Reporte Técnico Taller 03

Taller de Sistemas Operativos
Escuela de Ingeniería Informática
Ignacio Miranda Yáñez

Ignacio.miranda@alumnos.uv.cl

1 Introducción

Dentro de lo que tenemos entendido de este taller es la misma situación que el anterior en la cual se trabajo en el llenado de un arreglo y la suma del conjunto de número dentro de este arreglo de manera paralela, la única diferencia con lo que sabemos y tenemos que hacer es que ahora realizaremos este taller con la **API** llamada **OpenMP** esta fue desarrollada específicamente para el procesamiento de la memoria compartida en paralelo y consiste en un conjunto de directivas de compilación, rutinas de biblioteca y variables de entorno que influyen en el comportamiento en tiempo de ejecución.
También se dejará la misma estructura de informe correspondiente con el anterior que es la introducción propiamente tal, la información a nivel teórico de la programación paralela en general además de añadir información acerca de la **API**, también se realizaran pruebas y se analizaran los resultados con respecto a estas mismas y como tal un diseño de la solución del problema

2 Conjunto Teórico

En esta sección se presentarán un conjunto de conceptos con información para entender que aspectos se utilizaran para que la programación paralela sea posible

2.1 API OpenMP

Definido conjuntamente por proveedores de hardware y de software, **OpenMP** es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas, para plataformas que van desde las computadoras de escritorio hasta supercomputadoras. Una aplicación construida con un modelo de programación paralela híbrido se puede ejecutar en un cluster de computadoras utilizando **OpenMP** y **MPI**, o a través de las extensiones de **OpenMP** para los sistemas de memoria distribuida.

OpenMP comprende tres componentes complementarios:

- Un conjunto de directivas de compilador usado por el programador para comunicarse con el compilador en paralelismo
- Una librería de funciones en tiempo de ejecución que habilita la colocación e interroga sobre los parámetros paralelos que se van a usar, tal como numero de los hilos que van a participar y el número de cada hilo
- Un numero limitado de las variables de entorno que pueden ser usadas para definir en tiempo de ejecución parámetros del sistema en paralelo tales como el numero de hilos

Características:

- Paralelismo de memoria compartida basada en **threads**
- Modelo **fork-join**
- Basado en directivas al compilador
- Soporta paralelismo dentro del paralelismo
- **Threads** dinámicos

Funciones:

Dispone de una serie de funciones para obtener información y configurar el entorno paralelo. También permite manejar candados (locks) y tomar mediciones de tiempo.

Hay partes de la especificación que son dependientes de la implementación por lo que el programa no tiene porqué comportarse igual si el binario se genera primero con una implementación y luego por otra. Por ejemplo, la creación de regiones paralelas anidadas no especifica cómo se han de suministrar los hilos para las regiones anidadas. En el caso de **GNU OpenMP** mantiene un único equipo de hilos y lo hace crecer o decrecer en función de la cantidad de hilos demandada por la región paralela activa. Esto es una optimización que ayuda a mantener la localidad de datos y disminuir los sobrecostes de creación y destrucción de hilos. Sin embargo, una implementación podría no mantener un equipo único y crear nuevos equipos enteros. Esto provoca la expulsión de los hilos de regiones paralelas predecesoras. La asignación de identificadores a procesadores no sigue siempre el mismo orden salvo que se indique lo contrario. Si el código del programa depende del identificador del hilo para la asignación de datos a procesar, se puede perder la localidad espacial del programa. Crear y destruir múltiples equipos de hilos requiere también de más tiempo que reutilizar equipos de hilos ya existentes.

3 Descripción Del Problema

Específicamente se deben crear dos módulos, uno de estos módulos debe llenar un arreglo con números enteros de forma randomica del tipo **uint32_t** en forma paralela y el otro modulo vaya sumando el contenido del arreglo del primer módulo también de forma paralela, el problema también consiste en la entrada de un conjunto de parámetros como el tamaño del arreglo, numero de **threads** a utilizar y el límite inferior y superior del rango del conjunto de números aleatorios que entraran en el arreglo descripción

Campo	**Descripción**	**Ejemplo**
-N	Tamaño Del arreglo	0 – 1000000 enteros
-t	Numero de threads	0 – 16
-l	Límite inferior del rango aleatorio	10
-L	Límite superior del rango aleatorio	60
[-h]	Comando de ayuda para mostrar el uso del programa	Uso: ./sumArray -N <nro> -t <nro> -l <nro> -L <nro> [-h] Descripción: -N: tamaño del arreglo. -t: número de threads. -l: límite inferior rango aleatorio. -L: límite superior rango aleatorio. [-h]: muestra la ayuda de uso y termina

Tabla 1. Forma de uso y ejemplo de parámetros

Arreglo [] = Arreglo dinámico que se llena con números aleatorios indicado por el parámetro -N

ArregloSuma[] = Arreglo dinámico que se llena con la suma de los números aleatorios del arreglo Arreglo[] y su tamaño es indicado por el parámetro -N

openmp[] = Arreglo dinámico que se llena para ser utilizado con respecto a las librerías de **OpenMP**

Llenado() = función que llena el arreglo ArregloRandom de números aleatorios y se basa en la concepción **thread** safe y se indica por los parámetros -l y -L

Para medir el tiempo de ejecución del programa se utilizara el método **std::chrono** que está en la biblioteca $\langle chrono \rangle$

4 Diseño De La Solución

Primero que nada, para la realización de este diseño se analizó los requerimientos que se pedían y las variables que se iban a utilizar como también el formato de salida, entonces lo que se pide es tomar dos módulos uno llenara un arreglo dinámico con números randomicos y el otro modulo sumara los números y los dejara en otro arreglo, también el sistema general del código tiene variables dinámicas que se indican por parámetros vistos anteriormente

4.1 Módulo 1 (Llenado Del Arreglo)

En este proceso se indica el arreglo dinámico llamado **Arreglo_random[]** y este mismo se dividirá según los hilos que se hayan introducido por medio de los parámetros, cada hilo sabe cuál es la parte que le toca del arreglo por medio de un inicio **(beginIdx)** y un final **(endIdx)** indicados por lo que en esos rangos debe llenar su conjunto de números(ver Figura 9)[1], esto se realiza en forma paralela con el módulo 2 que se mostrara en la próxima etapa

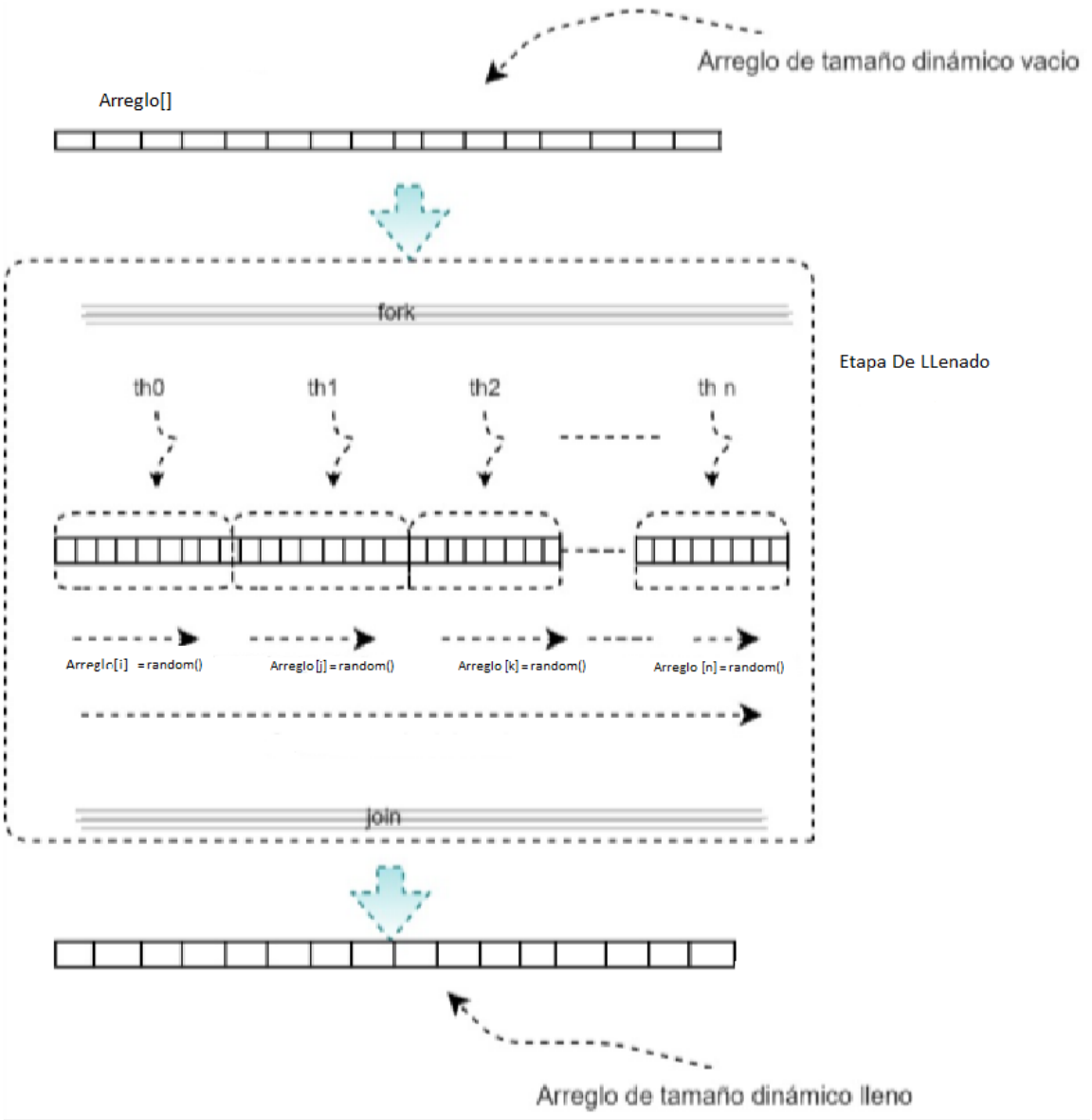

Figura 1. Modulo 1

4.2 Módulo 2 (Suma Del Arreglo)

En este módulo se toma el arreglo ya llenado anteriormente y como se realizó en el módulo anterior se divide el arreglo dependiendo de los hilos que se hayan indicado, cada hilo le corresponde una porción de arreglo y después de esto, se indican las sumas parciales que indicaron cada hilo, al final de todo el módulo, este conjunto de sumas se suma en total valga la redundancia

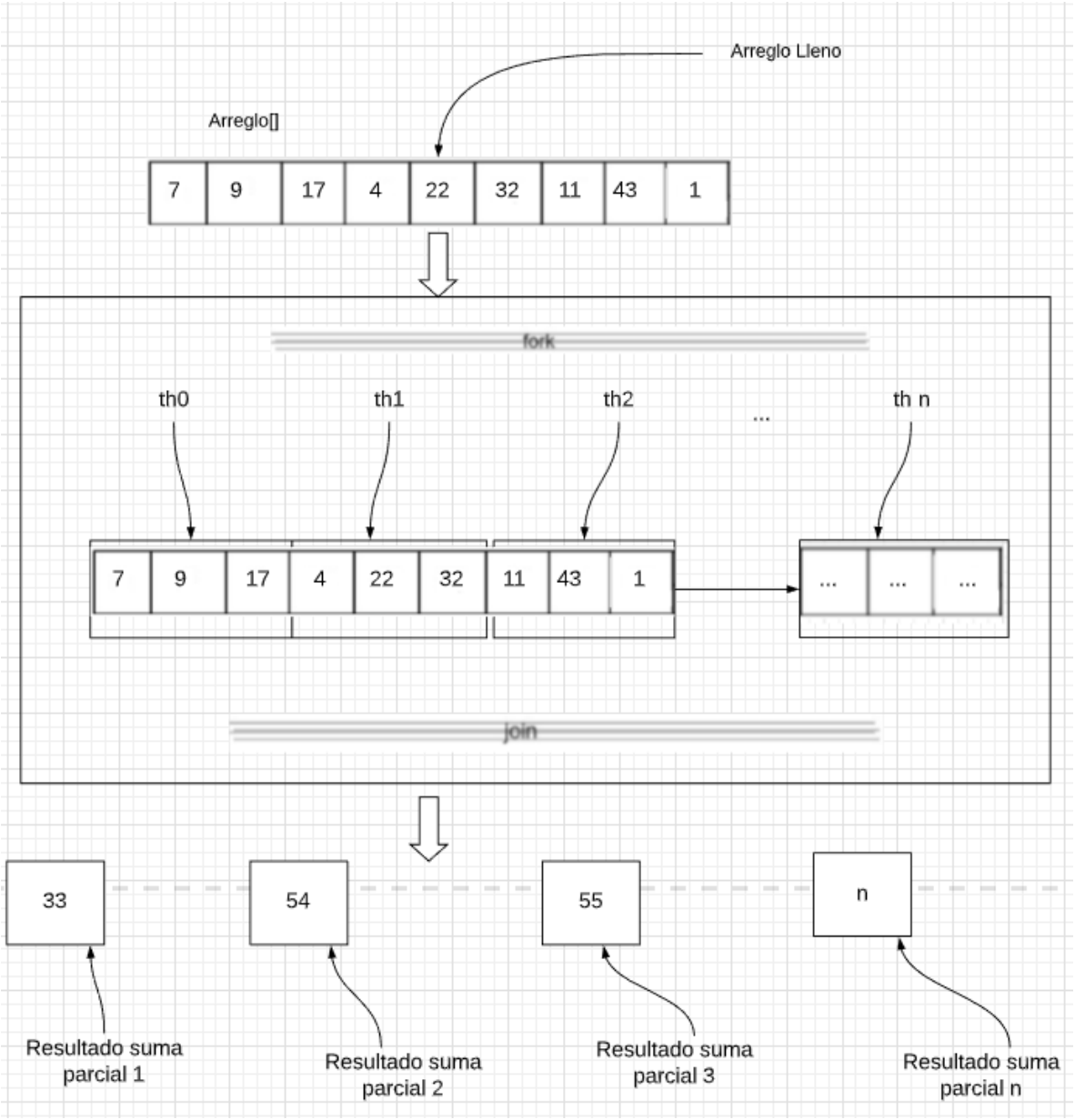

Figura 2. Suma del arreglo

5 Proceso De Ejecución

En un principio en la maquina virtual se debe tener instalada la biblioteca para la ejecución del script por lo tanto se colocan los parámetros como se ve en la imagen

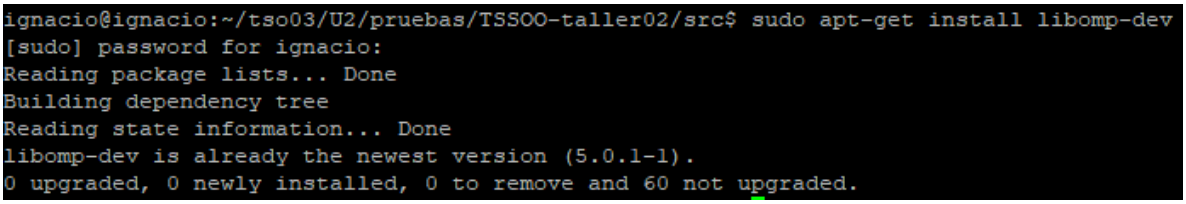

Imagen 1.

Después de la instalación de la biblioteca se puede proceder a la clonación del repositorio

En esta etapa se indicará el proceso que se debe realizar para clonar el repositorio indicado y ejecutar el script, se ingresó a una carpeta vacía y se comprobó con el comando **ls-l** que no había archivos, después de eso se clono el repositorio correspondiente

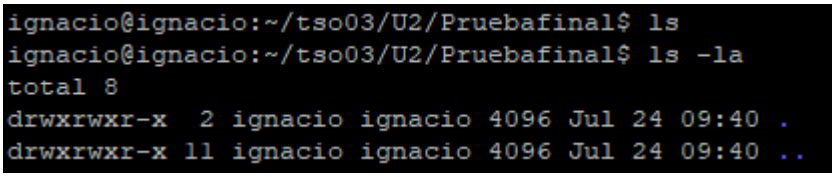

Imagen 2.

Clonado el repositorio se revisan si todos los archivos están en la carpeta clonada

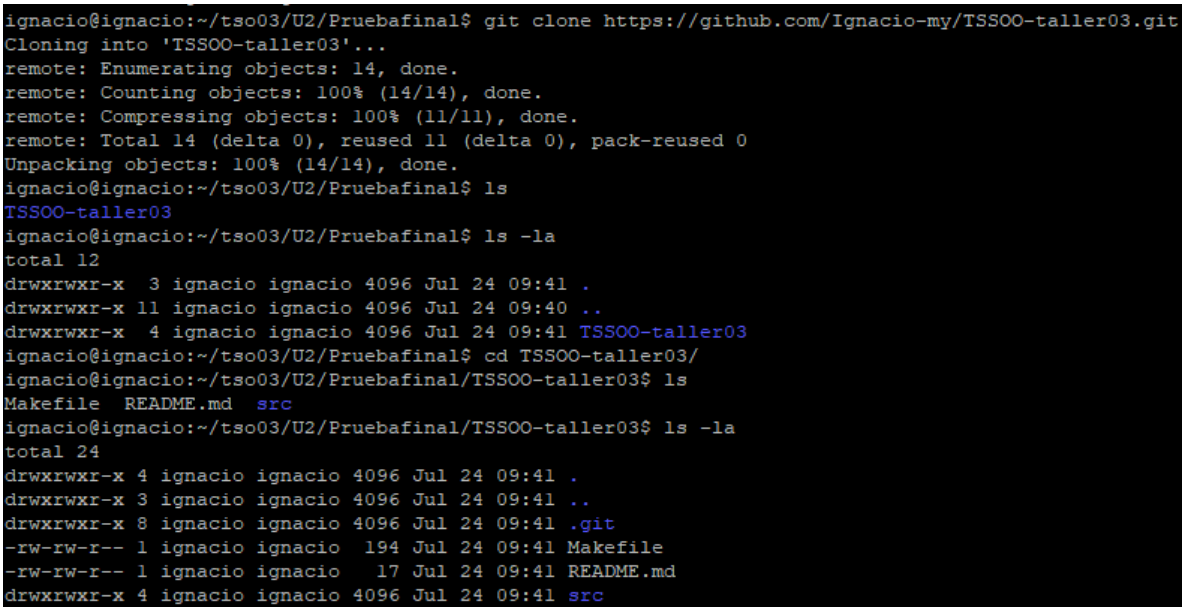

Imagen 3

.

Ya revisadas las carpetas, se procede a hacer un **make clean** correspondiente y un **make** para compilar el script, este se compila y se creara en una carpeta anterior la cual es **TSSOO-taller03**, y el script estaría listo para su uso, este ocupara los parámetros ya mencionados

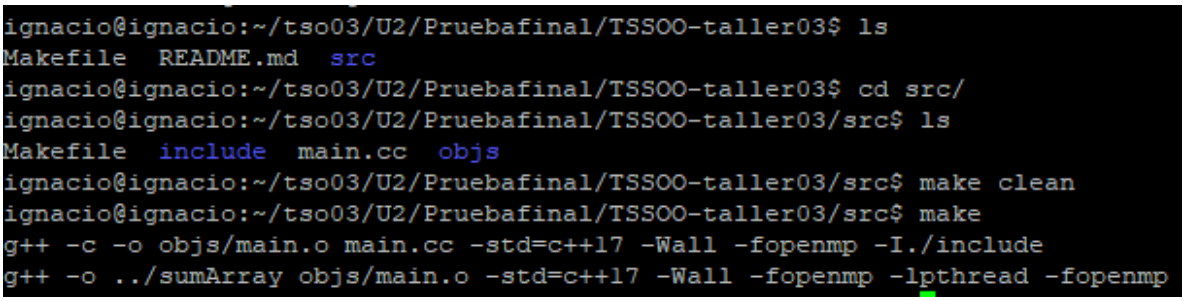

Imagen 4.

6 Resultados

Este conjunto de resultados fue realizado en un Pc con las siguientes especificaciones

- AMD Ryzen™ 5 2400G con gráficos Radeon™ RX Vega 11 3,6 GHz
- 16 Gb RAM
- Crucial® Unidad SSD 240GB Sata3 2.5" BX500

Demostraciones de 10 pruebas mostrando solo el numero promedio de estas con el mismo parámetro el cual sería 1000000 elementos y un rango de números aleatorios de 10 a 100.

6.1 Tiempo De Ejecución Total

En esta prueba se muestra la suma total del tiempo de ejecución del llenado y de la suma entre el uso de **threads** y **OpenMP**, Se puede demostrar en el caso general que el tiempo de ejecución total al usar **threads** se obtiene un resultado mejor ya que se demora menos a diferencia de lo que se demora el **OpenMP** y a la vez que se van agregando hilos el tiempo disminuye hasta que se llega al caso de los 8 hilos donde el tiempo aumenta de forma inusual y llega a ser mayor que cuando se ocupan 2 hilos

hilos	Tiempo Ejecucion Threads Total	Tiempo Ejecucion OpenMP Total
2	11	52,7
4	9,7	46,7
6	8,7	44,7
8	14	57,57

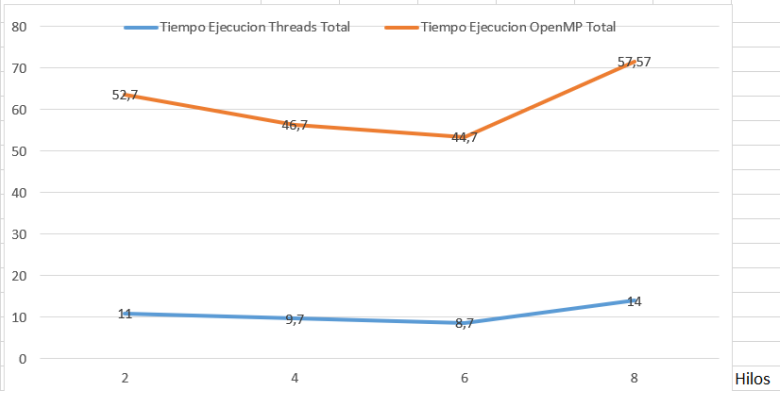

Imagen 5. Tiempo Ejecución Total

6.2 Tiempo Llenado Del Arreglo

En este estado es donde se muestra el mayor aporte de tiempo con respecto al tiempo general visto anteriormente ya que el tiempo de llenado aumenta considerablemente y existe una gran diferencia de tiempo en lo que respecta al uso de **threads** y **OpenMP** en lo que se puede demostrar un aumento de 5 veces el tiempo de ejecución por parte de **OpenMP**, y por lo que se ha dicho cada vez que se aumentan los hilos el tiempo disminuye y siempre aumenta cuando se llega a las 8 hilos, por lo que se concluye que en esta situación es mejor el uso de **threads**

hilos	Tiempo Llenado Threads	Tiempo Llenado OpenMP
2	5,3	51
4	6	45,4
6	5	42,6
8	9	54,2

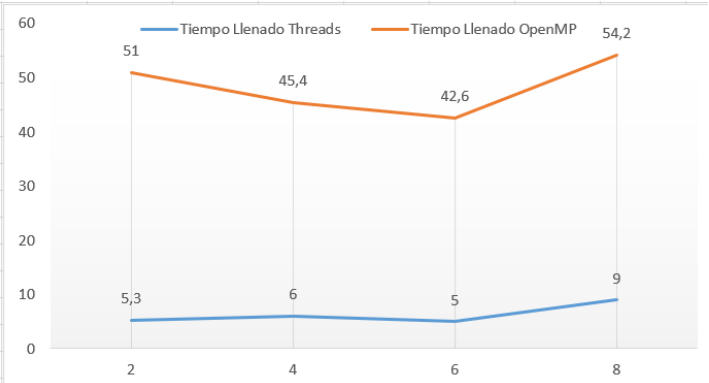

Imagen 6. Tiempo Llenado Del Arreglo

6.3 Tiempo Suma Total

En esta situación es donde se demuestra el tiempo más rápido con respecto a la diferencia entre el tiempo de llenado, y también se puede mostrar que en esta única situación es donde **OpenMP** resulta mas conveniente ya que logra tiempos de ejecución menores que cuando se ocupan **threads**, viendo en mas detalle cuando se ocupan 6 hilos hubo un aumento del tiempo cuando se ocupo **OpenMP** y dentro de lo que común en estas pruebas cuando se llega al uso de 8 hilos es donde el tiempo aumenta mas que cuando se ocupan 2 hilos

hilos	Tiempo Sumado Threads	Tiempo Sumado OpenMP
2	5,7	2,1
4	3,7	1,7
6	3,7	1,9
8	5	3,3

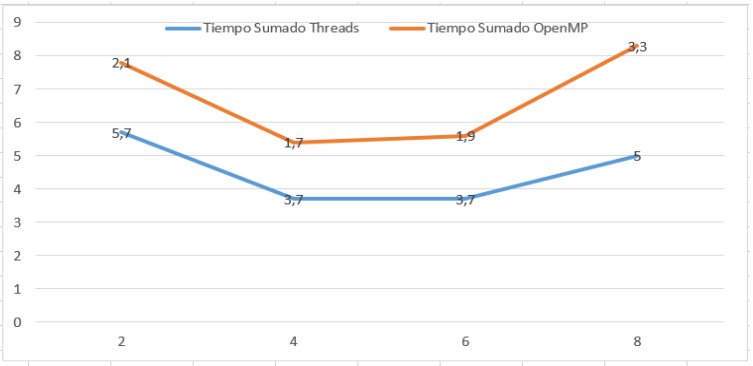

Imagen 7. Tiempo Suma Total

Análisis final, dentro del conjunto de pruebas que se realizaron solo cambiando el número de hilos, se indica que existe una diferencia de tiempos cuando se utiliza **threads** y cuando se utiliza **OpenMP** en general, y cuando se analiza la diferencia entre pruebas con respecto a cuantos hilos fueron utilizados desde 2 a 6 hubo una mejora en el tiempo, pero al llegar a ocupar 8 hilos el tiempo aumenta considerablemente a diferencia de las demás pruebas, y esto no debería suceder a lo más la diferencia de tiempo no debería cambiar, pero en este caso aumenta

7 Conclusión

Este trabajo se basó en la investigación en forma teórica y práctica de lo que tiene que ver con todo el concepto de la programación paralela, se dice todo el concepto ya que se estudió sobre un conjunto de modelos, patrones de diseños y las ventajas y desventajas que hay en el uso de la programación paralela añadiendo también de lo que sabíamos otra forma de uso con respecto a la programación paralela y de la que nos centramos en este taller en concreto que fue el uso de la **API OpenMP**, con respecto a la práctica se analizó el problema con detalle para luego indicar un diseño de la solución y se indicaron variables, métodos y ejemplos de variables a utilizar, ya que este script se basa meramente en la entrada de parámetros estos se explicaron y se indicó su uso correspondiente, también se indicaron un conjunto de resultados de las pruebas que se hicieron y se demostró a nivel de tiempo en los posibles ambientes donde se introducían diferentes parámetros para analizar el comportamiento del programa y se analizó propiamente tal la diferencia de tiempos entre el uso de **threads** y la **API OpenMP** en la que se demostró que esta última en su llenado era mucho más lenta que al usar **threads**, pero en el caso de sumar esta era más rápida. Por lo tanto, en pocas palabras se aprendió que a nivel general la programación paralela en especifico al ocupar la **API OpenMP** se llega indicar una facilidad de implementación en código a diferencia del uso de **threads** pero también dependiendo del problema que se requiera resolver una será mejor que otra en cuanto a tiempo y facilidad de implementación

8 Referencias

[1] Astudillo, G. (2020). Etapa De Llenado. [Diagrama]. Recuperado de fillArrayParallel.pdf