Ejemplo de llenado paralelo de arreglo

Taller de Sistemas Operativos

Escuela de Ingeniería Civil Informática,
Universidad de Valparaíso
2020, Primer Semestre

1 Problema

El problema consiste en llenar un arreglo unidimensional en forma paralela (utilizando threads) y realizar las pruebas de desempeño respectiva. El diseño que se presenta en este documento asume que se utilizarán dos threads y la implementación en C++ versión 2017.

2 Diseño

2.1 Descripción general

El diseño asume que el arreglo es de tamaño dinámico configurable a través de parámetros de entrada al programa y es global al proceso. Esto significa que es visible por todos los threads creados dentro de él. La solución se estructura en una etapa denominada "*Etapa de llenado*" (ver Figura 1). Cada thread conoce los índices de inicio y fin donde debe almacenar un número aleatorio. Para que exista una ganancia en velocidad de ejecución, sólo de deben utilizar funciones de generación de números aleatorios del tipo thread-safe, si no va a existir un cuello de botella y la solución con threads será más costosa en tiempo que la solución secuencial.

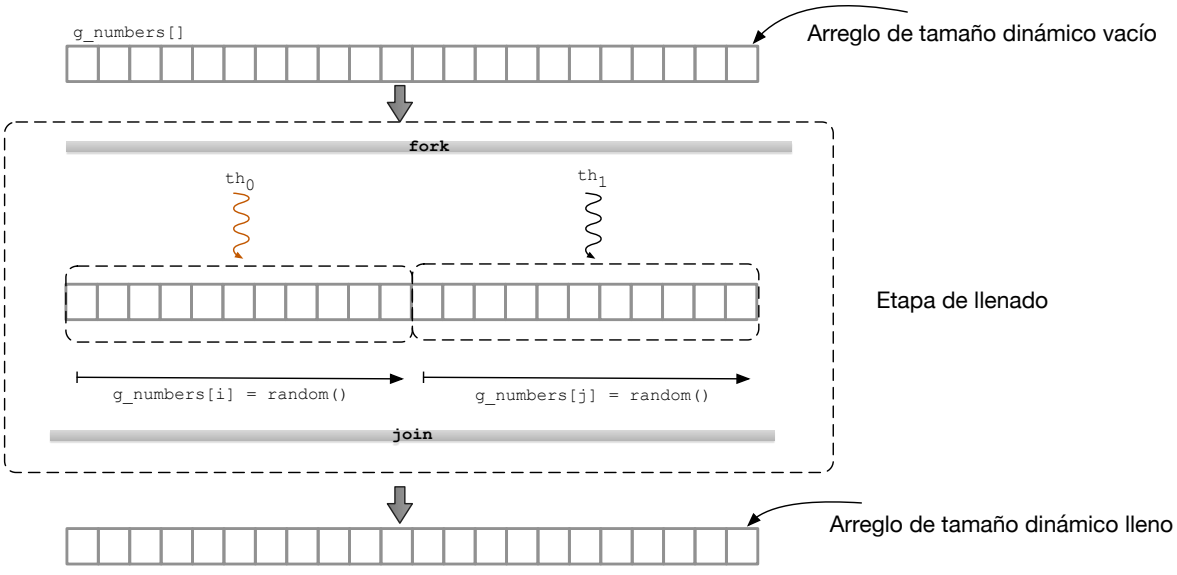

Figura 1

2.2 Etapa de llenado

El llenado del arreglo global **`g_numbers[]`**, se realizará a través de una función denominada **`fillArray`**, que tiene dos parámetros: uno que indica el índice de inicio (**`beginIdx`**) y otro para el índice de fin (**`endIdx`**). Esto permite que la función sea llamada en forma secuencial o parcial por los threads. La Figura 2 muestra el diseño de esta función.

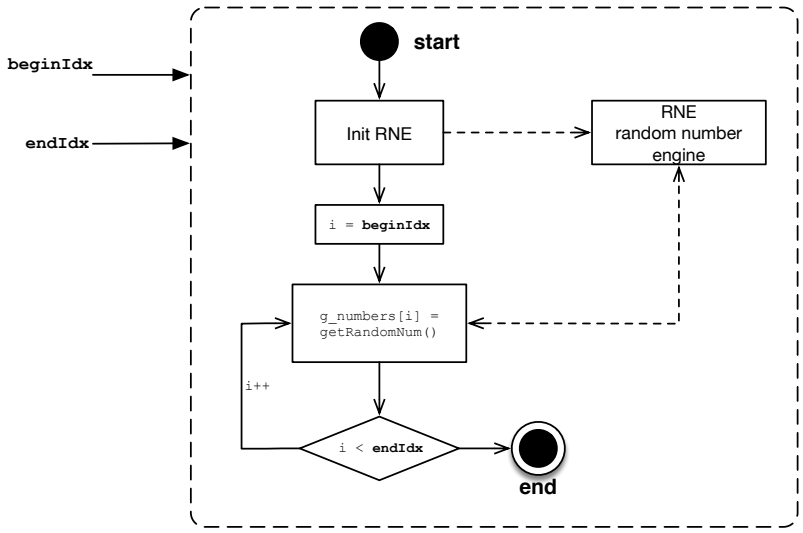

Figura 2

Esta función tendrá un generador de número aleatorios (RNE, Random Number Engine). Para las pruebas de funcionamiento, se probará con la función **`std::rand()`** disponible en la biblioteca **`<cstdlib>`** y con la función **`std::uniform_int_distribution<>`** disponible en **`<random>`**.

2.3 Implementación

2.3.1 Llenado del arreglo

La implementación de la función **`fillArray`** se muestra en la Tabla 1

Tabla 1

```
void fillArray(size_t beginIdx, size_t endIdx, size_t RNEtype) {
      //Used for std::rand
      std::srand(std::time(0)); //current time as seed

      //Used for std::uniform_int_distribution
      std::random_device device;
      std::mt19937 rng(device());
      std::uniform_int_distribution<> unif(0, RAND_MAX);

      for(size_t i = beginIdx; i < endIdx; ++i){
            switch(RNEtype){
                  case 0:
                        g_numbers[i] = std::rand();
                        break;
                  case 1:
                        g_numbers[i] = unif(rng);
                        break;
            }
      }
}
```

2.3.2 Medición del tiempo de ejecución

Para medir el tiempo de ejecución en un bloque de código, se utiliza los métodos de **`std::chrono`**, disponibles en **`<chrono>`**. Un ejemplo se muestra en la Tabla 2.

Tabla 2 Ejemplo de medición de tiempo de ejecución para el llenado secuencial

```
g_numbers = new uint64_t[totalElementos];
auto start = std::chrono::high_resolution_clock::now();
fillArray(0, totalElementos, RNEtype);
auto end  = std::chrono::high_resolution_clock::now();
auto elapsed =
    std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
auto totalTimeFill_serial = elapsed.count();
delete[] g_numbers;
```

3 Pruebas de desempeño RNE

Las pruebas de desempeño se diseñan para evaluar el comportamiento del algoritmo implementado bajo distintas implementaciones del módulo RNE, con un arreglo de 10000000 elementos. Para cada una de ellas, se toma el tiempo secuencial y el tiempo con dos threads y se calcula el índice de desempeño SpeedUp.

3.1 Caso RNE implementado con std::rand()

Figura 3

La Figura 3 muestra una prueba realizada con **`std::rand()`**. Tal como se mencionó en la parte del diseño, hay un costo elevado cuando se utiliza una función que no es thread-safe en multiprogramación. Prueba de esto, es que el SpeeUp es 0.03. Esto significa que el algoritmo con threads es un 97% más lento que su versión secuencial.

3.2 Caso RNE implementado con std::uniform_int_distribution<>

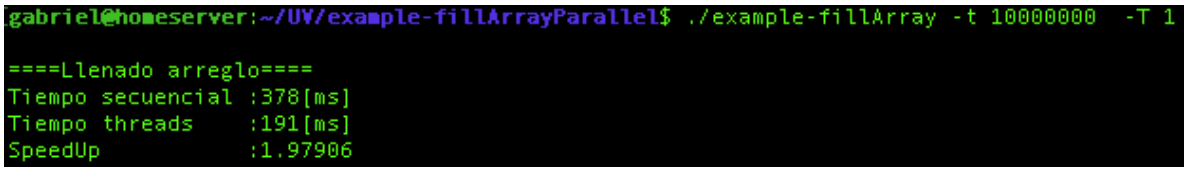

Figura 4

La Figura 4 muestra una prueba realizada con std::uniform_int_distribution<>. Debido a que esta función es thread-safe, los dos threads pueden acceder simultaneamente a ella. Esto implica que hay una ganancia en términos de tiempo de ejecución, con un SpeedUp de 1.98, lo que significa que la implementación con threads es un 98% más rápida que se contraparte secuencial.

4 Validación del algoritmo paralelo

Para comprobar que efectivamente la solución con threads está utilizando en forma paralela el computador, es posible utilizar una herramienta que establezca la afinidad del proceso. La afinidad de un proceso determina en qué core(s) se va a ejecutar. Para esto, hay que investigar cuál es el hardware que se dispone. En el caso de este ejemplo, el esquema del sistema utilizado se muestra en la Figura 5. Es un sistema con dos cores. Cada core tiene habilitado

hyper-threading, la que permite ejecutar dos threads de ejecución por cada core, pero en forma concurrente/paralela.

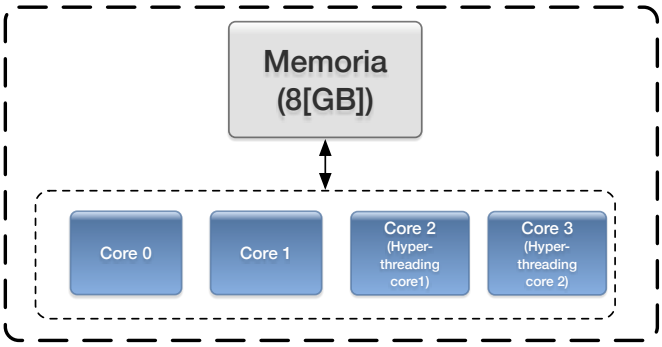

Figura 5

La herramienta que se utilizará es taskset, la que permite establecer en que core(s) se ejecutará el proceso.

4.1 Caso 1: ejecutar en un solo core

```
gabriel@homeserver:~/UV/example-fillArrayParallel$ taskset -c 0 ./example-fillArray -t 10000000  -T 1

====Llenado arreglo====
Tiempo secuencial :377[ms]
Tiempo threads    :371[ms]
SpeedUp           :1.01617
```

Figura 6

En la Figura 6 se muestra la ejecución del programa, confinado a un solo core. Como es de esperar, no hay diferencia entre los tiempos de ejecución secuencial y con threads, ya que no se puede utilizar más un core del sistema (SpeedUp= 1.0).

4.2 Caso 2: ejecutar en dos cores

```
gabriel@homeserver:~/UV/example-fillArrayParallel$ taskset -c 0,1 ./example-fillArray -t 10000000  -T 1

====Llenado arreglo====
Tiempo secuencial :377[ms]
Tiempo threads    :186[ms]
SpeedUp           :2.02688
```

Figura 7

En la Figura 7 se muestra la ejecución del mismo programa, pero ahora utilizando dos cores. Debido a que el SpeedUp es 2.0, se puede concluir que efectivamente hay ejecución paralela del algoritmo con threads.

4.3 Caso 3: ejecutar en dos solo cores, pero uno con hyper-threading

```
gabriel@homeserver:~/UV/example-fillArrayParallel$ taskset -c 0,2 ./example-fillArray -t 10000000  -T 1

====Llenado arreglo====
Tiempo secuencial :379[ms]
Tiempo threads    :257[ms]
SpeedUp           :1.47471
```

Figura 8

En la Figura 8 se muestra la ejecución del programa utilizando el core 0, pero con su core con hyper-threading. Si bien se observa que hay una mejora de rendimiento, con un SpeedUp de 1.47, esto significa que no hubo una ejecución paralela, sino que concurrente/paralela.

5 Conclusiones

Este documento mostró una forma básica de diseñar, implementar, verificar y validar un algoritmo paralela. El proble a resolver es llenar un arreglo de tamaño dinámico a través de dos threads. Se realizaron diversas pruebas para el algoritmo implementado, como funcionamiento con funciones de generación de números aleatorios thread-safe y no thread-safe y pruebas que validaron el comportamiento paralelo, restringiendo el uso de los cores del sistema a través de una herramiento del sistema operativo.

Queda pendiente realizar pruebas estadísticas para validar los resultados obtenidos.