Informe técnico taller 03

Taller de Sistemas Operativos

Profesor: Gabriel Astudillo Muñoz

Alumno: Sebastián González Morales

Escuela de Ingeniería Civil Informática,
Universidad de Valparaíso
24 de julio de 2020

1 Introducción

En este informe se explicará el contexto, el diseño, parte de la implementación, pruebas, resultados y conclusiones de un problema que se resolverá a través de programación paralela con el fin de dividir tareas en partes independientes para obtener resultados en menos tiempo.

El objetivo general es implementar un programa que llene un arreglo de números enteros y luego los sume. Ambas tareas se harán en forma paralela, implementadas con OpenMP.

OpenMP es una especificación para un conjunto de directivas de compilación, rutinas de biblioteca y variables de entorno que se pueden usar para especificar paralelismo de alto nivel en programas como **Fortran** y **C / C ++** [1].

Hay varias razones por las que es conveniente usarlo: (a) Es el estándar más extendido para sistemas SMP (Multiprocesamiento simétrico), admite 3 lenguajes diferentes (Fortran, C, C++) y ha sido implementado por muchos proveedores. (b) Es una especificación relativamente pequeña y simple y admite paralelismo incremental. (c) Se realiza mucha investigación, manteniéndolo actualizado con los últimos desarrollos de hardware [2].

OpenMP opera en un modelo de ejecución paralela de Fork – Join, esto se muestra en la Figura 1. Todos los programas OpenMP comienzan como un proceso único que se denomina hilo maestro. Este hilo maestro se ejecuta secuencialmente hasta que se encuentra una región paralela. En este punto, el hilo maestro se "bifurca" en varios hilos de trabajo paralelos. Las instrucciones en la región paralela son ejecutadas por este equipo de subprocesos de trabajo. Al final de la región paralela, los hilos se sincronizan y se unen para volver a ser el hilo maestro único. Por lo general, ejecutaría un subproceso por procesador, pero es posible ejecutar más. La paralelización con OpenMP se especifica a través de directivas del compilador que están incrustadas en el código fuente [3].

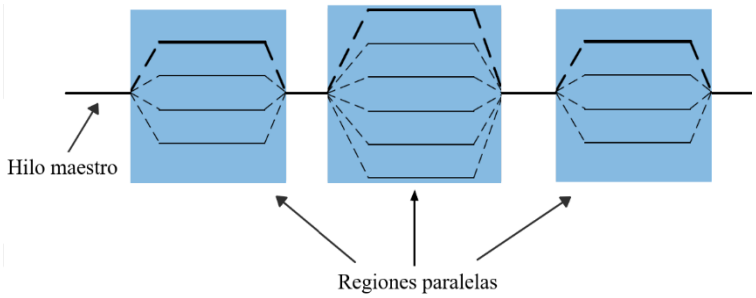

Figura 1.- Modelo fork-join

El documento está estructurado de la siguiente forma, en la sección dos se presenta el problema a resolver, en la sección tres se dará una descripción del diseño y una parte de la implementación, se presentarán diagramar para que haya un mejor entendimiento de lo que se piensa hacer. En la sección cuatro se mostrará que se hizo para poder trabajar con openMp en el sistema operativo de la máquina virtual. En la sección cinco, se muestran pruebas de desempeño realizadas tanto para la implementación secuencial con la implementación con OpenMP, tanto con esta con la implementación realizada en el taller 2 (POSIX Thread). En la sección seis se mostrarán las conclusiones tanto de este taller con el taller dos. Finalmente, en la sección siete se muestran las referencias utilizadas.

2 Problema

El problema está compuesto en dos módulos, el primero consiste en llenar un arreglo unidimensional de números aleatorios del **tipo uint32_t** y el segundo debe sumar el contenido del arreglo. Posteriormente se implementará y luego se realizarán pruebas de desempeño para ver el comportamiento del tiempo de ejecución de ambos módulos según el tamaño del arreglo, la cantidad de threads utilizados y el rango de números aleatorios, en esta ocasión se comparara con el taller anterior, específicamente con la implementación de threads, y no con la forma secuencial. La implementación será en **C++ versión 2017**.

Tanto el tamaño del arreglo, la cantidad de threads utilizados y el rango de números aleatorios serán en forma dinámica, configurable a través de parámetros de entrada del programa.

2.1 Forma de uso:

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

Parámetros:

```
-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.
```

2.2 Ejemplo de uso:

1) Crea un arreglo de 2000000 posiciones, con 5 threads. Los números enteros aleatorios están en el rango [20,60]

```
./sumArray -N 2000000 -t 5 -l 20 -L 60
```

2) Muestra la ayuda y termina

```
./sumArray -h
./sumArray
```

3 Diseño y implementación.

3.1 Descripción general

Como se dijo anteriormente, el tamaño del arreglo, la cantidad de threads utilizados y el rango de números aleatorios serán en forma dinámica, configurable a través de parámetros de entrada del programa y es global al proceso. Esto quiere decir que es visible para el número de threas creados dentro de él. La solución de la primera parte del módulo se denomina “Etapa de llenado” (ver Figura 2). Cada thread conoce los índices de inicio y fin donde debe almacenar un número aleatorio.

Cuando el arreglo termina de llenarse, se pasa al segundo modulo, este ira sumando los números antes ingresados en el arreglo, esta parte tiene el nombre de “Etapa de suma”(ver Figura 5). En esa figura se utilizaron 2 threads y el arreglo estaba compuestos por números entre 0 y 50.

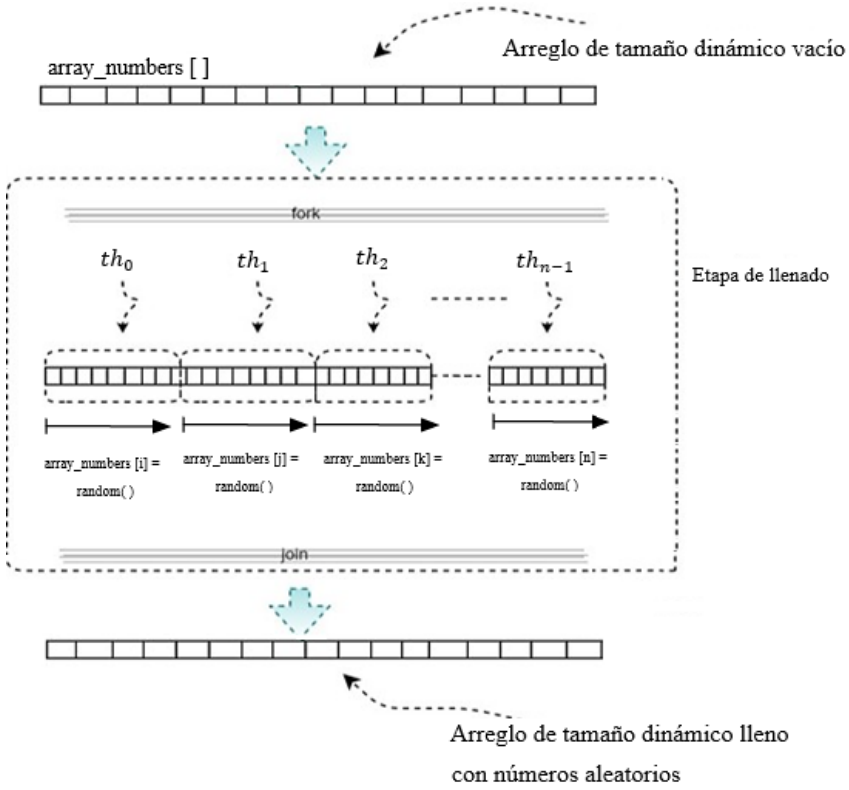

Figura 2

3.2 Etapa de llenado

El llenado del arreglo global **`array_numbers[]`**, se realizará a través de una función denominada **`fillArray`**, que tiene cuatro parámetros: el primero indica el índice de inicio (**`beginIdx`**), el segundo es para el índice de fin (**`endIdx`**), esto permite que la función sea llamada en forma secuencial o parcial por los threads. El tercer y cuarto parámetro es para indicar el límite inferior y el límite superior de los números que se ingresaran al arreglo, lo anteriormente descrito corresponde al taller 2, en este caso se utiliza un directiva for, la cual identifica un constructor de trabajo compartido el cual especifica que las iteraciones del ciclo asociado deben ser ejecutadas en paralelo, y se utiliza la cláusula num_threads, si está presente, entonces la expresión entera en la cláusula es el número de threads requerido, aquí es donde se implementan los parámetros ingresados.

Para las pruebas de funcionamiento, se probará con la función **`std::uniform_int_distribution<>`** disponible en la biblioteca **`<random>`**.

A continuación, en la figura 3, se puede ver una captura del código de la etapa de llenado.

```
====SUMA===Secuencial
start = std::chrono::high_resolution_clock::now();
for (size_t i=0; i< totalElementos; i++){
    acumSecuencial+= array_numbers[i];
}
```

Figura 3.

3.3 Etapa de suma

Esta etapa se realizará a través de una función llamada **sumasParalelo**, la cual tendrá dos parámetros de entrada, el índice de inicio y el índice fin de los números que irán sumando el thread. En esta función se irán sumando las sumas parciales para obtener la suma total. Lo anteriormente descrito corresponde al taller 2, en esta ocasiónse utiliza la directiva for descrita anteriormente en la etapa de llenado, y también se utiliza un operador de reducción, este realiza una operación de reducción sobre las variables que aparecen en la lista utilizando el operador +. A continuación, en la figura 4, se puede ver una captura del código de la etapa de suma.

```
=====SUMA DE NUMEROS===OpenMP

#pragma omp parallel for reduction(+:acumParalel) num_threads(numThreads)
for(size_t i = 0; i < totalElementos; ++i){
        acumParalel += array_numbers[i];
}
```

Figura 4.

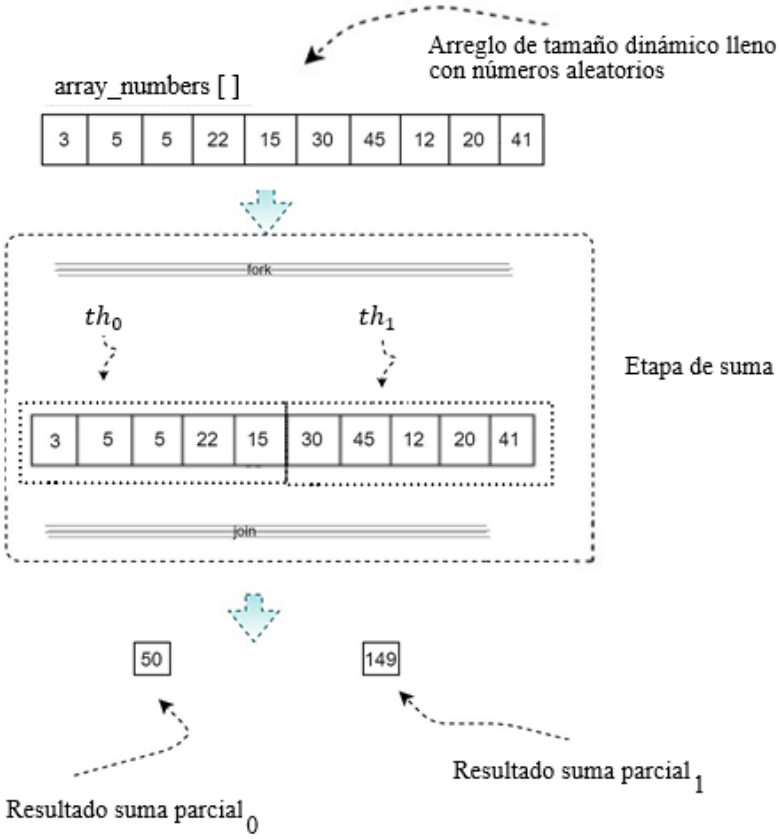

Figura 5

4 Verificando si se tiene instalado OpenMp.

El compilador que posee el sistema operativo de la máquina virtual en la cual se realizara la implementación es **GCC**, el cual se instaló al principio del ramo, por lo tanto, se ingresó a la página oficial de OpenMp y se verifico que esta Api admitiera este compilador, en la figura 6, se puede mostrar una captura de la página en donde se comprueba, lo anteriormente señalado.

Vendor/Source	Compiler/Language	Information
GNU	GCC C/C++/Fortran	Free and open source – Linux, Solaris, AIX, MacOSX, Windows, FreeBSD, NetBSD, OpenBSD, DragonFly BSD, HPUX, RTEMS ▪ From GCC 4.2.0, OpenMP 2.5 is fully supported for C/C++/Fortran. ▪ From GCC 4.4.0, OpenMP 3.0 is fully supported for C/C++/Fortran. ▪ From GCC 4.7.0, OpenMP 3.1 is fully supported for C/C++/Fortran. ▪ In GCC 4.9.0, OpenMP 4.0 is supported for C and C++, but not Fortran. ▪ From GCC 4.9.1, OpenMP 4.0 is fully supported for C/C++/Fortran. ▪ From GCC 6.1, OpenMP 4.5 is fully supported for C and C++. ▪ From GCC 7.1, OpenMP 4.5 is partially supported for Fortran. ▪ From GCC 9.1, OpenMP 5.0 is partially supported for C and C++. Compile with -fopenmp to enable OpenMP. Online documentation: https://gcc.gnu.org/onlinedocs/libgomp/ OpenMP support history: https://gcc.gnu.org/projects/gomp/

Figura 6

Algo a destacar es que en la información que nos muestra esta página, es que para compilar se debe ingresar la sentencia **-fopenmp,** la cual se agregara al archivo Makefile correspondiente a nuestra implementación

Posteriormente, en la máquina virtual se ingresó el comando apt show libomp-dev, para verificar que el paquete de desarrollo LLVM OpenMP runtime, estuviera en el sistema operativo. Este paquete estaba instalado.

5 Pruebas de desempeño según cantidad de hilos y resultados.

5.1 Implementación secuencial vs OpenMP

Las pruebas de desempeño se realizaron para evaluar el comportamiento del algoritmo bajo distintos parámetros de entrada, para este caso se toma el tiempo secuencial y el tiempo con OpenMP con uno, dos, tres y cuatro hilos, en las etapas de llenado, de suma y el tiempo total, también se calcula el índice de desempeño SpeedUp. El tamaño del arreglo es 100000000, los números que se generaran aleatoriamente son entre 10 y 50.

Para medir el tiempo de ejecución en un bloque de código se utilizó los métodos std::crono, disponible en **<crono>.**

En la figura 7, se puede apreciar que se utiliza un hilo, el tiempo secuencial y el tiempo con OpenMp son muy parecidos, no hay una diferencia notable.

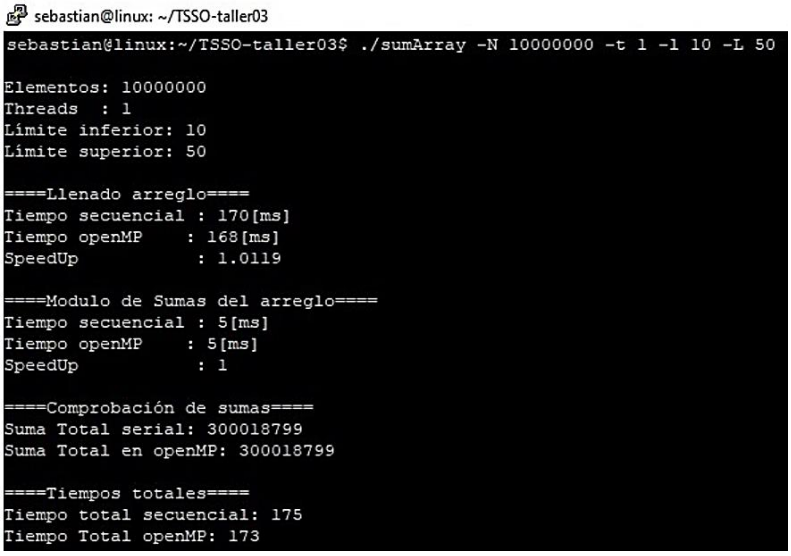

Figura 7

En la figura 8, se puede apreciar que se utilizan 2 hilos, y aquí si se puede ver una diferencia mayor, principalmente en la etapa de llenado, mientras que para la etapa de llenado, no hay una diferencia tan grande, la diferencia entre el tiempo total secuencial y el tiempo con openMp es de 70 [ms].

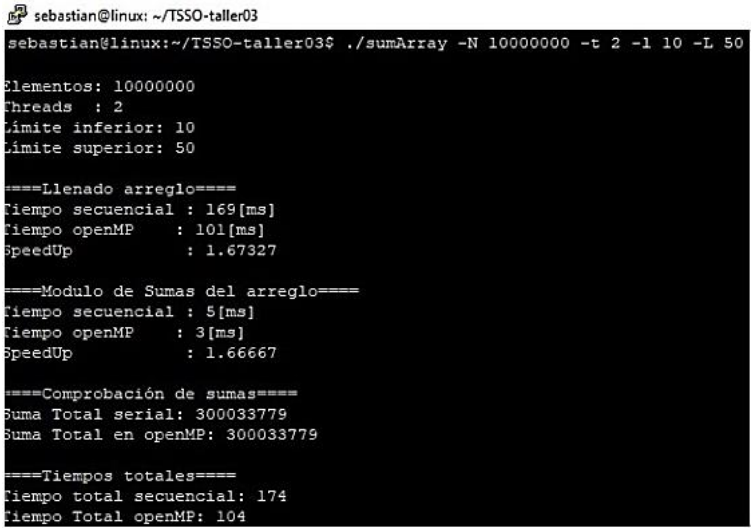

Figura 8

En la figura 9, se puede apreciar que se utilizan 3 hilos, en la etapa de llenado del arreglo hay un mayor tiempo secuencial respecto al tiempo con OpenMP, speedUp igual a 1,91, en la etapa de suma speedUp igual a 1,5.

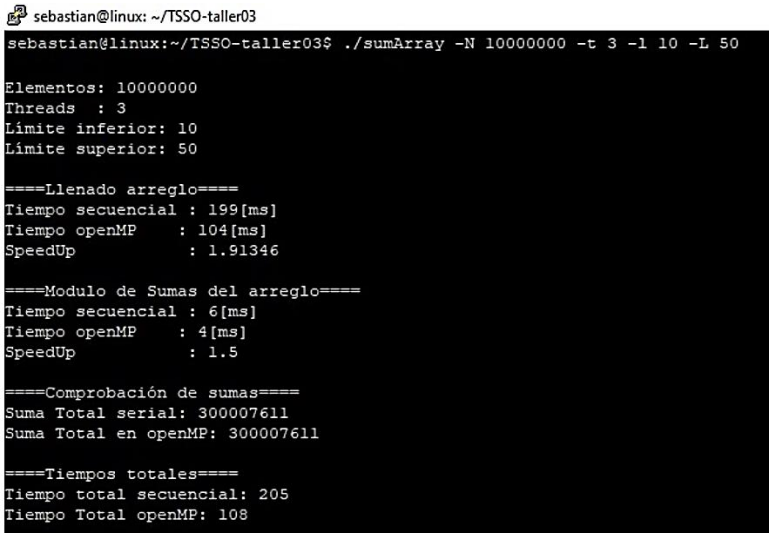

Figura 9

En la figura 10, finalmente se utilizan 4 hilos, se puede apreciar que la diferencia entre los tiempos total secuencial y con openMp es igual a 53 [ms].

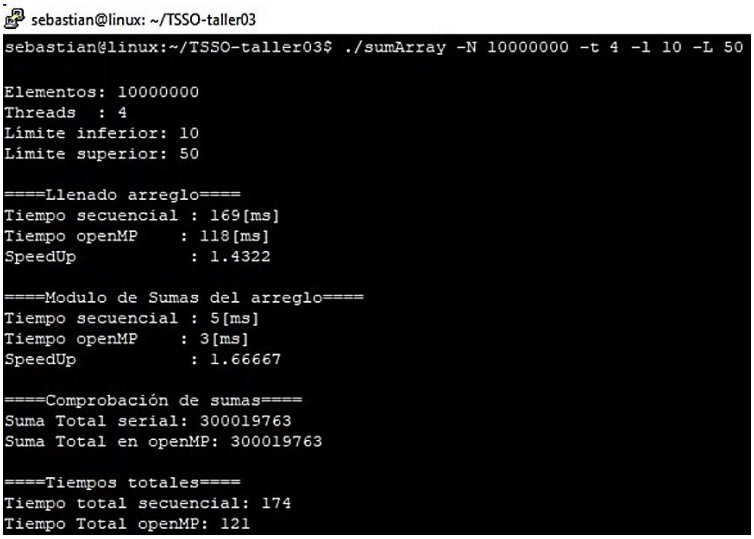

Figura 10

Hay que destacar que, en las 4 imágenes, tanto la suma total serial y la suma con OpenMp, es la misma, por tanto, hay una consistencia en los datos

En la figura 11, se puede apreciar un grafico realizado mediante la obtención de datos, modificando la cantidad de threads, donde se puede ver que la implementación con OpenMp es más rápida que la secuencial.

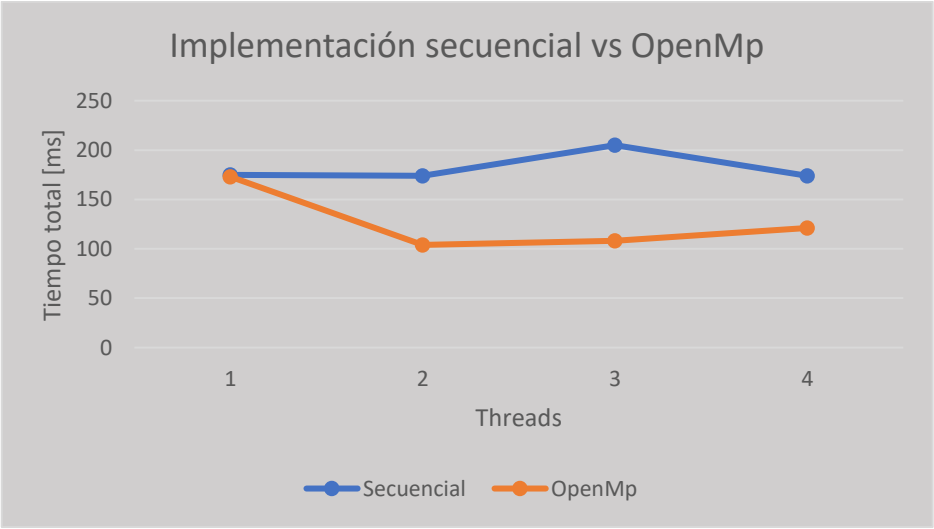

Figura 11.- Gráfico implementación secuencial vs OpenMp

5.2 Implementación POSIX Threads vs OpenMP.

A continuación, se hará una comparación entre los tiempos de la implementación con **POSIX Threads** (taller 2) y la implementación con OpenMP, esto se realizara ejecutando el código del taller 2 y comparándolo con los datos obtenidos anteriormente en este taller.

En la figuras 12, 13,14 y 15se puede ver la ejecución del taller 2, se utilizan uno, dos, tres y cuatros hilos, el tamaño del arreglo es el mismo que para la ejecución anterior con OpenMP, 100000000 y los números que se generaran aleatoriamente también, 10 y 50, en esta ocasión, no se hablara del tiempo secuencial del taller 2, solo se intenta ver cual implementación en forma paralela es más eficiente, POSIX threads o OpenMp,

```
sebastian@linux:~/TSSOO-taller02$ ./sumArray -N 10000000 -t 1 -l 10 -L 50

Elementos: 10000000
Threads  : 1
Límite inferior: 10
Límite superior: 50

====Modulo de Llenado del arreglo====
Tiempo secuencial :183[ms]
Tiempo threads    :185[ms]
SpeedUp           :0.989189

====Modulo de Sumas del arreglo====
Tiempo secuencial :6[ms]
Tiempo threads    :6[ms]
SpeedUp           :1

====Comprobación de sumas====
Suma Total serial: 299991958
Suma Total en Paralelo: 299991958

====Tiempos totales====
Tiempo total secuencial: 189
Tiempo Total paralelo: 191
```

Figura 12.- Implementación con un hilo

```
sebastian@linux:~/TSSOO-taller02$ ./sumArray -N 10000000 -t 2 -l 10 -L 50

Elementos: 10000000
Threads  : 2
Límite inferior: 10
Límite superior: 50

====Modulo de Llenado del arreglo====
Tiempo secuencial :171[ms]
Tiempo threads    :88[ms]
SpeedUp           :1.94318

====Modulo de Sumas del arreglo====
Tiempo secuencial :5[ms]
Tiempo threads    :6[ms]
SpeedUp           :0.833333

====Comprobación de sumas====
Suma Total serial: 299997121
Suma Total en Paralelo: 299997121

====Tiempos totales====
Tiempo total secuencial: 176
Tiempo Total paralelo: 94
```

Figura 13.- Implementación con dos hilos

```
sebastian@linux:~/TSSOO-taller02$ ./sumArray -N 10000000 -t 3 -l 10 -L 50

Elementos: 10000000
Threads  : 3
Límite inferior: 10
Límite superior: 50

====Modulo de Llenado del arreglo====
Tiempo secuencial :190[ms]
Tiempo threads    :95[ms]
SpeedUp           :2

====Modulo de Sumas del arreglo====
Tiempo secuencial :5[ms]
Tiempo threads    :7[ms]
SpeedUp           :0.714286

====Comprobación de sumas====
Suma Total serial: 300034586
Suma Total en Paralelo: 300034586

====Tiempos totales====
Tiempo total secuencial: 195
Tiempo Total paralelo: 102
```

Figura 14.- Implementación con tres hilos.

```
sebastian@linux:~/TSSOO-taller02$ ./sumArray -N 10000000 -t 4 -l 10 -L 50

Elementos: 10000000
Threads  : 4
Límite inferior: 10
Límite superior: 50

====Modulo de Llenado del arreglo====
Tiempo secuencial :183[ms]
Tiempo threads    :90[ms]
SpeedUp           :2.03333

====Modulo de Sumas del arreglo====
Tiempo secuencial :5[ms]
Tiempo threads    :8[ms]
SpeedUp           :0.625

====Comprobación de sumas====
Suma Total serial: 299946272
Suma Total en Paralelo: 299946272

====Tiempos totales====
Tiempo total secuencial: 188
Tiempo Total paralelo: 98
```

Figura 15.- Implementación con cuatro hilos

En la figura 16, se puede ver un grafico con la comparación de los tiempos totales entre la implementación con POSIX Threads y con OpenMp, en la cual se puede apreciar que no hay una diferencia tan grande, en las distintas ejecuciones.

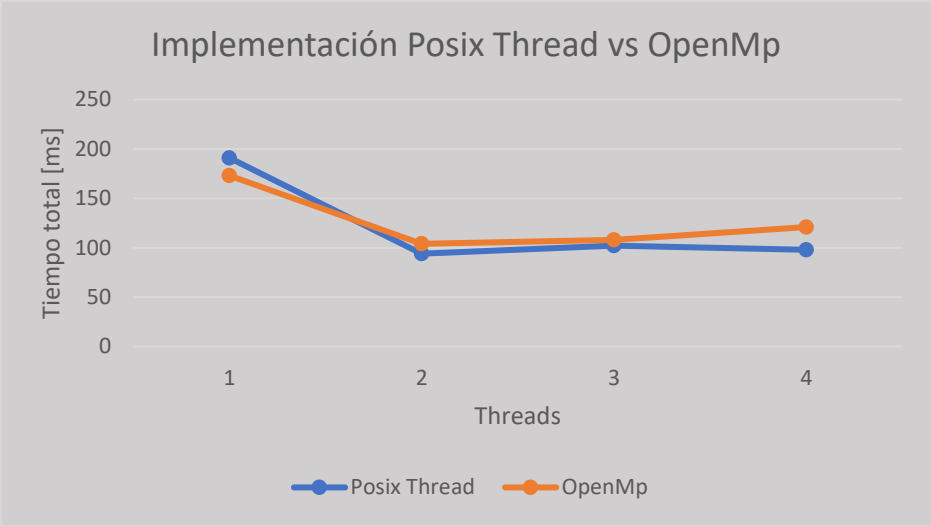

Figura 16.- Gráfico de la implementación con POSIX Thread vs OpenMp.

6 Conclusiones.

Se puede decir, que luego de realizar este taller y el taller 2, se verifico que la programación paralela es mas eficiente que la programación secuencial, esto de acuerdo a los datos obtenidos sobre los tiempos de ejecución.

También se puede decir, que se esperaba que OpenMp fuera mucho más eficiente en tiempo de ejecución que al utilizar POSIX Threads, claramente esto demuestra que no por que una implementación sea más fácil de realizar, esta va a ser más eficiente.

7 Referencias

[1] https://www.openmp.org/about/openmp-faq/#WhatIs.

[2] https://www.openmp.org/about/openmp-faq/#WhatIs.

[3] https://www.roe.ac.uk/ifa/postgrad/pedagogy/2009_kiessling.pdf