Informe técnico taller 02

Taller de Sistemas Operativos

Profesor: Gabriel Astudillo Muñoz

Alumno: Sebastián González Morales

Escuela de Ingeniería Civil Informática,
Universidad de Valparaíso

18 de junio de 2020

1 Introducción

En este informe se explicará el contexto y el diseño 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 threads POSIX.

POSIX Threads, generalmente conocido como pthreads, es un modelo de ejecución que existe independientemente de un lenguaje, así como un modelo de ejecución paralela. Permite que un programa controle múltiples flujos de trabajo diferentes que se superponen en el tiempo. Cada flujo de trabajo se conoce como un subproceso, y la creación y el control sobre estos flujos se logra haciendo llamadas a la API POSIX Threads. POSIX Threads es una API definida por POSIX.1c estándar, extensiones de Threads (IEEE Std 1003.1c-1995) [1].

El documento esta 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 se presentaran diagramar para que haya un mejor entendimiento lo que se piensa hacer. En la sección cuatro se muestra la metodología utilizada, en la sección cinco se muestran detalles de cómo se comprobó que lo que realizaba el programa concordaba con los parámetros de entrada, en la sección seis se muestran pruebas de desempeño realizadas tanto para la etapa de llenado como para la etapa de suma, estas etapas serán medidas con una biblioteca disponible, 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. 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

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 1). 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 2). En esa figura se utilizaron 2 threads y el arreglo estaba compuestos por números entre 0 y 50.

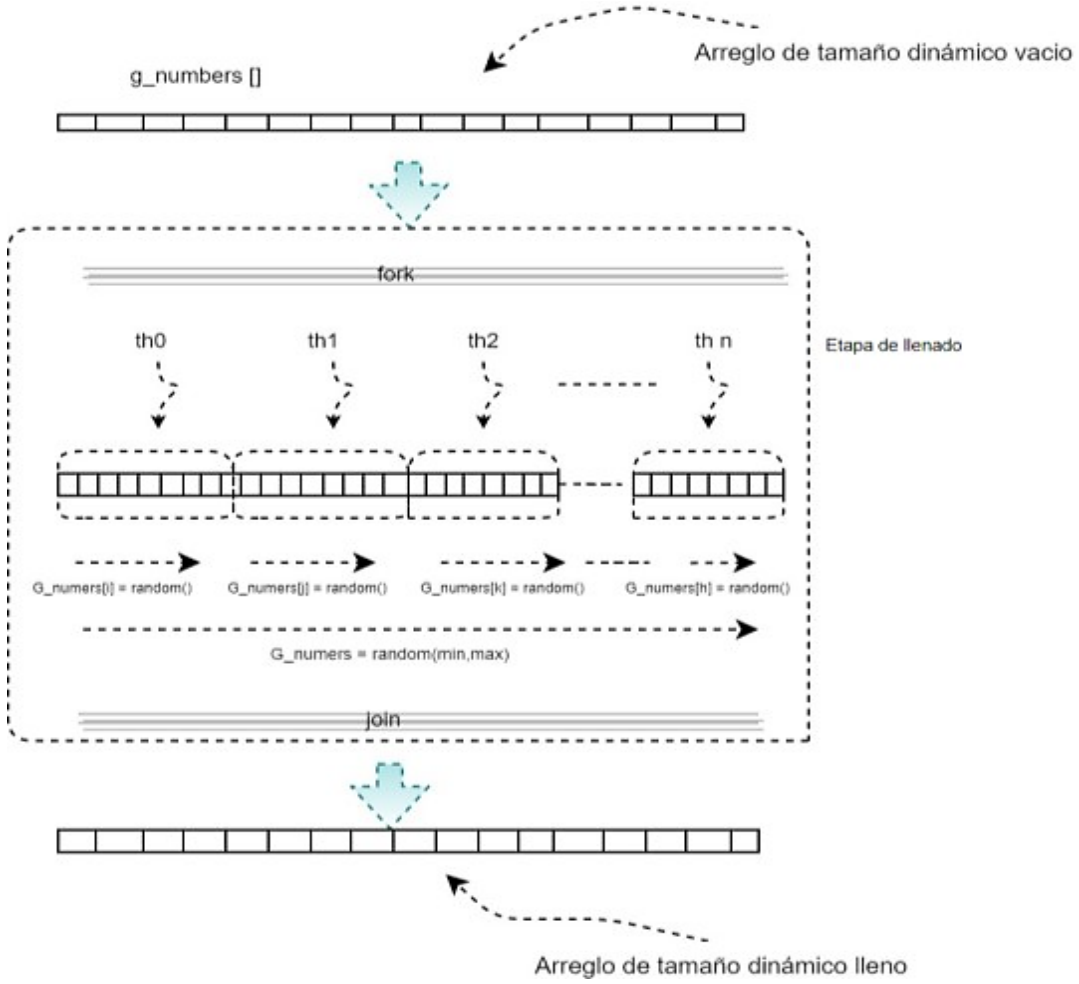

Figura 1

3.2 Etapa de llenado

El llenado del arreglo global **`g_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 limite inferior y el limite superior de los números que se ingresaran al arreglo.

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

3.3 Etapa de suma

Esta etapa se realizará a través de una función llamada **sumasParalelo**, la cual tendrá dos parámetros de entrada, el índices 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.

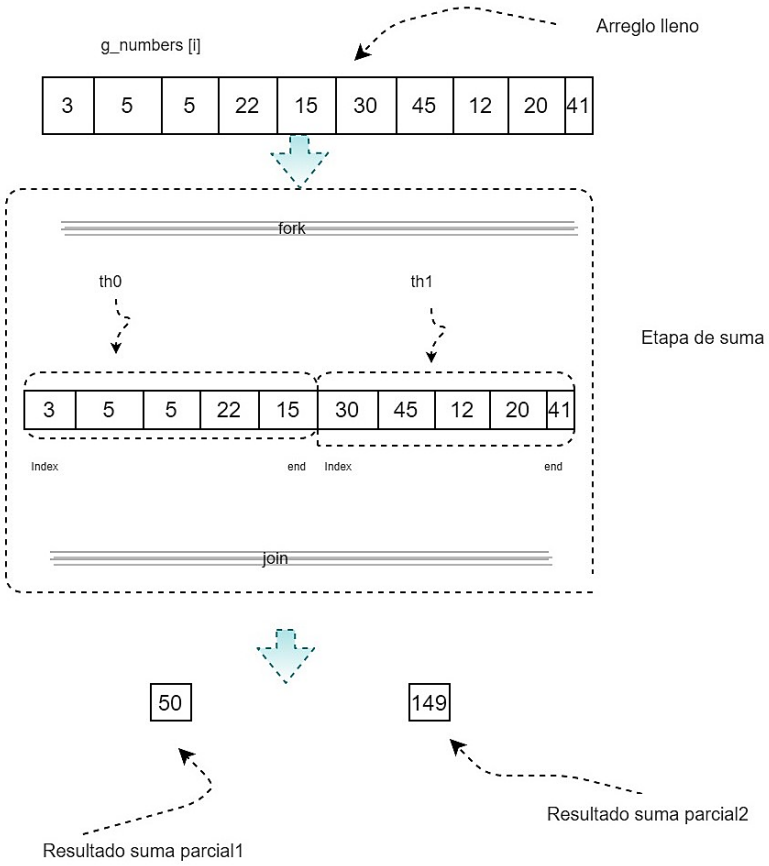

Figura 2

4 Metodología

Lo primero que se hizo fue familiarizarse con la sintaxis de c++, es decir ver los ejemplos entregados por el profesor, y buscar información en internet. Primero se partió armando el código del modulo de llenado, ahí se definió, esta función y se empezaron definir el límite inferior y el limite superior de los números aleatorios. Posteriormente a eso, se buscó la forma de crear n threads y distribuir el trabajo que haría cada uno para el llenado del arreglo, lo primero se solucionó con un vector de threads, y lo segundo se realizo mediante un ciclo donde la variable i se inicia de 0 hasta que i sea menor al total de elementos del arreglo, cabe señalar que el total de elementos del arreglo es ingresado por parámetros, luego se llama a la función fillArray, y se pasan por parámetros el índice de inicio, final y los límites de los números aleatorios.

Para el modulo dos, “sumas del arreglo”, se hizo algo muy parecido, se creó la función sumaParalelo, con dos parámetros de entrada, el índice inicial y el final, que indican que parte del arreglo va a sumar cada thread, esta función es llamada de la principal.

5 Pruebas.

Para realizar esta parte, se modificó el código y se mostró por pantalla lo que realizaba cada módulo del programa, se vio si los resultados eran acorde a lo que se ingresaba por parámetro. En la figura 3, se puede ver un ejemplo de la ejecución, se ingresan diez números al arreglo, se utilizan dos hilos, el limite inferior es doce y el límite superior es cien. De acuerdo a esto, se puede decir que lo mostrado por pantalla es coherente a lo ingresado por parámetros.

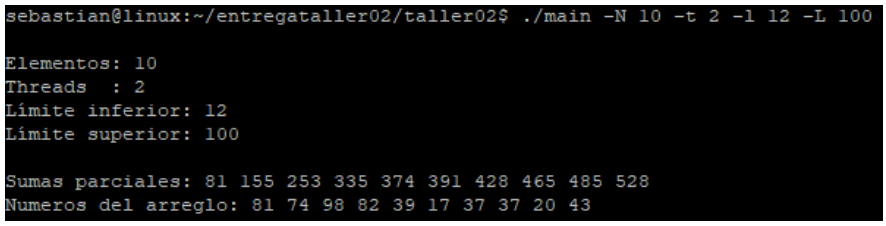

Figura 3

Luego se modifican los parámetros de entrada, y ahora lo que se intenta ver es si los resultados de la suma secuencial y la suma en paralelo arroja los mismos resultados. Como se puede ver en la figura 4, estos resultados concuerdan

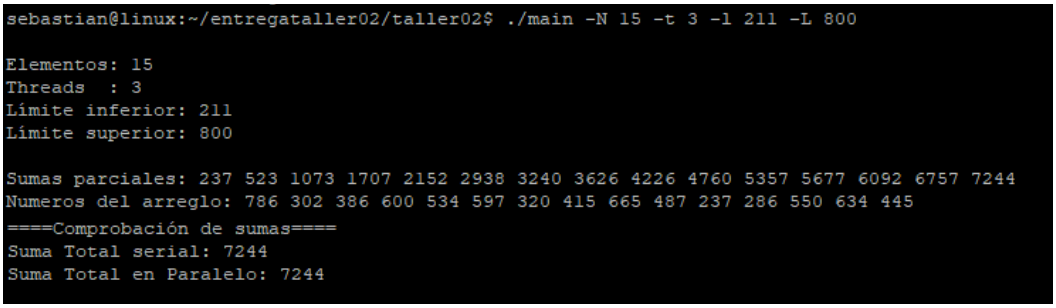

Figura 4

Antes de esto, existía una inconsistencia en lo que mostraba la suma secuencial y lo que mostraba la suma en paralelo, eso se solucionó a través de funciones Posix sincronización, mediante un mecanismo de exclusión mutua (mutex) disponible en **<mutex>:**

Estas funciones son mecanismos de señalización del cumplimiento de condiciones por parte de variables, y mecanismos de acceso de variables que se modifican en forma exclusiva, pero pueden ser leídas en forma compartida. Las funciones para el manejo de zonas de acceso exclusivo tienen el prefijo pthread_mutex

Un mutex es una variable especial que puede tener estado tomado (locked) o libre (unlocked). Es como una compuerta que permite el acceso controlado. Si un hilo tiene el mutex entonces se dice que es el dueño del mutex. Si ningún lilo lo tiene se dice que está libre (o unclucked). Cada mutex tiene una cola de hilos que están esperando para tomar el mutex. El uso de mutex es eficiente, pero debería ser usado sólo cuando su acceso es solicitado por corto tiempo.[2]

En este caso se utilizó lock(), lo que permite solicitar acceso al mutex, el hilo se bloquea hasta su obtención, y unlock lo que permite liberar el mutex.

6 Pruebas de desempeño según cantidad de hilos

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 paralelo con uno, tres y cinco threads, en las etapas de llenado y de suma, también se calcula el índice de desempeño SpeedUp. El tamaño del arreglo es 100000, los números que se generaran aleatoriamente son entre 211 y 800.

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

En la figura 5, se puede apreciar que se utiliza un hilo y ver que los tiempos secuencial y paralelo son muy parecidos, no hay una diferencia notable.

```
sebastian@linux:~/entregataller02/taller02$ ./main -N 1000000 -t 1 -l 211 -L 800

Elementos: 1000000
Threads  : 1
Límite inferior: 211
Límite superior: 800

Sumas parciales:
====Modulo de Llenado del arreglo====
Tiempo secuencial :88[ms]
Tiempo threads    :83[ms]
SpeedUp           :1.06024

====Modulo de Sumas del arreglo====
Tiempo secuencial :3[ms]
Tiempo threads    :3[ms]
SpeedUp           :1

====Comprobación de sumas====
Suma Total serial: 505401297
Suma Total en Paralelo: 505401297
```

Figura 5

En la figura 6, se puede apreciar que se utilizan 3 hilos, y aquí si se puede ver una diferencia mayor, principalmente en la etapa de llenado con un speedUp igual a 4,3, mientras que para la etapa de llenado, no hay una diferencia tan grande, el speedUp es igual a 0,5.

Figura 6

En la figura 7, finalmente se puede apreciar que se utilizan 5 hilos, en la etapa de llenado del arreglo hay un mayor tiempo secuencial respecto a los cinco hilos, speedUp igual a 2,03, mientras que en la etapa de llenado hay un mayor utilizando hilos que la forma secuencial, speedUp igual a 0,02.

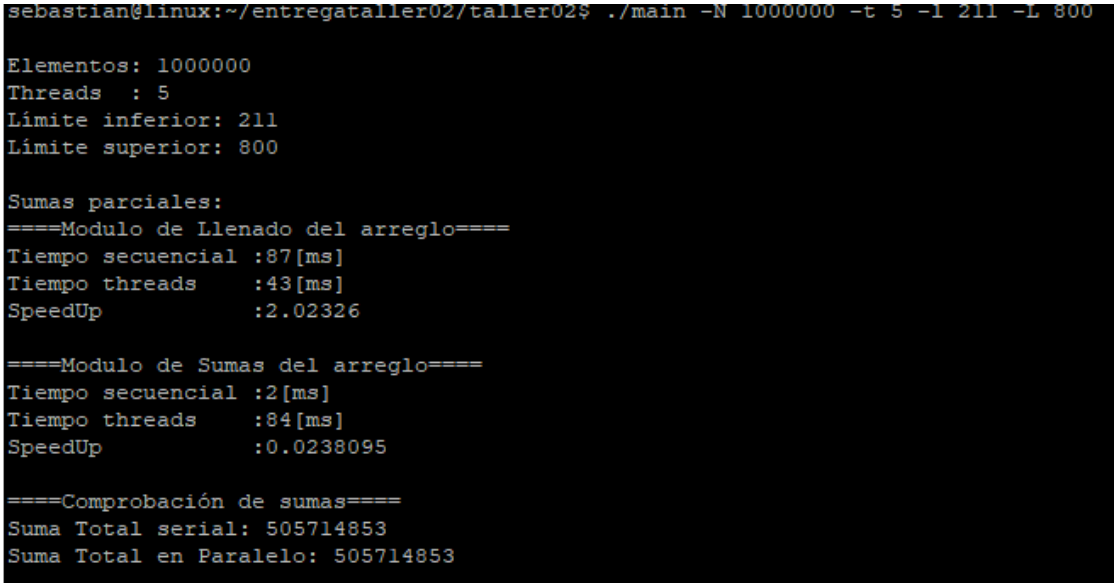

Figura 7

Hay que destacar que en las 3 imágenes, tanto el tiempo la suma secuencial como la suma en paralelo, es la misma, por tanto, hay una consistencia en los datos.

7 Referencias

[1] https://en.wikipedia.org/wiki/POSIX_Threads

[2] http://profesores.elo.utfsm.cl/~agv/elo330/2s06/lectures/POSIX_threads/POSIX_Threads_Synchronization.html