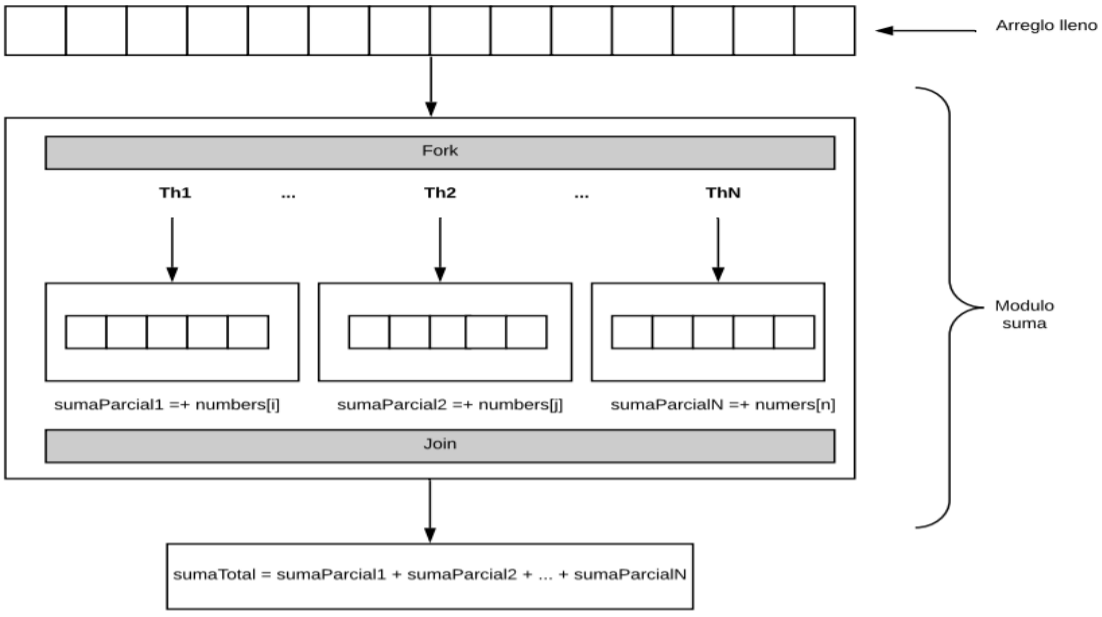

Figura 9 – Diagrama del módulo suma.

3 Implementación

En las subsecciones siguientes 3.1 y 3.2 serán explicadas las funciones realizadas para los diseños vistos en 2.1 y 2.2 respectivamente, además de las diferencias en cómo se logra el trabajo secuencial y paralela con estas.

3.1 Función fillArray

```
void fillArray(size_t beginIdx, size_t endIdx, size_t limInferior, size_t limSuperior){
	std::random_device device;
	std::mt19937 rng(device());
	std::uniform_int_distribution<> unif(limInferior,limSuperior);

	for(size_t i = beginIdx; i < endIdx; i++){
		arreglo[i] = unif(rng);
	}
}
```

Figura 10 – Implementación de la función fillArray.

```
//Etapa de llenado

//Secuencial
arreglo = new uint64_t[totalElementos];

auto start = std::chrono::high_resolution_clock::now();

for(size_t i = 0; i < totalElementos; ++i){
	arreglo[i] = unif(rng);
}

auto end = std::chrono::high_resolution_clock::now();
auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
auto tiempoLlenadoTotal_S = elapsed.count();

delete[] arreglo;
//Paralela
arreglo = new uint64_t[totalElementos];

start = std::chrono::high_resolution_clock::now();
//Directiva de formato de OpenMP
#pragma omp parallel for num_threads(numThreads)
for(size_t i=0; i < totalElementos; ++i){
	arreglo[i] = unif(rng);
}

end = std::chrono::high_resolution_clock::now();
elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end-start);
auto tiempoLlenadoTotal_P = elapsed.count();
```

Figura 11 – Implementación del llenado paralelo con OpenMP.

Como se puede observar, las implementaciones varían en que la implementación paralela posee la directiva de formato de OpenMP, **pragma omp parallel for num_threads(numThreads)**, donde se define el número de threads que serán usados a la hora de ejecutar el programa.

3.2 Función sumaParcial

```
void fillArray(size_t beginIdx, size_t endIdx, size_t limInferior, size_t limSuperior){
	std::random_device device;
	std::mt19937 rng(device());
	std::uniform_int_distribution<> unif(limInferior,limSuperior);

	for(size_t i = beginIdx; i < endIdx; i++){
		arreglo[i] = unif(rng);
	}
}
```

Figura 12 – Implementación de la función fillArray.

```
//Etapa de suma

//Secuencial
uint64_t sumaSecuencial=0;

start = std::chrono::high_resolution_clock::now();

for(size_t i = 0; i < totalElementos; ++i){
	sumaSecuencial += arreglo[i];
}

end = std::chrono::high_resolution_clock::now();
elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
auto tiempoTotalSuma_S = elapsed.count();

//Paralela
uint64_t sumaParalela=0;
start = std::chrono::high_resolution_clock::now();

//Directiva de formato de OpenMP
#pragma omp parallel for reduction(+:sumaParalela) num_threads(numThreads)
for(size_t i=0; i<totalElementos; ++i){
	sumaParalela += arreglo[i];
}

end     = std::chrono::high_resolution_clock::now();
elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
auto tiempoTotalSuma_P = elapsed.count();
```

Figura 13 – Implementación de la manera secuencial y paralela de la función sumaParcial.

De la misma manera que el módulo de llenado, para la suma se debe declarar primero la cantidad de threads en los cuales se ejecutará el código.

4 Resultados

Para la ejecución del programa, es recomendable ejecutar el comando **make clean**, make clean elimina los archivos objeto y realiza una compilación desde cero, seguido por un **make**, para compilar el programa[2].

Para una buena muestra de los resultados, se ejecutó el programa con distintos parámetros de entrada, mostrando por pantalla el tiempo de ejecución de ambas soluciones, secuenciales y paralelas para cada módulo.

4.1 Hipótesis en base a la cantidad de threads

Se espera que mientras más threads disponga el usuario para la solución, menor sea el tiempo de ejecución de la solución paralela para cada uno de los módulos. Para esta hipótesis se harán tres pruebas, donde se asignarán un, dos y cuatro threads a cada una y en cada una se utilizarán diez millones de elementos. En las Figuras 14, 15 y 16 se mostrarán los resultados de estos experimentos.

```
arudolphy@tso2020:~/TSSOO-Taller03$ ./sumArray -N 10000000 -t 1 -l 1 -L 100 [-h]
Elementos: 10000000
Threads: 1
Limite inferior: 1
Limite superior: 100
Resultados suma:

Suma secuencial total: 504909081
Suma en paralelo total: 504909081

Tiempos de ejecución módulo de llenado:

Tiempo Llenado Secuencial: 178[ms]
Tiempo Llenado Paralelo: 177[ms]

Tiempos de ejecución módulo de suma:

Tiempo Suma Secuencial: 7[ms]
Tiempo Suma Paralela: 9[ms]
```

Figura 14 – Prueba de desempeño con 1 thread.

Como era de esperar, el tiempo de ejecución con un thread no varía substancialmente en comparación con la ejecución secuencial.

```
arudolphy@tso2020:~/TSSOO-Taller03$ ./sumArray -N 10000000 -t 2 -l 1 -L 100 [-h]
Elementos: 10000000
Threads: 2
Limite inferior: 1
Limite superior: 100
Resultados suma:

Suma secuencial total: 505100461
Suma en paralelo total: 505100461

Tiempos de ejecución módulo de llenado:

Tiempo Llenado Secuencial: 176[ms]
Tiempo Llenado Paralelo: 114[ms]

Tiempos de ejecución módulo de suma:

Tiempo Suma Secuencial: 10[ms]
Tiempo Suma Paralela: 6[ms]
```

Figura 15 – Prueba de desempeño con 2 threads.

Para la segunda prueba se utilizaron dos threads, y como se puede observar, el tiempo de llenado se redujo a la mitad en comparación a la contraparte secuencial, esto da un indicio de que la hipótesis propuesta inicialmente es correcta.

```
arudolphy@tso2020:~/TSSOO-Taller03$ ./sumArray -N 10000000 -t 4 -l 1 -L 100 [-h]
Elementos: 10000000
Threads: 4
Limite inferior: 1
Limite superior: 100
Resultados suma:

Suma secuencial total: 505000599
Suma en paralelo total: 505000599

Tiempos de ejecución módulo de llenado:

Tiempo Llenado Secuencial: 176[ms]
Tiempo Llenado Paralelo: 93[ms]

Tiempos de ejecución módulo de suma:

Tiempo Suma Secuencial: 9[ms]
Tiempo Suma Paralela: 6[ms]
```

Figura 16 – Prueba de desempeño con 4 threads.

En la tercera prueba ya es posible observar un patrón, donde mientras más threads haya, el tiempo de llenado del arreglo va disminuyendo aún más.

4.1.1 Conclusiones sobre la hipótesis 4.1.

En base a los resultados obtenidos, en la Figura 16 se muestra un gráfico del experimento, este gráfico puede comprobar que la hipótesis propuesta en un comienzo es correcta, dado que, efectivamente, mientras más threads sean dispuestos para el programa, menor será su tiempo de ejecución en el módulo de llenado y por ende su tiempo de ejecución total. Si bien en el tiempo del módulo de suma no se presenta una diferencia substancial, una hipótesis al respecto podría ser que es debido a que la tarea es muy pequeña y con una mayor cantidad de elementos se pudiese observar alguna diferencia entre los tiempos del módulo suma.

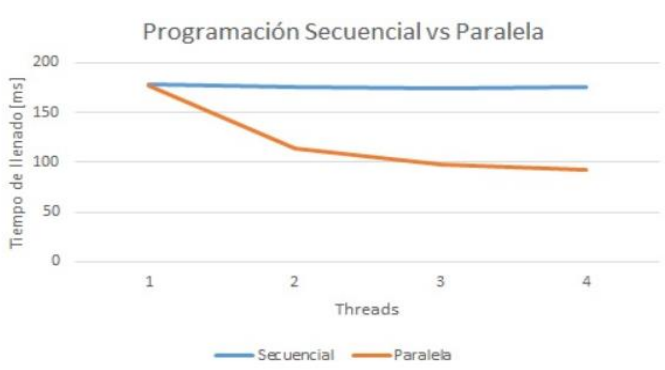

Figura 17 – Gráfico del experimento.

4.2 POSIX Threads vs OpenMP.

Si bien existen múltiples lenguajes de programación y frameworks que hacen posible que se ejecute un código de manera paralela en diferentes threads, esta comparación se hará entre dos de los frameworks existentes para el lenguaje C++, POSIX Threads (PThreads) y OpenMP. La comparación se hará en base al rendimiento de cada uno, para esta comparación se usará un código hecho con PThreads previamente preparado. La hipótesis inicial, es que una implementación llevada a cabo en OpenMP, si bien es una API de más alto nivel que PThreads[4], su tiempo de ejecución no debería variar de una manera substancial como para representar que una es más eficiente que la otra, en las Figuras 18, 19 y20 se muestran ejecuciones del mismo código hecho con PThreads.

```
arudolphy@tso2020:~/TSSOO-Taller02$ ./sumArray -N 10000000 -t 1 -l 1 -L 100 [-h]
Elementos: 10000000
Threads: 1
Límite inferior: 1
Límite superior: 100
Suma secuencial total: 504897774
Suma en paralelo total: 504897774

Tiempos de ejecución etapa de llenado:

Tiempo Llenado Secuencial: 180[ms]
Tiempo Llenado Paralelo: 188[ms]

Tiempos de ejecución etapa de suma:

Tiempo Suma Secuencial: 10[ms]
Tiempo Suma Paralela: 7[ms]
```

Figura 18 – Pruebas de desempeño con 1 thread en PThreads.

De la misma manera que la implementación con OpenMP, los resultados son los esperados ya que con un thread no debería existir una diferencia substancial entre las ejecuciones. Y en comparación con OpenMP, el tiempo de llenado con PThreads demoró 11[ms] menos.

```
arudolphy@tso2020:~/TSSOO-Taller02$ ./sumArray -N 10000000 -t 2 -l 1 -L 100 [-h]
Elementos: 10000000
Threads: 2
Límite inferior: 1
Límite superior: 100
Suma secuencial total: 505171462
Suma en paralelo total: 505171462

Tiempos de ejecución etapa de llenado:

Tiempo Llenado Secuencial: 190[ms]
Tiempo Llenado Paralelo: 103[ms]

Tiempos de ejecución etapa de suma:

Tiempo Suma Secuencial: 8[ms]
Tiempo Suma Paralela: 8[ms]
```

Figura 19 – Pruebas de desempeño con 2 threads en PThreads.

En esta etapa ya se empieza a presentar una diferencia entre los tiempos de ejecución paralela y secuencial, pero, la ejecución demoró 9[ms] menos en comparación a su ejecución con OpenMP.

```
arudolphy@tso2020:~/TSSOO-Taller02$ ./sumArray -N 10000000 -t 4 -l 1 -L 100 [-h]
Elementos: 10000000
Threads: 4
Límite inferior: 1
Límite superior: 100
Suma secuencial total: 504976968
Suma en paralelo total: 504976968

Tiempos de ejecución etapa de llenado:

Tiempo Llenado Secuencial: 199[ms]
Tiempo Llenado Paralelo: 80[ms]

Tiempos de ejecución etapa de suma:

Tiempo Suma Secuencial: 11[ms]
Tiempo Suma Paralela: 10[ms]
```

Figura 20 – Pruebas de desempeño con 4 threads en PThreads.

Finalmente, para cuatro threads PThreads volvió a ser más eficiente en su ejecución, demorando 10[ms] menos.

4.2.1 Conclusiones sobre la hipótesis 4.2

En comparación con las Figuras 14, 15 y 16, donde están los tiempos de ejecución del programa ejecutado en OpenMP, se observan los siguientes resultados en base al tiempo de llenado.

THREADS	PTHREADS	OPENMP
1	188	177
2	103	114
3	90	97
4	80	90

Tabla 2 – Comparación en base al tiempo de llenado[ms].

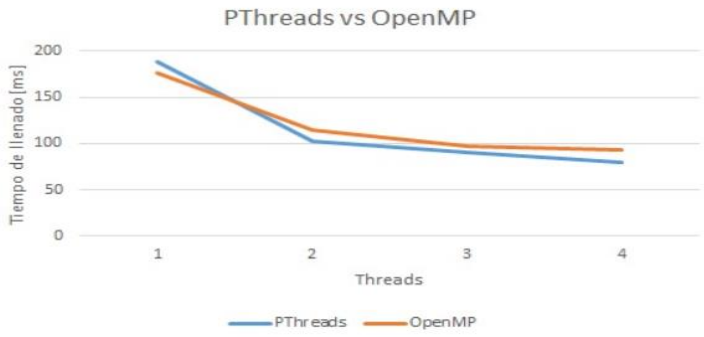

Figura 21 – Gráfico del experimento.

En base a los resultados, si bien se observa una pequeña diferencia en favor a la ejecución con PThreads, no se puede considerar substancial al no ser algo que demuestre en su totalidad que PThreads sea más eficiente que OpenMP, por lo que es correcto asumir que la hipótesis planteada inicialmente es correcta.

5 Conclusiones

En base a lo visto en este documento, es correcto asumir que la programación paralela, para este tipo de tareas, es más eficiente que la programación secuencial. Esto fue comprobado mediante pruebas de desempeño y experimentos sobre un programa que realiza pruebas sobre ambos tipos de programación; fue determinante para el correcto desarrollo de la experiencia que el programa estuviese bien hecho, para esto, fue necesario un diseño óptimo que facilitara el trabajo.

A su vez, también se pudo comprobar que no existe una diferencia substancial en el rendimiento entre PThreads y OpenMP para este tipo de tareas, por otro lado, OpenMP posee ciertas ventajas en comparación con PThreads, tales como:

- Una manera más sencilla de implementar paralelización en códigos C++.
- Implementa estructuras de datos **thread-safe** de una manera más fácil para el desarrollador.

Por otra parte, para un buen entendimiento de los resultados del problema, fue necesario buscar una manera clara de mostrar estos, por ello se utilizaron gráficos complementarios a las capturas de los resultados del código.

Finalmente, como conclusión sobre la manera de ver el experimento, fue bueno proponer una hipótesis previa, para dar a entender al lector lo que se busca mostrar en este.

6 Referencias

[1] https://computing.llnl.gov/tutorials/openMP/

[2] https://es.wikipedia.org/wiki/Computaci%C3%B3n_paralela

[3] https://iie.fing.edu.uy/~vagonbar/gcc-make/make.htm

[4] https://www.cs.colostate.edu/~cs675/OpenMPvsThreads.pdf