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 y diagramas para entender que aspectos se utilizaran para que la programación paralela sea posible

2.1 Modelos De Programacion Paralela

En este trabajo en particular se habla de llegar a una solución ocupando la programación paralela, pero al momento de buscar sobre esta, nos encontramos que existen varios modelos que se basan en la programación paralela algunos de los cuales son:

. - **Posix Threads**: **POSIX** significa Portable Operating System Interface **(for Unix)**. Es un estándar orientado a facilitar la creación de aplicaciones aplicaciones confiables y portables. La mayoría de las versiones populares de **UNIX (Linux, Mac OS X)** cumplen este estándar en gran medida. La biblioteca para el manejo de hilos en **POSIX** es **pthread**.

. - **MPI**: Es un estándar de programación en paralelo mediante paso de mensajes que permite crear programas portables y eficientes, es una biblioteca que incluye interfaces para **FORTRAN, C y C++.**
Define varias formas de comunicación lo que permite programar de manera natural cualquier algoritmo en paralelo

. - **PVM**: La máquina paralela virtual es una máquina que no existe, pero un API apropiado que permite programar como si existiese. El modelo abstracto que nos permite usar el API de la **PVM** consiste en una máquina multiprocesador completamente escalable (es decir, que podemos aumentar y disminuir el número de procesadores en caliente). Para ello, nos va a ocultar la red que estemos empleando para conectar nuestras máquinas, así como las máquinas de la red y sus características específicas

. - **Intel TBB (Intel Threading Building Blocks)**: Modelo de programación paralela basado en rutinas que utilizan hilos.
Provee una serie de plantillas, tipos de datos y algoritmos. **TBB** está pensado de cara al rendimiento, por lo que es compatible con otras bibliotecas y paquetes.

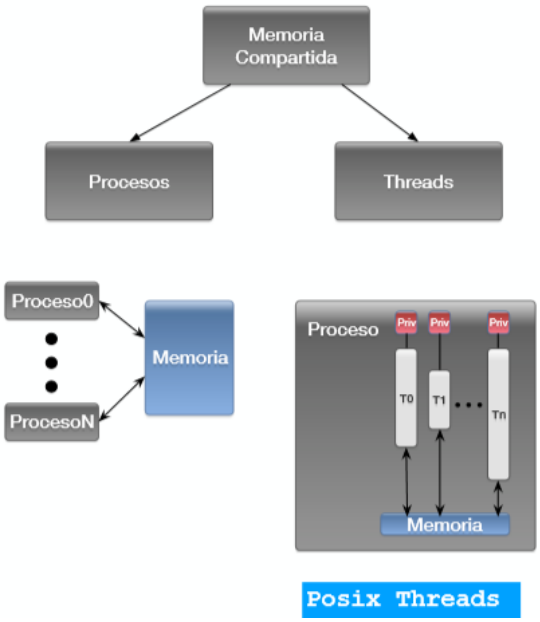

Figura 1. Posix Threads

2.2 Ventajas y Desventajas

Ventajas	Desventajas
Resuelve problemas que no se pueden resolver en un tiempo razonable	Mayor consumo de energía
Permite ejecutar problemas de un orden y complejidad mayor	Dificultad para lograr una buena sincronización y comunicación entre las tareas
Permite ejecutar código de manera más rápida (aceleración)	Retardos ocasionados por comunicación ente tareas
Permite la ejecución de varias instrucciones en simultáneo	Número de componentes usados es directamente proporcional a los fallos potenciales
Permite dividir una tarea en partes independientes	

Tabla 1. Ventajas y Desventajas

2.3 Hilos y Sincronización

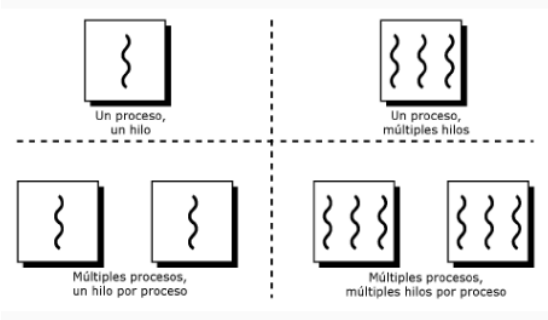

Figura 2. Hilos

Un proceso pesado padre puede convertirse en varios procesos livianos hijos, ejecutados de manera concurrente. Cada uno de estos procesos livianos se conoce como hilo. Estos se comunican entre ellos a través de la memoria global.

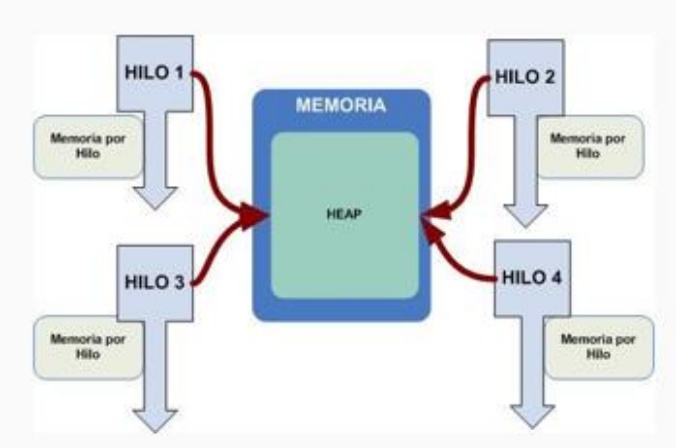

Figura 3. Sincronización

Los programas en paralelo necesitan la coordinación de procesos e hilos, para que haya una ejecución correcta. Los métodos de coordinación y sincronización en la programación paralela están fuertemente asociados a la manera en que los procesos o hilos intercambian información, y esto depende de cómo está organizada la memoria en el hardware.

2.4 Patrones De Diseño Paralelo

Se mostrarán un conjunto de patrones, no se mostrarán absolutamente todos ya que son varios en existencia y el que se utiliza en este trabajo será el patrón **Fork-Join**

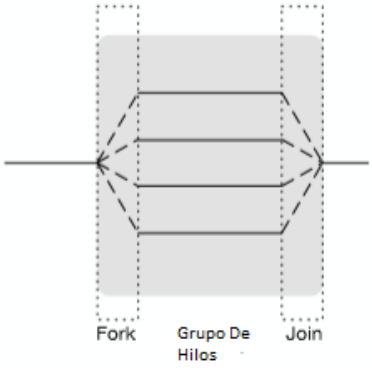

Figura 4. Fork-Join

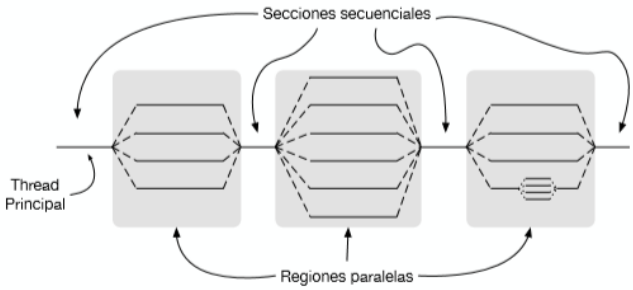

Figura 5. Fork-Join

Fue diseñado para la ejecución de tareas que pueden dividirse en otras subtareas más pequeñas, ejecutándose estas en paralelo y combinando posteriormente sus resultados para obtener el resultado de la tarea única. Las subtareas deberán ser independientes unas de otras, y no contendrán estado.

Realiza la paralelización de tareas de forma recursiva, aplicando el principio Divide y Vencerás. Existirá un umbral bajo el cual una tarea será indivisible, definido por el tamaño de la misma. Una vez el tamaño de las subtareas llegue al umbral, el rendimiento de estas disminuirá en caso de seguir dividiéndolas.

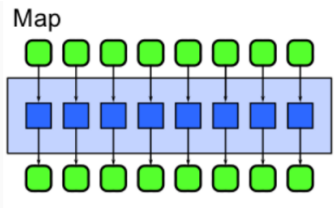

Figura 6. Map

Es un patrón que replica una función sobre todos los elementos de un conjunto de entrada. La función que está siendo replicada se llama función elemental, dada que la misma se aplica a una colección real de datos

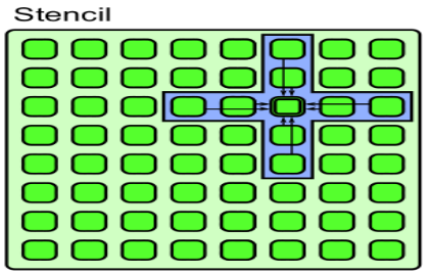

Figura 7. Stencil

Stencil es una generalización del patrón de **Map**, en el cual una función elemental tiene acceso no solo a un elemento del conjunto de entrada sino también a un conjunto de "vecinos".
Como la estructura de datos no es infinita se deben tener en cuenta el manejo de excepciones para los bordes de la misma

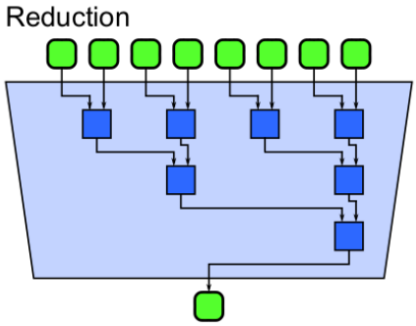

Figura 8. Reduction

Una reducción combina cada elemento de una colección en uno solo utilizado una función asociativa conocida como función combinatoria. Como es asociativa las tareas se pueden distribuir de muchas maneras y si la función resultara ser también conmutativa el número de posibilidades aumentaría aún más.

2.5 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 2. 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 dinamico que se llena para ser utilizado con respecto a las librerias 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

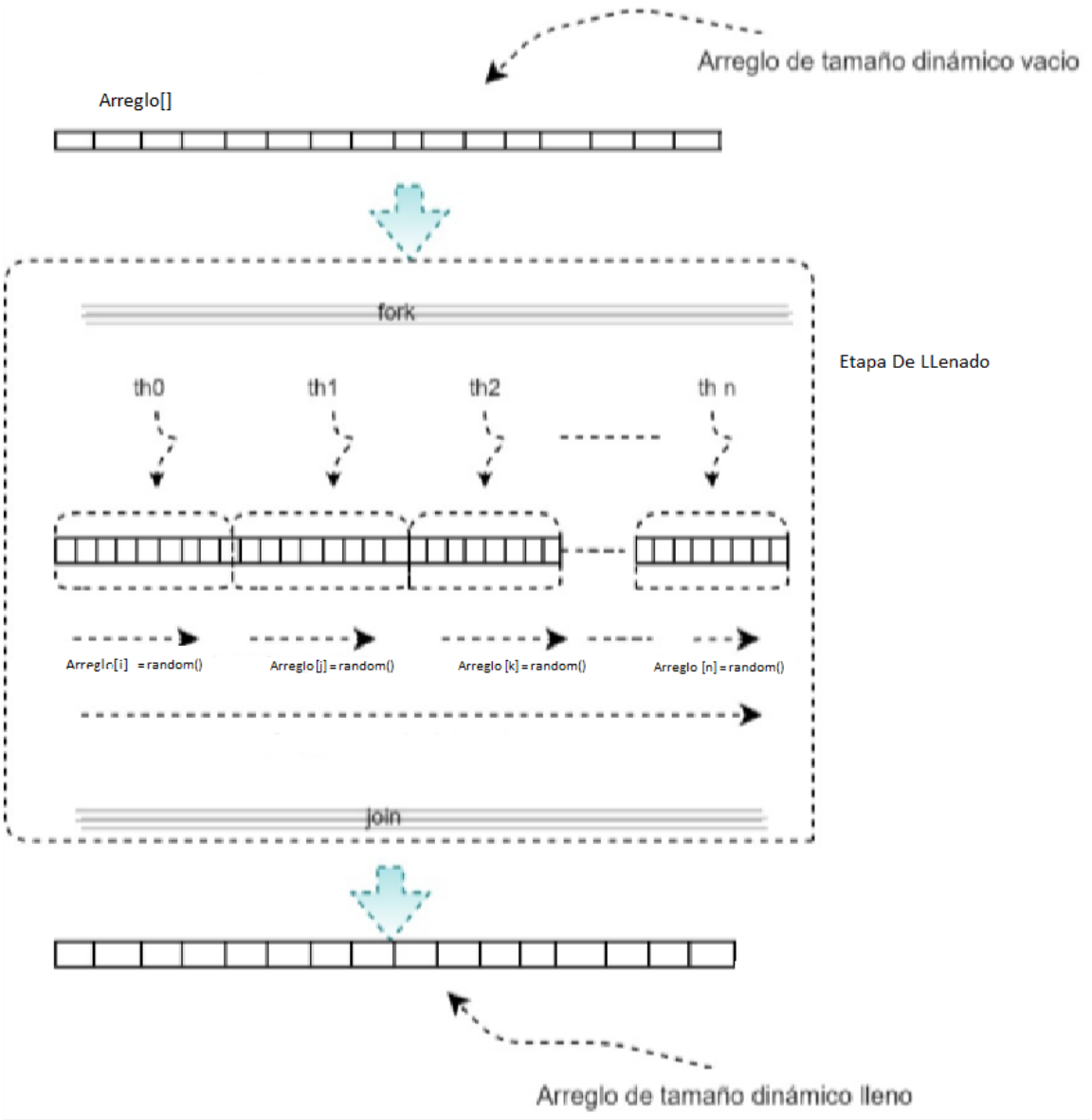

Figura 9. Modulo 1

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

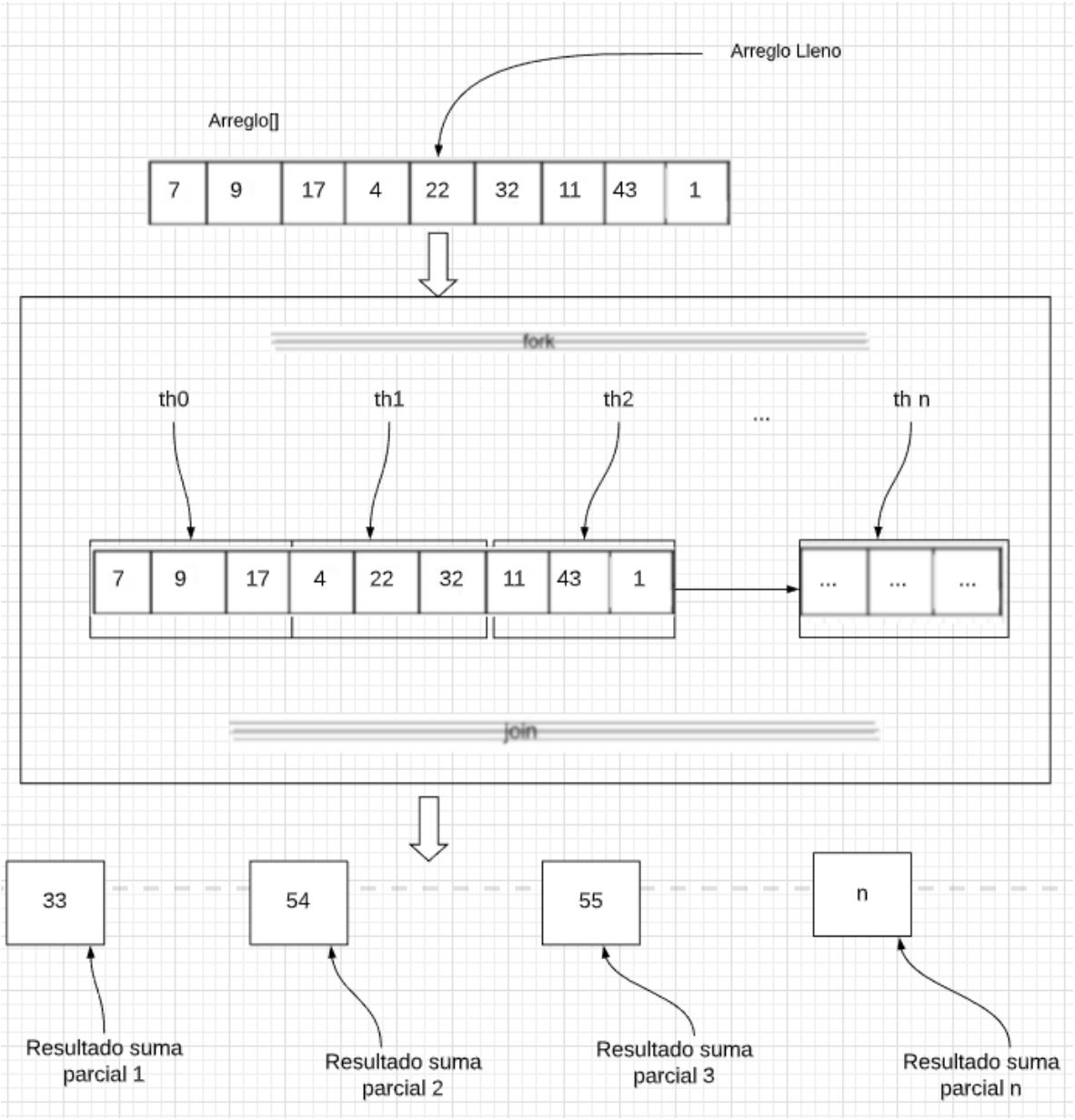

Figura 10. Suma del arreglo

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

```
ignacio@ignacio:~/tso03/U2/pruebas/TSSOO-taller02/src$ sudo apt-get install libomp-dev
[sudo] password for ignacio:
Reading package lists... Done
Building dependency tree
Reading state information... Done
libomp-dev is already the newest version (5.0.1-1).
0 upgraded, 0 newly installed, 0 to remove and 60 not upgraded.
```

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

```
ignacio@ignacio:~/tso03/U2/Pruebafinal$ ls
ignacio@ignacio:~/tso03/U2/Pruebafinal$ ls -la
total 8
drwxrwxr-x  2 ignacio ignacio 4096 Jul 24 09:40 .
drwxrwxr-x 11 ignacio ignacio 4096 Jul 24 09:40 ..
```

Imagen 2.

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

```
ignacio@ignacio:~/tso03/U2/Pruebafinal$ git clone https://github.com/Ignacio-my/TSSOO-taller03.git
Cloning into 'TSSOO-taller03'...
remote: Enumerating objects: 14, done.
remote: Counting objects: 100% (14/14), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 14 (delta 0), reused 11 (delta 0), pack-reused 0
Unpacking objects: 100% (14/14), done.
ignacio@ignacio:~/tso03/U2/Pruebafinal$ ls
TSSOO-taller03
ignacio@ignacio:~/tso03/U2/Pruebafinal$ ls -la
total 12
drwxrwxr-x  3 ignacio ignacio 4096 Jul 24 09:41 .
drwxrwxr-x 11 ignacio ignacio 4096 Jul 24 09:40 ..
drwxrwxr-x  4 ignacio ignacio 4096 Jul 24 09:41 TSSOO-taller03
ignacio@ignacio:~/tso03/U2/Pruebafinal$ cd TSSOO-taller03/
ignacio@ignacio:~/tso03/U2/Pruebafinal/TSSOO-taller03$ ls
Makefile  README.md  src
ignacio@ignacio:~/tso03/U2/Pruebafinal/TSSOO-taller03$ ls -la
total 24
drwxrwxr-x 4 ignacio ignacio 4096 Jul 24 09:41 .
drwxrwxr-x 3 ignacio ignacio 4096 Jul 24 09:41 ..
drwxrwxr-x 8 ignacio ignacio 4096 Jul 24 09:41 .git
-rw-rw-r-- 1 ignacio ignacio  194 Jul 24 09:41 Makefile
-rw-rw-r-- 1 ignacio ignacio   17 Jul 24 09:41 README.md
drwxrwxr-x 4 ignacio ignacio 4096 Jul 24 09:41 src
```

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

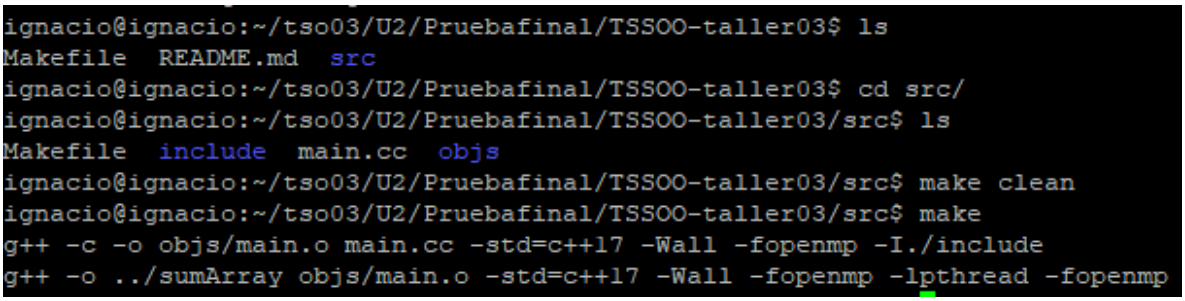

Imagen 4.

6 Resultados

Conjunto De resultados de la demostración de 3 pruebas con el mismo parámetro el cual sería 2 hilos, 1000000 de elementos y un rango de números aleatorios de 10 a 100, por lo que se demuestra hay una demora promedio total con la utilización de **OpenMP** 5 veces mayor a diferencia de la que se realiza por hilos, también se puede analizar mas en detalle que la suma en **OpenMP** se realiza más rápido que la de hilos, pero en el llenado es donde **OpemMP** se demora mas

```
ignacio@ignacio:~/tso03/U2/Pruebafinal/TSSOO-taller03$ ./sumArray -N 1000000 -t 2 -l 10 -L 100
Conjunto de Numeros Del Arreglo: 1000000
Numero de Hilos  : 2
Limite Inferior Rango Aleatorio: 10
Limite Superior Rango Aleatorio: 100
Suma Total en Paralelo: 55004497
TiempoLLenadoParaleloThreads:5[ms]
TiempoSumaParalelo:5[ms]
TiempoEjecucionTotal:10[ms]
TiempoLlenadoParaleloOpenMP:52.737[ms]
TiempoSumaParaleloOpenMP:2.3293[ms]
TiempoEjecucionTotalOpenMP:55.0663[ms]
ignacio@ignacio:~/tso03/U2/Pruebafinal/TSSOO-taller03$ ./sumArray -N 1000000 -t 2 -l 10 -L 100
Conjunto de Numeros Del Arreglo: 1000000
Numero de Hilos  : 2
Limite Inferior Rango Aleatorio: 10
Limite Superior Rango Aleatorio: 100
Suma Total en Paralelo: 54972323
TiempoLLenadoParaleloThreads:6[ms]
TiempoSumaParalelo:5[ms]
TiempoEjecucionTotal:11[ms]
TiempoLlenadoParaleloOpenMP:53.1092[ms]
TiempoSumaParaleloOpenMP:2.61481[ms]
TiempoEjecucionTotalOpenMP:55.724[ms]
ignacio@ignacio:~/tso03/U2/Pruebafinal/TSSOO-taller03$ ./sumArray -N 1000000 -t 2 -l 10 -L 100
Conjunto de Numeros Del Arreglo: 1000000
Numero de Hilos  : 2
Limite Inferior Rango Aleatorio: 10
Limite Superior Rango Aleatorio: 100
Suma Total en Paralelo: 54988082
TiempoLLenadoParaleloThreads:5[ms]
TiempoSumaParalelo:7[ms]
TiempoEjecucionTotal:12[ms]
TiempoLlenadoParaleloOpenMP:47.1626[ms]
TiempoSumaParaleloOpenMP:1.43367[ms]
TiempoEjecucionTotalOpenMP:48.5962[ms]
```

Imagen 5. 2 hilos

Conjunto De resultados de la demostración de 3 pruebas con el mismo parámetro el cual sería 4 hilos, 1000000 de elementos y un rango de números aleatorios de 10 a 100, por lo que se indica la suma siempre varia pero no de gran manera, y prácticamente en los resultados del tiempo total hay una diferencia mínima, y con respecto al tiempo de llenado y sumado, el de llenado se demorara generalmente más que el de sumado, y la diferencia con respecto a los threads que se ocupan es que se nota una leve mejora de tiempo ya que se aumentaron 2 threads a diferencia de la prueba anterior

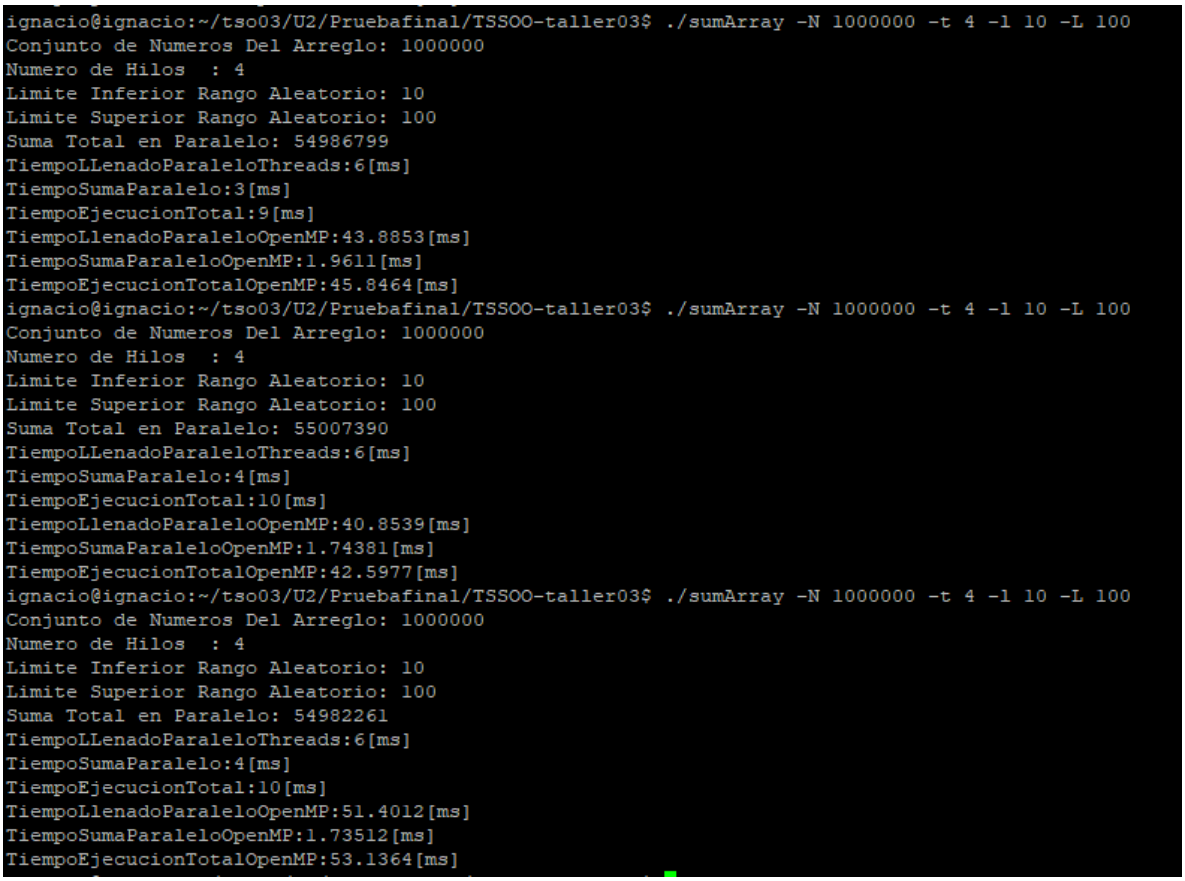

Imagen 6. 4 hilos

Conjunto De resultados de la demostración de 3 pruebas con el mismo parámetro el cual sería 6 hilos, 1000000 de elementos y un rango de números aleatorios de 10 a 100, aquí se logra ver una diferencia en el tiempo total de nuevo, pero esta sigue siendo muy mínima con respecto a la anterior ya que se analizaron los promedios y en este caso fue de 3[ms] mas rápido el tiempo total de **OpenMP**

```
ignacio@ignacio:~/tso03/U2/Pruebafinal/TSSOO-taller03$ ./sumArray -N 1000000 -t 6 -l 10 -L 100
Conjunto de Numeros Del Arreglo: 1000000
Numero de Hilos  : 6
Limite Inferior Rango Aleatorio: 10
Limite Superior Rango Aleatorio: 100
Suma Total en Paralelo: 54958406
TiempoLLenadoParaleloThreads:4[ms]
TiempoSumaParalelo:4[ms]
TiempoEjecucionTotal:8[ms]
TiempoLlenadoParaleloOpenMP:36.4805[ms]
TiempoSumaParaleloOpenMP:1.45894[ms]
TiempoEjecucionTotalOpenMP:37.9394[ms]
ignacio@ignacio:~/tso03/U2/Pruebafinal/TSSOO-taller03$ ./sumArray -N 1000000 -t 6 -l 10 -L 100
Conjunto de Numeros Del Arreglo: 1000000
Numero de Hilos  : 6
Limite Inferior Rango Aleatorio: 10
Limite Superior Rango Aleatorio: 100
Suma Total en Paralelo: 55051347
TiempoLLenadoParaleloThreads:6[ms]
TiempoSumaParalelo:4[ms]
TiempoEjecucionTotal:10[ms]
TiempoLlenadoParaleloOpenMP:46.5697[ms]
TiempoSumaParaleloOpenMP:2.92828[ms]
TiempoEjecucionTotalOpenMP:49.498[ms]
ignacio@ignacio:~/tso03/U2/Pruebafinal/TSSOO-taller03$ ./sumArray -N 1000000 -t 6 -l 10 -L 100
Conjunto de Numeros Del Arreglo: 1000000
Numero de Hilos  : 6
Limite Inferior Rango Aleatorio: 10
Limite Superior Rango Aleatorio: 100
Suma Total en Paralelo: 55001587
TiempoLLenadoParaleloThreads:5[ms]
TiempoSumaParalelo:3[ms]
TiempoEjecucionTotal:8[ms]
TiempoLlenadoParaleloOpenMP:44.9608[ms]
TiempoSumaParaleloOpenMP:1.51227[ms]
TiempoEjecucionTotalOpenMP:46.473[ms]
```

Imagen 7. 6 hilos

Conjunto De resultados de la demostración de 3 pruebas con el mismo parámetro el cual sería 8 hilos, 1000000 de elementos y un rango de números aleatorios de 10 a 100, en este conjunto de pruebas se nota una diferencia teóricamente inusual ya que al aumentar el número de hilos también se aumento el tiempo total de ejecución con respecto a las dos maneras posibles que se sacaron ya que con respecto a la forma de hilos aumento el tiempo a diferencia de las pruebas anteriores y también con respecto a la forma del **OpenMP**

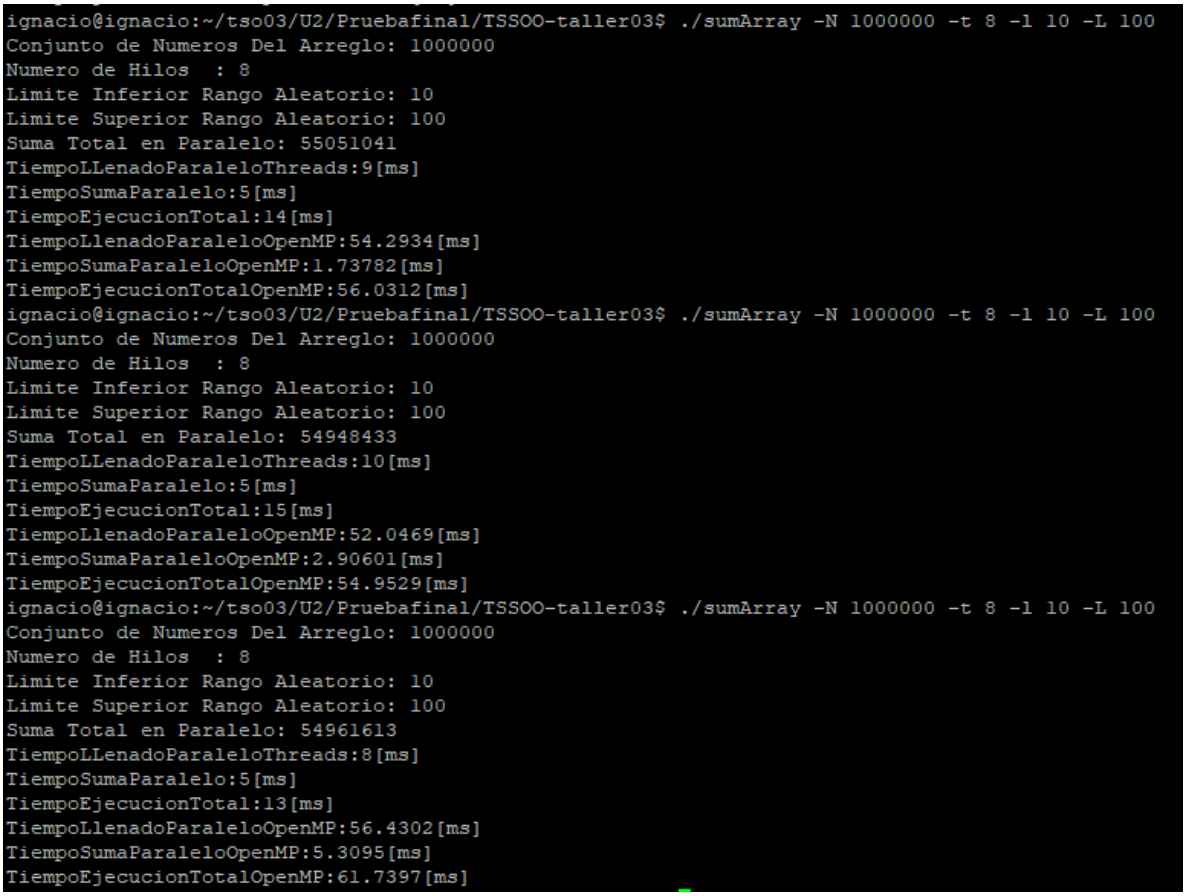

Imagen 8. 8 hilos

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