Informe Taller 3

Taller de Sistemas Operativos

Escuela de Ingeniería Informática

Abel Núñez Cataldo

Abel.nunez@alumnos.uv.cl

Resumen. *Este informe documenta los procedimientos que se llevaron a cabo sobre el desarrollo de un programa utilizando threads, y el diseño de una solución que permita el llenado de un arreglo y la suma de sus contenidos en forma paralela, usando las herramientas proveidas en clase.*

1. Introducción

Bash es una herramienta de desarrollo de software que trabaja directamente sobre el sistema operativo, agilizando la utilización de memoria y procesamiento de código. Para llevar a cabo este ejercicio se utilizó, el software de virtualización distribuido por Oracle llamado VirtualBox, con el cuál se instalo una máquina virtual de Ubuntu Server versión 18.04, que se basa principalmente en el úso de comandos Linux. En esta entrega se prioriza la implementación de threads (hilos), para lo cual se utilizara principalmente el lenguaje de programación C++ estándar 2014 o superior.

En computación de alto rendimiento, hay herramientas que ayudan a la programación con multi-threads en paralelo, procesamiento en memoria distribuida y plataformas de multiprocesadores de memoria compartida. OpenMP es un programa basado en directivas de aplicación de programas de interfase de API, el cual fue desarrollado específicamente para procesamiento de memoria compartida en paralelo.

El objetivo de este taller, consiste en realizar el diseño e implementación de un programa que llene un arreglo de números enteros y luego los sume. Estas acciones deben ser ambas realizadas de forma paralela, utilizando OpenMP.

Este documento cuenta con una introducción que abarca la tecnología utilizada, los objetivos a alcanzar. Seguida de la descripción del problema planteado en el taller, que incluye la descripción de los datos utilizados, el significado de las variables utilizadas y algunos ejemplos en que se instancian. Finalizando con el diseño de la solución, en la que se muestran diagramas de alto nivel y los métodos utilizados para la solución del problema.

2. Descripción del problema

El prolema planteado, practicamente el mismo del taller anterior, crear un programa que esté compuesto de dós módulos. Uno que llene un arreglo dinámico con números enteros aleatorios en forma paralela, y otro que sume el contenido del arreglo en forma paralela. Se deben realizar prubas de desempeño, que generen datos que permitan visualizar el tiempo de ejecución de ambos módulos, dependiendo del tamaño del problema y de la cantidad de threads utilizados. Se establece que la forma de ejecutar este programa debe ser la siguiente:

```
./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.
```

2.1 Descripcion de los datos

Para esta situación a diferencia del primer taller, no se hara uso de datos almacenados previamente en archivos de texto, sino que se utilizarán datos de tipo entero generados aleatoriamente para llenar un arreglo, al igual que en el segundo taller. Los otros datos que se apreciarán, son los del desempeño que tengan los procesos del programa a la hora de su ejecución.

3. Diseño de la solución

3.1 Metodología

La metodología consiste en analizar los requerimientos, diseñar un modelo del funcionamiento del programa, donde se pueda ver un orden de los procesos a realizar para lograr las acciones de llenado del arreglo y la suma de su contenido. Una vez estudiado el funcionamiento general de la solución , se debe establecer un modelo de diseño para los módulos de la solución, es decir, para cada módulo es necesario realizar un diseño de comportamiento, ya que si bien el procedimiento de bifurcación para los hilos es similiar, estos realizan distintas funciones.

3.2 Diseño

Para el diseño de la solución primero se procede a analizar el problema, estableciendo los siguientes requerimientos:

- Debe recibir como parametros de entrada, el tamaño del arreglo, número de threads y rango de los números en el arreglo.
- Generar un arreglo de tamaño dinámico vacío.
- Llenar el arreglo con números enteros aleatorios en forma paralela.
- Sumar el contenido del arreglo en forma paralela y almacenarlo.

Seguido por el diseño de un diagrama de alto nivel, que ayude a explicar el funcionamiento general de la posible solución para el problema. Para esto se utilizó el mismo diseño de la entrega anterior basado en el ejemplo de llenado paralelo de arreglo proveído por el profesor, en el que se asúme que el tamaño del arreglo es dinámico, configurable por parámetros de entrada. La solución se estructura en las etapas denominadas "*Etapa de llenado*" y "*Etapa de sumado*" (Ver Figura 1), esta última etapa se comporta de forma similar a la lógica de la primera.

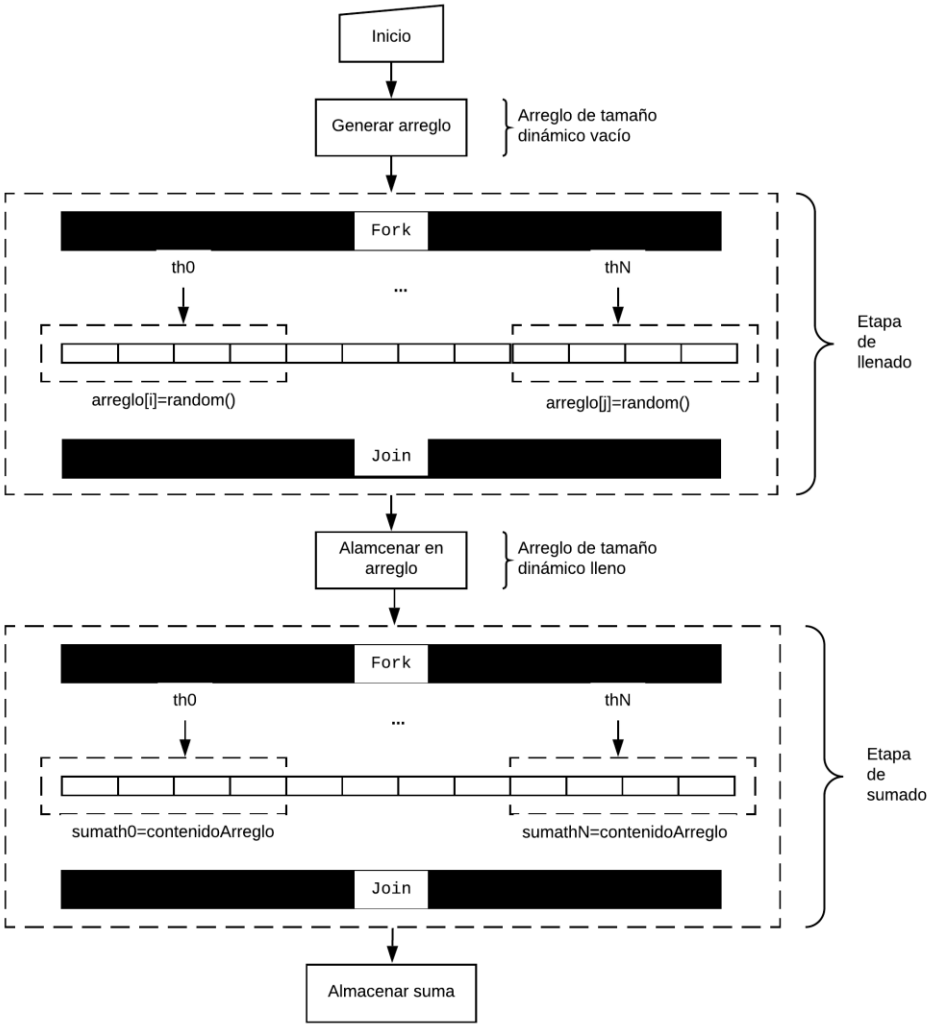

Figura 1 Diseño general de solución.

En la Figura anterior, se puede apreciar el funcionamiento general de la solución, todo este procedimiento se debe completar a través del uso de hilos con OpenMP de forma paralela. Previamente se solicito que a la hora de ejecutar el programa, este reciba como parametros de entrada, el tamaño del arreglo, número de hilos y el rango de los número aleatorios. Una vez ejecutado, se procede a la etapa de llenado del arreglo en el modulo uno, dividiendo el hilo principal en la cantidad de hilos ingresada. En esta estapa, se decidio utilizar una de las funciones del "RNE" (Random Number Engine), proveída por el profesor en el ejemplo de llenado de arreglo. Sólo se utilizó la función **std::uniform_int_distribution** disponible en la librería <random>, reconocida por ser *thread-safe*. Tras terminar el llenado del arreglo, se procede a la etapa de suma con un procedimiento similar, pero con distinta finalidad que es la entrega de la suma total de los datos contenidos en el arreglo. Ambas etapas se realizaron de forma secuencial y paralela, para llevar a cabo un analisis de rendiemiento de los procedimientos de cada una de ellas.

4. Implementación

En este caso no fue necesario la implementación se una función de llenado, con ayuda de OpenMP se puede paralelizar el procedimiento de la implementación secuencial.

4.1 *Etapa de llenado: versión secuencial v/s versión paralela.*

En la Tabla 1 se muestra una comparación de las implementaciónes de la etapa de llenado del arreglo, en cada modalidad, para las cuales en ambas se utilizó la función de generación de números randómicos thread-safe, proveída en los ejemplos, implementada en una sección anterior del código. Notaremos que luego de la implementación secuencial se elimina el contenido del arreglo para realizar el llenado en paralelo, luego éste será el utilizado para realizar la etapa de suma.

Tabla 1

```
//Secuencial
arreglo = new uint64_t[totalElementos];
for(size_t i = 0; i < totalElementos; ++i){
      arreglo[i] = unif(rng);
}
delete[] arreglo;
//Paralelo
arreglo = new uint64_t[totalElementos];
#pragma omp parallel for num_threads(numThreads)
for(size_t i=0; i < totalElementos; ++i){
      arreglo[i] = unif(rng);
}
```

4.2 *Etapa de suma: versión secuencial v/s versión paralela.*

En la Tabla 2, se muestra una comparación, esta vez de las implementaciones de la etapa de sumado, en cada modalidad.

Tabla 2

```
//Secuencial
uint64_t sumaSecuencial=0;
for(size_t i = 0; i < totalElementos; ++i){
      sumaSecuencial += arreglo[i];
}
```

```
//Paralela
uint64_t sumaParalela=0;
#pragma omp parallel for reduction(+:sumaParalela) num_threads(numThreads)
for(size_t i=0; i<totalElementos; ++i){
	sumaParalela += arreglo[i];
}
```

4.3 *Medición de tiempo de ejecución*

Para la medición del tiempo de ejecución en un bloque de código, utilizamos los métodos de **std::chrono**, de la librería <**chrono**>. En la Tabla 3, se puede apreciar un ejemplo para la sección de la etapa de sumado secuencial.

Tabla 3

```
uint64_t sumaSecuencial=0;
start = std::chrono::high_resolution_clock::now();
for(size_t i = 0; i < totalElementos; ++i){
	sumaSecuencial += arreglo[i];
}
end = std::chrono::high_resolution_clock::now();
elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
auto tiempoTotalSuma_S = elapsed.count();
```

5. Resultados – Pruebas de desempeño.

En esta sección, veremos los resultados, realizando pruebas de desempeño para evaluar el comportamiento de la implementación, bajo las dos implementaciónes de cada módulo, con un arreglo de 10000000 de elementos, y un rango de números aleatorios establecido entre 1 y 50. Para cada prueba, se toma el tiempo secuencial y el tiempo con uno y dos threads, y se calcula el índice de desempeño SpeedUp. Para casos con un número mayor de threads, existirán diferencias entre los tiempos de ejecución, pero no son tan apreciables como la diferencia entre uno y dos threads.

5.1 Caso con un solo thread.

```
abel@abelnc:~/prueba/TSSOO-Taller03/src$ ./sumArray -N 100000000 -t 1 -l 1 -L 50
Elementos: 100000000
Threads  : 1
Limite inferior: 1
Limite superior: 50
Suma secuencial total: 2550213745
Suma en paralelo total: 2550213745
------- Tiempos de ejecución --------
Tiempo Llenado Secuencial: 9254[ms]
Tiempo Llenado Paralelo: 9337[ms]
SpeedUp Etapa de Llenado: 0.991111
-------------------------------------
Tiempo Suma Secuencial: 978[ms]
Tiempo Suma Paralela: 780[ms]
SpeedUp Etapa de Suma: 1.25385
-------------------------------------
```

Figura 2 Salida con 100.000.000 de datos con un hilo.

La Figura 2 muestra como se puede apreciar, tanto la suma secuencial y paralela coinciden entre ellas. También notar que, para la etapa de llenado, no existe una diferencia significativa entre los resultados del tiempo secuencial y paralelo, con un SpeedUp de 0.99, lo que significa que la etapa en paralelo es despreciablemente más lenta que la versión secuencial. Sin embargo, en el tiempo de sumado se logra apreciar que el SpeedUp es de 1.25, lo que se traduce en una ganancia de un 25% en terminos de tiempo de ejecución, para esta etapa, siendo que solo se uso un thread.

5.2 Caso con dos threads.

```
abel@abelnc:~/prueba/TSSOO-Taller03/src$ ./sumArray -N 100000000 -t 2 -l 1 -L 50
Elementos: 100000000
Threads  : 2
Limite inferior: 1
Limite superior: 50
Suma secuencial total: 2549866646
Suma en paralelo total: 2549866646
------- Tiempos de ejecución --------
Tiempo Llenado Secuencial: 8751[ms]
Tiempo Llenado Paralelo: 4959[ms]
SpeedUp Etapa de Llenado: 1.76467
-------------------------------------
Tiempo Suma Secuencial: 739[ms]
Tiempo Suma Paralela: 338[ms]
SpeedUp Etapa de Suma: 2.18639
-------------------------------------
```

Figura 3 Salida con 100.000.000 de datos con dos hilos.

La Figura 3 muestra que las sumas aun siguen coincidiendo, pero lo principal se puede notar en los tiempos de ejecución. Existe una diferencia mayormente significativa, con una ganancia en cuanto a tiempo de ejecución, con un SpeedUp de 1.76 en la etapa de llenado y 2.19 en la etapa de sumado, lo que da a entender que las implementacion con dos threads es un 76% y un 119% (respectivamente en cada etapa), más rápida que la versión secuencial.

Tabla 4 Promedio de tiempos de ejecución x cantidad de hilos

# threads	Tiempo de llenado secuencial [ms]	Tiempo de llenado paralelo [ms]	Tiempo de suma secuencial [ms]	Tiempo de suma paralela [ms]
2	8751	4959	739	338
4	8634	5021	736	329
6	8607	4925	748	328
8	8548	5014	734	319
10	8619	4971	735	325

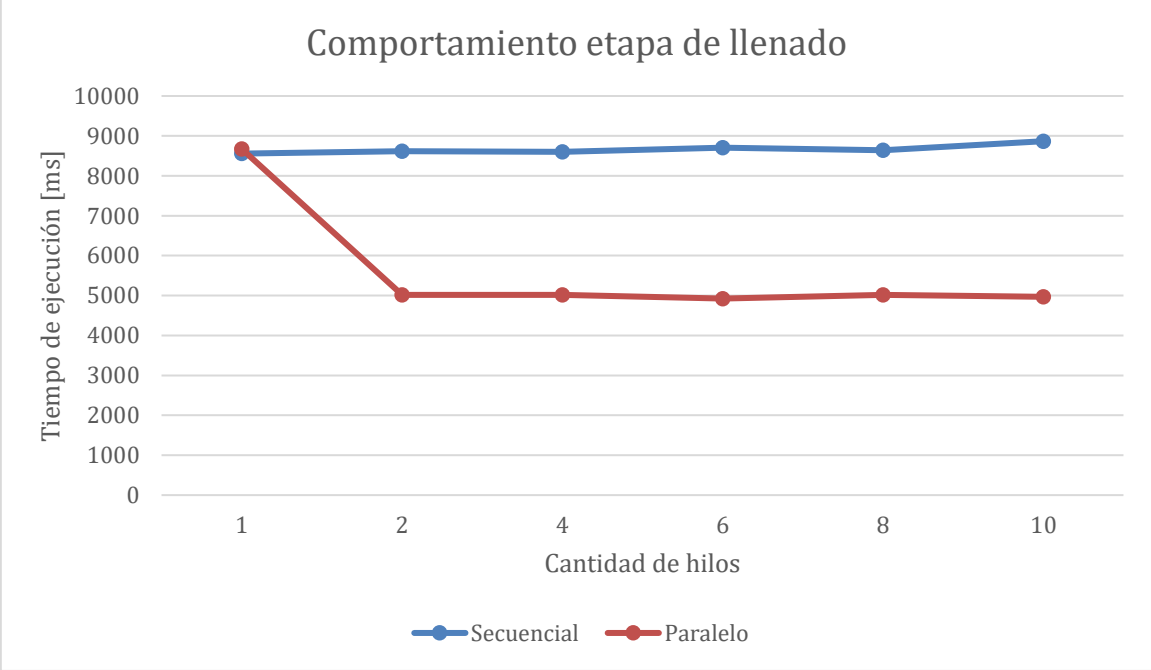

Gráfico 1 Comportamiento de los tiempos de ejecución x hilos (Etapa de llenado).

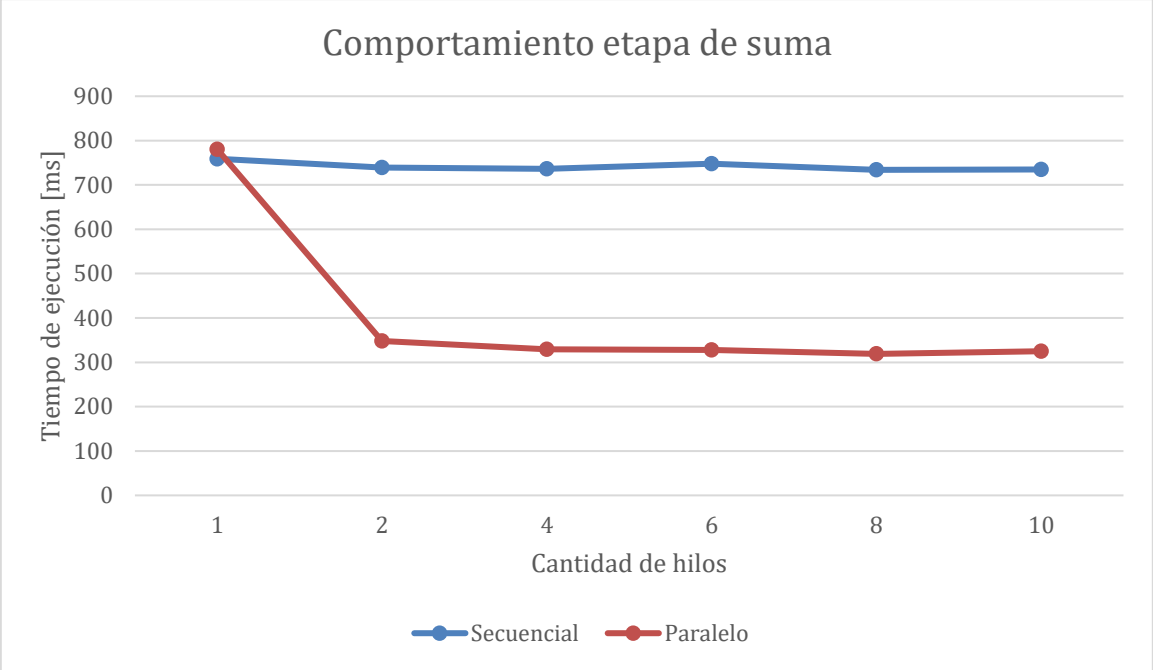

Gráfico 2 Comportamiento de los tiempos de ejecución x hilos (Etapa de suma).

6. Conclusiones

Este documento mostró el desarrollo del taller 3 de la asignatura Taller de Sistemas Operativos, que consistía en el diseño e implementación de una solución de un problema, utilizando threads en forma paralela con OpenMP, con el lenguaje de programación C++. El problema a resolver al igual que en el taller anterior, consiste llenar un arreglo de tamaño dinámico, cantidad de threads y rangos definidos por el usuario. Se realizaron pruebas para el algoritmo implementado, solamente con funciones de generación de números aleatorios thread-safe.