Implementación y análisis de OpenMP en C ++

Taller de Sistemas Operativos
Escuela de Ingeniería Informática

Ignacio Alvarado Torreblanca

Ignacio.alvarado@alumnos.uv.cl

Resumen. *El objetivo principal del informe, es implementar un programa que llene un arreglo de números enteros y luego los sume, ambas tareas se realizarán en forma paralela, implementadas con OpenMP en lenguaje de programación C++ junto a sus librerías correspondientes, todo bajo un ambiente contenido en un servidor Ubuntu Server 18.04.*

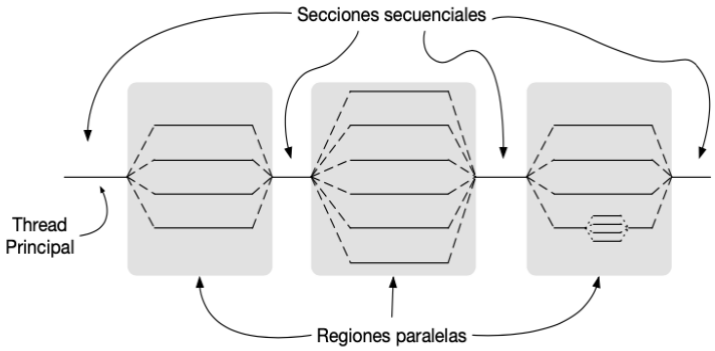
1 Introducción

Debido a las limitaciones físicas que impiden el aumento de la frecuencia, el consumo de energía, y por consiguiente la generación de calor de las computadoras constituye una preocupación en los últimos años, la computación en paralelo se ha convertido en el paradigma dominante en la arquitectura de computadores, principalmente en forma de procesadores multinúcleo.

La programación paralela divide el problema en partes independientes de modo que cada elemento de procesamiento pueda ejecutar su parte del algoritmo de manera simultánea con los otros, donde cada parte es secuencial. Para tener una programación paralela es necesario implementar los threads.

Los threads son una secuencia de tareas encadenadas muy pequeña que puede ser ejecutada por un sistema operativo, estos poseen un estado de ejecución y pueden sincronizarse entre ellos para evitar problemas de compartición de recursos. Generalmente, cada thread tiene una tarea específica y determinada, como forma de aumentar la eficiencia del uso del procesador. Algunos lenguajes de programación tienen características de diseño expresamente creadas para permitir a los programadores lidiar con hilos de ejecución (como Java o Delphi), otros desconocen la existencia de hilos de ejecución y estos deben ser creados mediante llamadas de biblioteca especiales que dependen del sistema operativo en el que estos lenguajes están siendo utilizados (como es el caso del C y del C++). En la figura 1 se puede ver un diagrama del funcionamiento de los thread.

Figura 1: Funcionamiento de un thread



Las interfaces de programación de aplicaciones, más conocidas como APIs, dada la definición en inglés (application programming interface), son un conjunto de funcionalidades, procedimiento automatizados que nos entrega un proveedor de servicios, mediante un software, sistema o aplicación WEB, representados como librerías. OpenMP es una API para la realización de multiprocesos de memoria compartida en múltiples plataformas. sobre la base del modelo de ejecución fork-join. Está disponible en muchas arquitecturas, incluidas las plataformas de Unix y de Microsoft Windows. Se compone de un conjunto de directivas de compilador, rutinas de biblioteca, y variables de entorno que influyen el comportamiento en tiempo de ejecución, OpenMP se basa en el modelo fork-join, paradigma que proviene de los sistemas Unix, donde una tarea muy pesada se divide en N hilos (fork) con menor peso, para luego "recolectar" sus resultados al final y unirlos en un solo resultado (join).

El desarrollo del trabajo se realiza en el lenguaje de programación C++, con la Librería del API para la programación en multiprocesos OpenMP, que automatiza la paralización de un código serial.

La estructura del informe esta está compuesta por una sección que introduzca sobre los conceptos primordiales del objetivo del informe, para luego presentar el trabajo realizado, explicando el problema a implementar junto a un contexto de los datos, para luego explicar la descripción de los parámetros del programa. Luego, se muestra el diseño del problema donde se encuentran diagramas de alto nivel de la solución. Después, se tiene la sección de pruebas, en la que se visualizará las ejecuciones del código con distintos parámetros, para después pasar a la sección de resultados, donde se evaluará el desempeño del problema en paralelo, comparándolo con el desempeño del mismo problema en forma secuencial, para luego terminar con la conclusión.

2 Descripción del problema

2.1 Trabajo a realizar

Se creará un programa que esté compuesto de dos módulos. Uno que llenara un arreglo de números enteros aleatorios del tipo uint32_t en forma paralela y otro que sumará el contenido del arreglo en forma paralela, estos dos módulos se implementaran con OpenMP. Para generar los números aleatorios, se utilizaran funciones que sean thread safe para que se vea una mejora en el desempeño del programa.

Para la ejecución del programa, se usarán los parámetros de la siguiente forma: ./sumArray -N <nro> -t <nro> -l <nro> -L <nro> [-h]. En la tabla 1, se detallan los parámetros junto a su respectiva descripción.

Tabla 1: Parámetros del programa

Parámetro	Descripción
-N	Tamaño del arreglo
-t	Número de threads
-l	Límite inferior del rango aleatorio
-L	Límite superior del rango aleatorio

[-h]	Muestra la ayuda de uso y termina

2.2 Ejemplo de uso

El programa que se busca desarrollar tendrá dos formas de uso, una la cual generará el paralelismo de los procesos ya nombrados mediante hilos, y la otra que mostrará la ayuda de uso del programa.

1) Para ejecutar el programa creando un arreglo de 1000000 posiciones, con 4 threads y que los números enteros aleatorios están en el rango [10,50] será necesario tener los siguientes parámetros: ./sumArray -N 1000000 -t 4 -l 10 -L 50.
2) Para mostrar la ayuda de uso del programa será necesario tener el siguiente parámetro: ./sumArray -h.

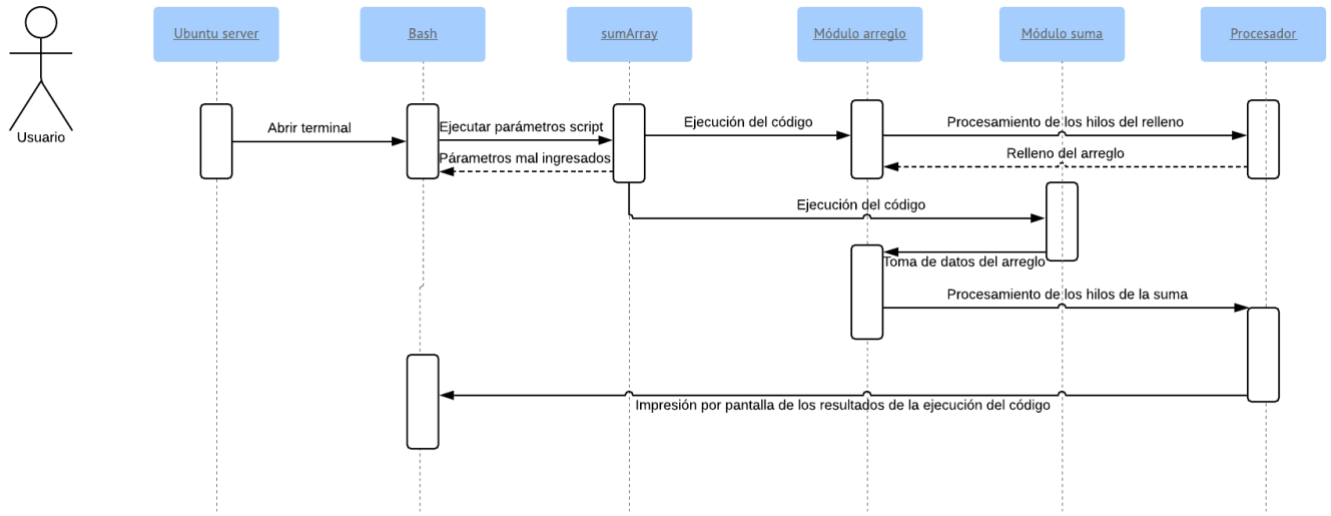
2.3 Análisis de los datos

Luego de tener el programa listo, se realizarán pruebas de desempeño que generen datos que permitan visualizar el comportamiento del tiempo de ejecución de ambos módulos dependiendo del tamaño del problema y de la cantidad de threads utilizados. Esto se hará mediante comandos que permitan medir el tiempo de ejecución de los módulos en la terminal bash.

3 Diseño de la solución

En la figura 2 se puede observar un diagrama de secuencia que explica el funcionamiento de los procesos que se ejecutarán para llegar a la solución del problema.

Figura 2: Diagrama de secuencia del problema



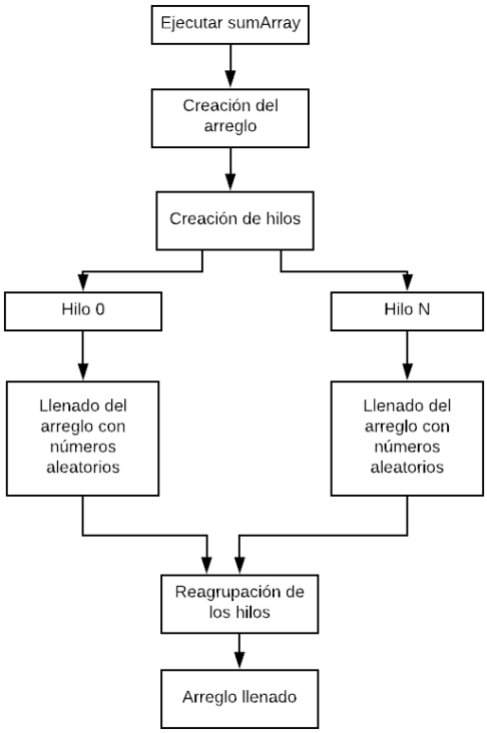
Para resolver el problema será necesario conocer el lenguaje de programación c ++ junto a OpenMP que se encarga de la paralelización del problema, como también los comandos para medir los tiempos de ejecución de las ejecuciones del programa los cuales serán necesarios para el posterior análisis de estos.

Una vez ejecutado el programa este debe comenzar a crear el arreglo vacío, con la cantidad de hilos y el total de elementos agregados en los parámetros al momento de la ejecución. Para la paralelización de los módulos se usarán funciones acorde a lo que pide cada módulo. El llenado se debe hacer mediante el encargado de generar números aleatorios en base a los límites propuestos a la hora de ejecutar el programa. Tras realizar el llenado del arreglo, se debe ejecutar el módulo de suma del arreglo.

3.1 Módulo llenado

En base al diseño general de la solución , se puede diseñar el módulo respectivo al llenado del arreglo. El diseño de este módulo se puede ver en la figura 3.

Figura 3: Diagrama de flujo módulo llenado del arreglo

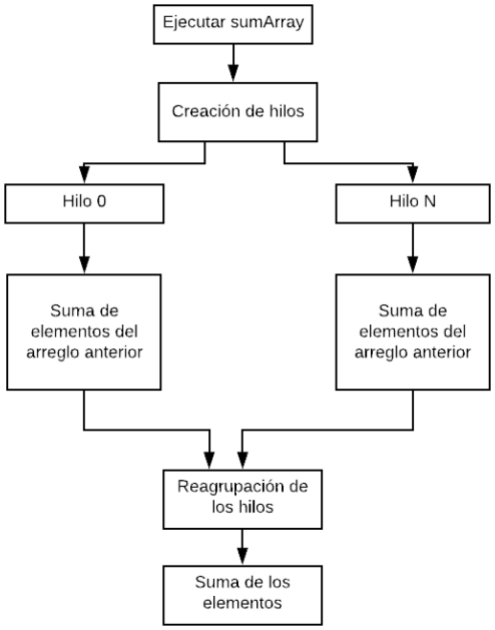


El arreglo debe separarse para cada hilo instanciado con OpenMP, una vez realizado esto, cada hilo debe rellenar cada fragmento del arreglo , con números aleatorios dentro de los rangos establecidos por los parámetros de entrada. Una vez realizado esto por cada hilo, se debe reagrupar todos los pedazos del arreglo en uno solo para ser usado luego por el siguiente módulo.

3.2 Módulo suma

Tras rellenar el arreglo en el módulo anterior, se debe instanciar nuevamente cada hilo registrado en los parámetros, esto con OpenMP, para que realice el proceso de suma del contenido del arreglo. El diseño de este módulo se puede ver en la figura 4.

Figura 4: Diagrama de flujo módulo de suma del arreglo



En base al diseño, se toma el arreglo llenado por cada hilo, para luego unir estos hilos y finalmente igualar el arreglo de las sumas a una variable que representará la suma total de elementos del arreglo, este proceso será automatizado por OpenMP.

4 Pruebas de ejecución

Luego de la finalización del codigo, al iniciar el programa con parámetros de entrada que no corresponden al uso, se muestra su forma correcta de uso. En la figura 5 se puede visualizar como se debería ejecutar el programa:

Figura 5: Forma de uso del sumArray

```
[ignacio@tso:~/taller03/U2/taller03$ ./sumArray
Uso: ./sumArray -N <nro> -t <nro> -l <nro> -L <nro> [-h] Párametros:
        -N : tamaño del arreglo.
        -t : número de threads.
        -l : limite inferior rango aleatorio.
        -L : límite superior rango aleatorio.
        [-h] : muestra la ayuda de uso y termina.
```

También se muestra su forma de uso cuando cualquiera de los parámetros de entrada de sumArray no corresponde con el del programa.

Para estas pruebas se realizaron un número de 12 pruebas con los mismos parámetros de entrada a excepción de los threads que serán recorridos del 1 al 12 para ser analizados posteriormente en la sección de resultados. Como se puede ver en la figura 6 el tiempo de llenado del arreglo se realiza con 1 thread.

Figura 6: Prueba realizada con 1 thread

```
ignacio@tso:~/taller03/U2/taller03$ ./sumArray -N 10000000 -t 1 -l 1 -L 100000
Elementos: 10000000
Threads: 1
limite inferior: 1
limite Superior: 100000
Suma de elementos del arreglo con OpenMP: 499910157310
Suma de elementos del arreglo con threads: 499910157310
Suma de elementos del arreglo secuencial: 499910157310
-------Tiempos modulo suma ------
Tiempo de suma de los elementos con OpenMP: 9.26325
Tiempo de suma de los elementos con threads: 11.789
Tiempo de suma de los elementos secuencial: 11.5876
Aceleracion  etapa de Suma `Threads v/s Secuencial`: 0.982912
Aceleracion  etapa de Suma `OpenMP v/s Secuencial`: 1.25092
-------Tiempos modulo Llenado -------
Tiempo de llenado con OpenMP: 485.083
Tiempo de llenado con threads: 485.114
Tiempo llenado secuencial: 488.608
Aceleracion etapa de Llenado ´Threads v/s Secuencial´: 1.0072
Aceleracion etapa de Llenado ´OpenMP v/s Secuencial´: 1.00727
```

En la figura 7 y 8, se tiene la ejecución de sumArray con distintos parámetros de entrada para evidenciar el correcto funcionamiento de este. Cabe recalcar que las pruebas de la figura 7 y 8 no fueron tomadas en cuenta en la sección de resultados.

Figura 7: Primera prueba realizada con distintos parámetros de entrada

```
ignacio@tso:~/taller03/U2/taller03$ ./sumArray -N 7126362 -t 3 -l 2392 -L 9283742
Elementos: 7126362
Threads: 3
limite inferior: 2392
limite Superior: 9283742
Suma de elementos del arreglo con OpenMP: 33080313462773
Suma de elementos del arreglo con threads: 33080313462773
Suma de elementos del arreglo secuencial: 33080313462773
-------Tiempos modulo suma ------
Tiempo de suma de los elementos con OpenMP: 4.08257
Tiempo de suma de los elementos con threads: 5.04811
Tiempo de suma de los elementos secuencial: 9.05498
Aceleracion  etapa de Suma `Threads v/s Secuencial`: 1.79373
Aceleracion  etapa de Suma `OpenMP v/s Secuencial`: 2.21796
-------Tiempos modulo Llenado -------
Tiempo de llenado con OpenMP: 217.753
Tiempo de llenado con threads: 146.96
Tiempo llenado secuencial: 347.34
Aceleracion etapa de Llenado ´Threads v/s Secuencial´: 2.36349
Aceleracion etapa de Llenado ´OpenMP v/s Secuencial´: 1.59511
```

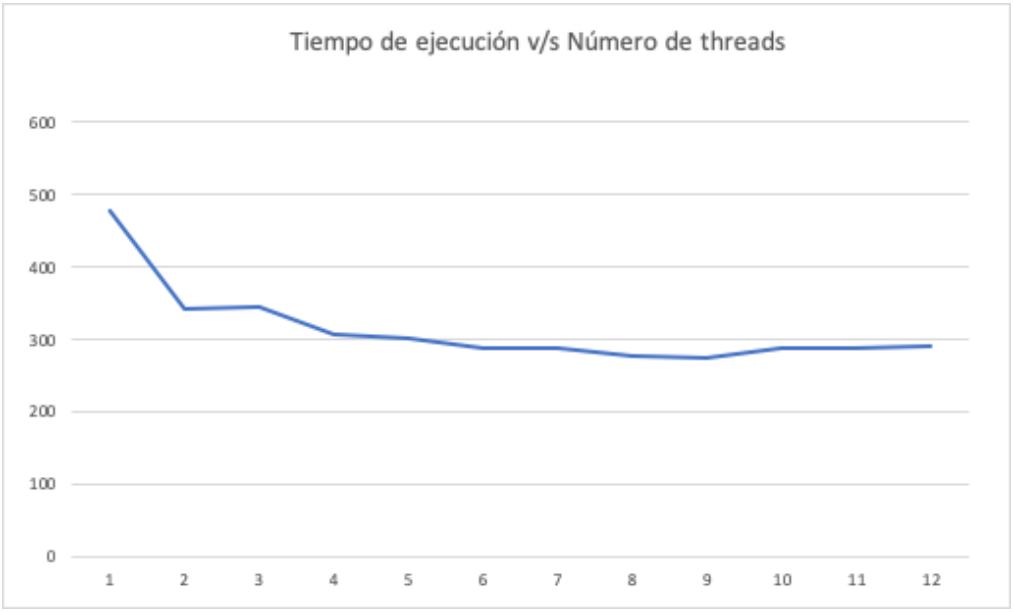
Figura 8: Segunda prueba realizada con distintos parámetros de entrada

```
ignacio@tso:~/taller03/U2/taller03$ ./sumArray -N 73284238 -t 3 -l 8234 -L 47834827
Elementos: 73284238
Threads: 3
limite inferior: 8234
limite Superior: 47834827
Suma de elementos del arreglo con OpenMP: 1753099129539832
Suma de elementos del arreglo con threads: 1753099129539832
Suma de elementos del arreglo secuencial: 1753099129539832
-------Tiempos modulo suma ------
Tiempo de suma de los elementos con OpenMP: 47.3368
Tiempo de suma de los elementos con threads: 57.4821
Tiempo de suma de los elementos secuencial: 86.7436
Aceleracion  etapa de Suma `Threads v/s Secuencial`: 1.50905
Aceleracion  etapa de Suma `OpenMP v/s Secuencial`: 1.83248
-------Tiempos modulo Llenado -------
Tiempo de llenado con OpenMP: 2267.58
Tiempo de llenado con threads: 1764.91
Tiempo llenado secuencial: 3616.89
Aceleracion etapa de Llenado ´Threads v/s Secuencial´: 2.04933
Aceleracion etapa de Llenado ´OpenMP_v/s Secuencial´: 1.59505
```

5 Resultados

Como fue nombrado en la sección anterior, se analizaron 12 pruebas en las que se iba a aumentando el número de threads, manteniendo el mismo total de elementos del arreglo, que fueron un total de 10000000 elementos y el rango para generar el número aleatorio, que iba desde 1 hasta 100000. En la figura 9, se puede ver el gráfico del módulo de relleno del arreglo, que representa el tiempo de ejecución en milisegundos en función del número de threads que variaba en cada prueba realizada.

Figura 9: Gráfico de tiempo de ejecución v/s Número de threads correspondiente al módulo llenado del arreglo implementado con OpenMP

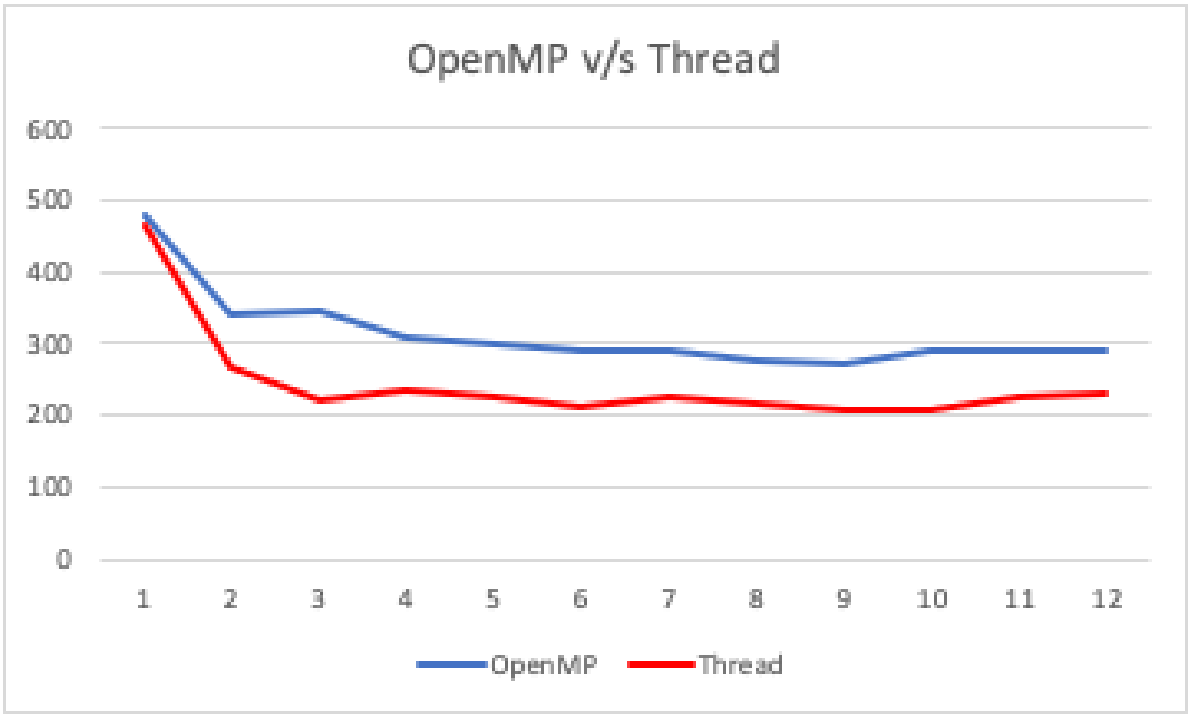


En la figura 10, se puede ver el gráfico del módulo de suma del arreglo, que representa el tiempo de ejecución en milisegundos en función del número de threads que variaba en cada prueba realizada.

Figura 10: Gráfico de tiempo de ejecución v/s Número de threads correspondiente al módulo suma del arreglo implementado con OpenMP

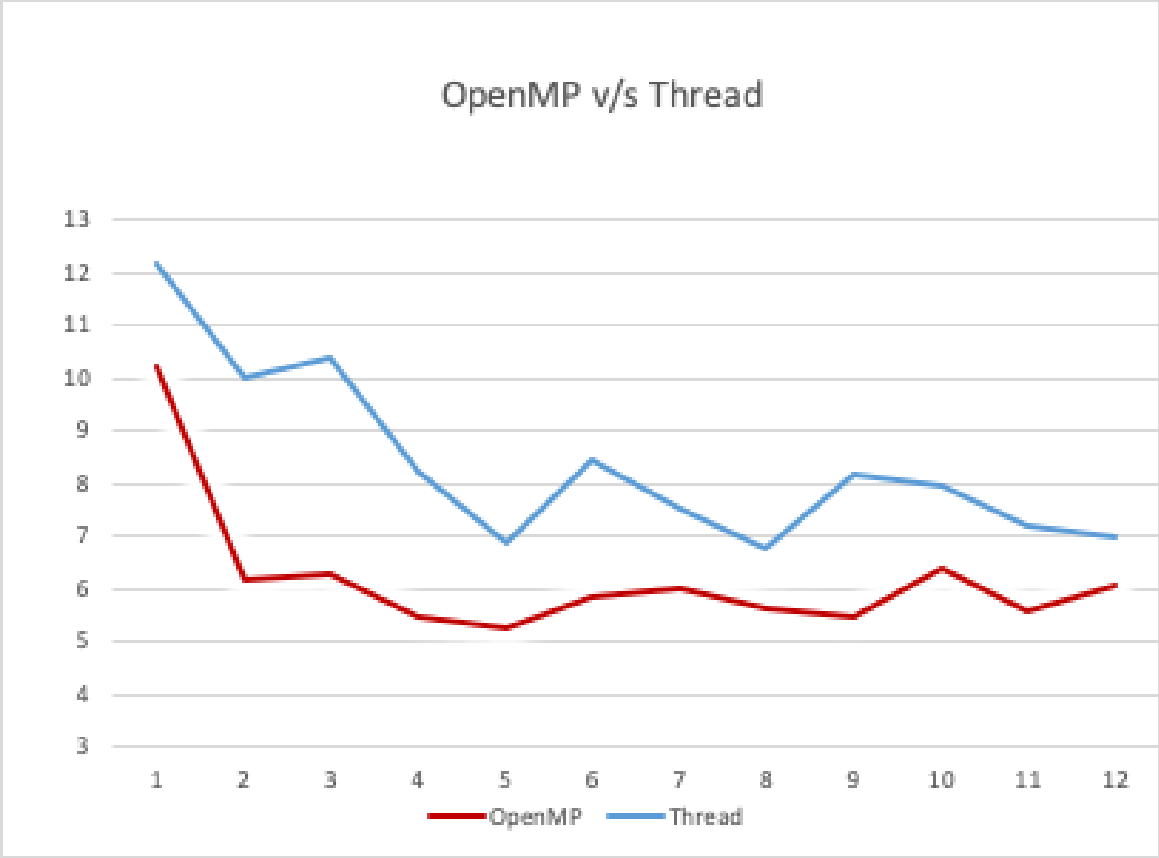
En la figura 11, se puede ver el gráfico de la comparación entre los tiempos de ejecución en milisegundos del módulo de llenado en forma paralela implementado con OpenMP versus la implementada con threads incluidos en la librería "thread" de C ++. Esta comparación se realizó con los mismos parámetros en ambas implementaciones.

Figura 11: Gráfico comparación del tiempo de ejecución de la implementación del módulo llenado en OpenMP y en la librería "thread" de C ++



En la figura 12, se puede ver el gráfico de la comparación entre los tiempos de ejecución en milisegundos del módulo de suma en forma paralela implementado con OpenMP versus la implementada con threads incluidos en la librería “thread” de C ++. Esta comparación se realizó con los mismos parámetros en ambas implementaciones.

Figura 12: Gráfico comparación del tiempo de ejecución de la implementación del modulo suma en OpenMP y en la librería “thread” de C ++



6 Conclusión

En conclusion, se cumplio el objetivo del informe, ya que como se pudo ver en la sección de pruebas, el programa funcionaba con cualquier parámetro que corresponda a la forma de uso.

También, en base a la sección de resultados, se puede concluir que a partir de ocupar 4 threads hasta 12 threads, el tiempo de ejecución tanto del llenado como de la suma se mantiene e incluso aumenta en ciertas ocasiones, y los mejores tiempos de ejecución se dan cuando se ocupan entre 2 a 4 threads. Otra cosa a tener en cuenta es que al comparar la implementación de ambos módulos en OpenMP con la librería “thread” de C ++, el módulo de llenado tiene menor tiempo de ejecución implementado con la librería “thread”, en cambio el modulo de suma tiene mejor tiempo de ejecución con OpenMP.