

Reporte técnico: Taller 02

Taller de Sistemas Operativos
Escuela de Ingeniería Informática

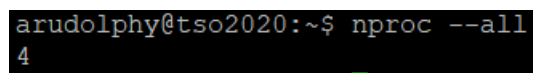
Alejandro Rudolphy Fernández

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, 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 planteará una 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. 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[1], 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 1 se muestra la cantidad de procesadores usados para el desarrollo de este informe.



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

Figura 1 – 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 Threads POSIX. POSIX es una norma escrita y una marca registrada por la IEE (Insitute of Electrical and Eletronics Engineers). Dicha norma define una interfaz estándar del sistema operativo y el entorno, incluyendo un intérprete de comandos (o "shell"), y programas de utilidades comunes para apoyar la portabilidad de las aplicaciones a nivel de código fuente[1].

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 2 se puede observar la estructura de la forma de uso del comando, y en la Tabla 1, el significado de cada uno de los parámetros.

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

Figura 2 – 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 3 – Ejemplo de uso del comando.

1.2 Objetivo

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 **uint32_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, donde 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 (Figura 4), el primer módulo será la parte de llenado, donde el programa recibirá un arreglo de tamaño predeterminado por el usuario, 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 `uint32_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.

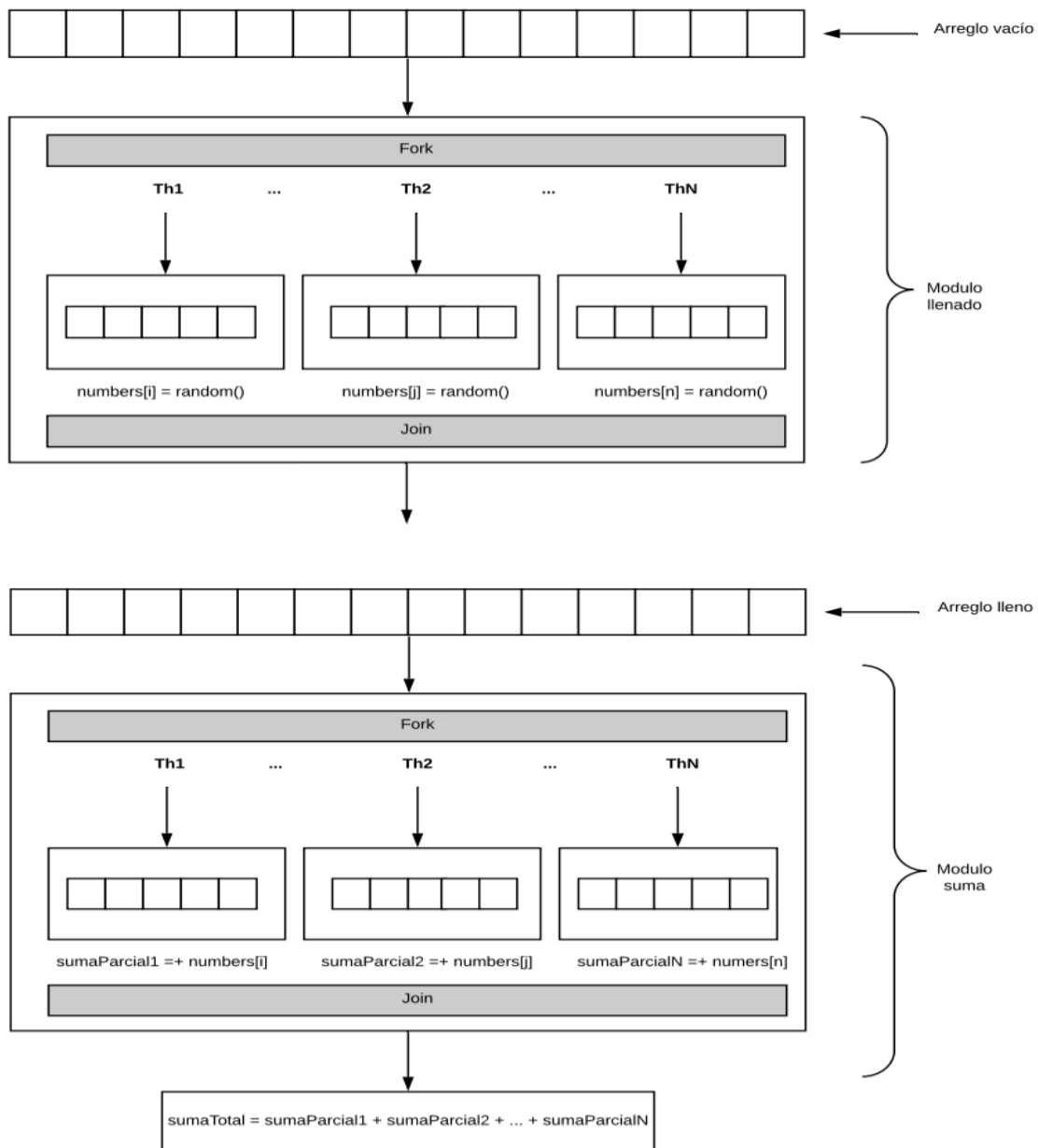


Figura 4 – Diseño general de la solución.

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 5 se puede observar el diagrama de componentes de la función y en la Figura 6 el módulo en detalle.

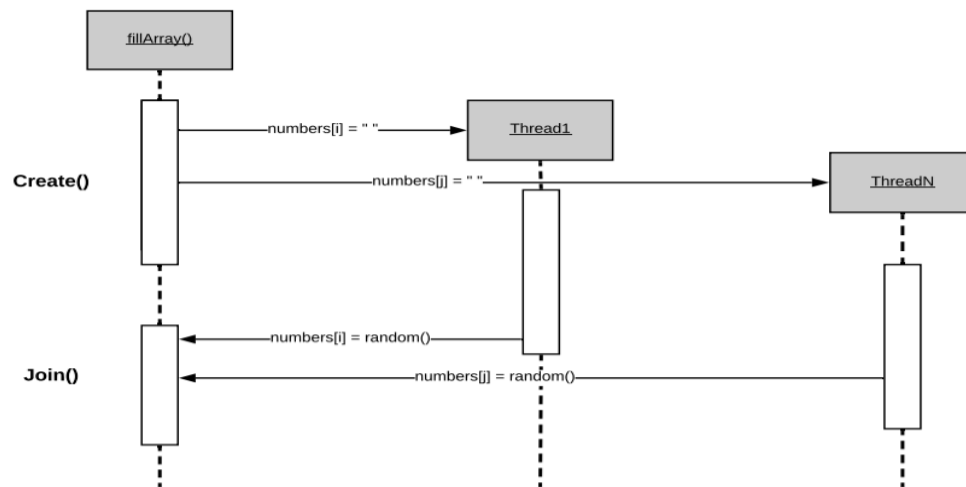


Figura 5 – Diagrama de secuencia módulo de llenado.

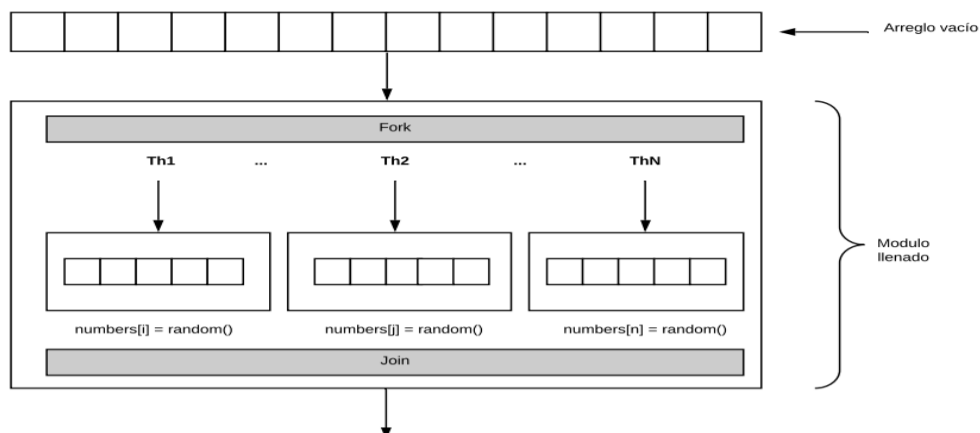


Figura 6 – Diagrama del módulo llenado.

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 7 y 8 se pueden observar diagramas de esta función.

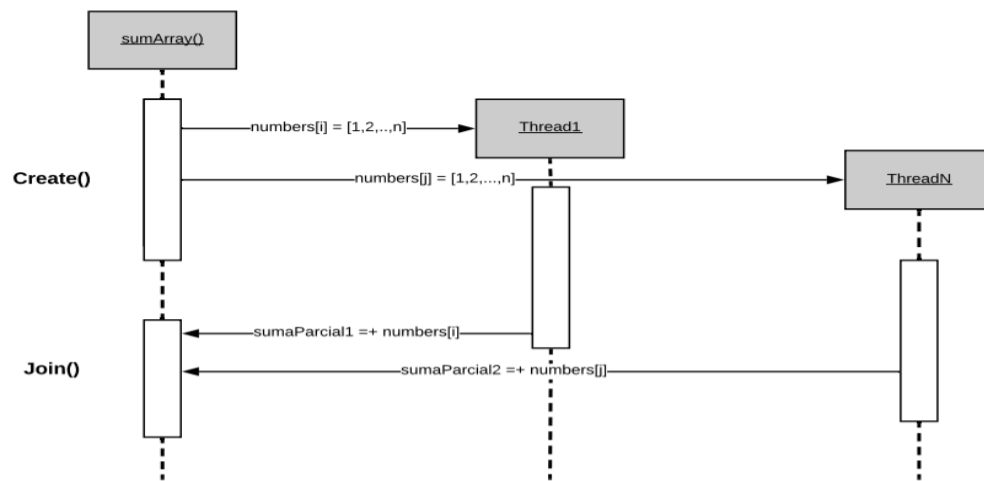


Figura 7 – Diagrama secuencial módulo de suma.

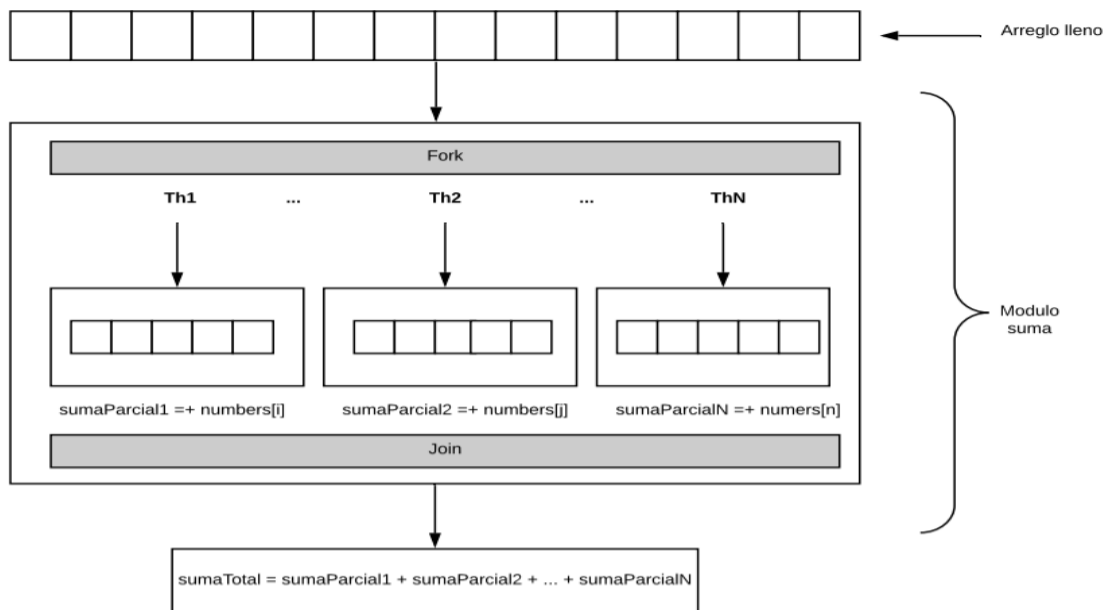


Figura 8 – Diagrama del módulo suma.

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 9 – Implementación de la función fillArray.

```
//Etapa de llenado
//Secuencial
arreglo = new uint64_t[totalElementos];

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

fillArray(0, totalElementos, limInf, limSup);

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;
//Paralelo
arreglo = new uint64_t[totalElementos];

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

//En esta sección se crean los threads en base al valor numThreads ingresado y se distribuye el trabajo entre ellos
for(size_t i=0; i < numThreads; i++){
    threads.push_back(new std::thread(fillArray, i*(totalElementos)/numThreads, (i+1)*(totalElementos)/numThreads, limInf, limSup));
}

for(auto &thFilled : threads){
    thFilled->join();
}

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

Figura 10 – Implementación de la manera secuencial y paralela de la función fillArray.

Como se puede observar en las figuras, las implementaciones difieren debido a que la paralela necesita definir en cuantos threads dividirá la tarea recibida, y en base a esto, dividir la cantidad de elementos en ellos. Por otro lado, su contraparte secuencial crea el ejecuta la función con todos los elementos en un solo arreglo.

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 11 – Implementación de la función fillArray.

```
//Etapa de suma
//Secuencial
uint64_t sumaSecuencial=0;

start = std::chrono::high_resolution_clock::now();
sumaParcial(std::ref(sumaSecuencial), 0, totalElementos);

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();
//En esta sección se crean los threads en base al valor numThreads ingresado y se distribuye el trabajo entre ellos
for(size_t i=0;i<numThreads;i++){
    threadSuma.push_back(new std::thread(sumaParcial, std::ref(sumaParalela), i*(totalElementos)/numThreads, (i+1)*(totalElementos)/numThreads));
}
for(auto &thSuma : threadSuma){
    thSuma->join();
}
end = std::chrono::high_resolution_clock::now();
elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
auto tiempoTotalSuma_P = elapsed.count();
```

Figura 12 – Implementación de la manera secuencial y paralela de la función sumaParcial.

Al igual que en el módulo anterior, se pueden observar las diferencias entre la solución secuencial y paralela con respecto a la creación de threads en la parte paralela y la repartición del trabajo entre estos.

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 13, 14 y 15 se mostrarán los resultados de estos experimentos.

```
arudolph@tso2020:~/TSSOO-Taller02$ ./sumArray -N 10000000 -t 1 -l 1 -L 100
Elementos: 10000000
Threads: 1
Límite inferior: 1
Límite superior: 100
Suma secuencial total: 505061480
Suma en paralelo total: 505061480

Tiempos de ejecución etapa de llenado:

Tiempo Llenado Secuencial: 186[ms]
Tiempo Llenado Paralelo: 196[ms]

Tiempos de ejecución etapa de suma:

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

Figura 13 – 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.

```
arudolph@tso2020:~/TSSOO-Taller02$ ./sumArray -N 10000000 -t 2 -l 1 -L 100
Elementos: 10000000
Threads: 2
Límite inferior: 1
Límite superior: 100
Suma secuencial total: 505101235
Suma en paralelo total: 505101235

Tiempos de ejecución etapa de llenado:

Tiempo Llenado Secuencial: 228[ms]
Tiempo Llenado Paralelo: 126[ms]

Tiempos de ejecución etapa de suma:

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

Figura 14 – Prueba de desempeño con 2 threads.

Para la segunda prueba se utilizaron dos thread, 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.

```

arudolph@tso2020:~/TSSOO-Taller02$ ./sumArray -N 10000000 -t 4 -l 1 -L 100
Elementos: 10000000
Threads: 4
Límite inferior: 1
Límite superior: 100
Suma secuencial total: 504980092
Suma en paralelo total: 504980092

Tiempos de ejecución etapa de llenado:

Tiempo Llenado Secuencial: 206[ms]
Tiempo Llenado Paralelo: 81[ms]

Tiempos de ejecución etapa de suma:

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

```

Figura 15 – 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.

4.2 Resultados sobre la hipótesis

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 el tiempo del módulo de suma no varía, 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.

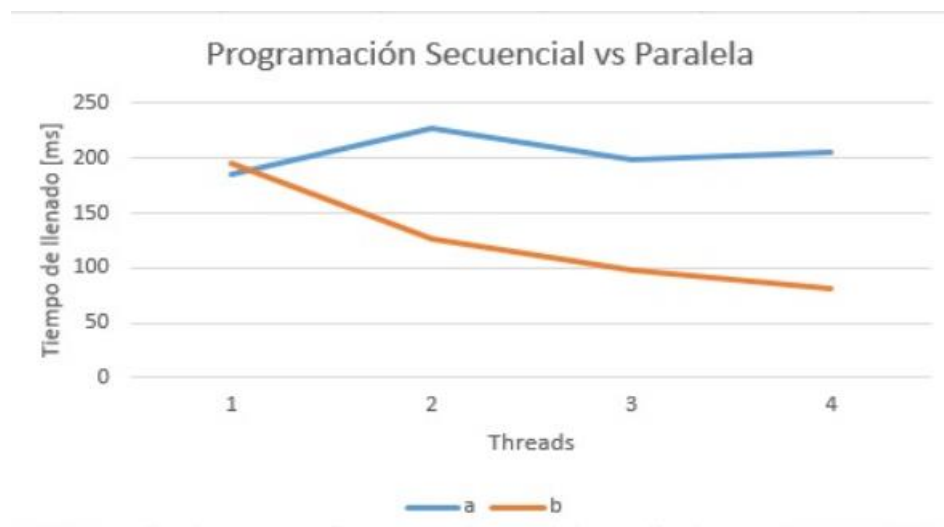


Figura 16 – Gráfico del experimento

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.

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 con este experimento.

6 Referencias

- [1] https://es.wikipedia.org/wiki/Computaci%C3%B3n_paralela
- [2] <https://es.wikipedia.org/wiki/POSIX>
- [3] <https://iie.fing.edu.uy/~vagonbar/gcc-make/make.htm>

