Universidad del Rosario

# PROYECTO DE ALGORITMOS Y ESTRUCTURAS DE DATOS

# ***MULTITHREADING***

***~ENTREGA FINAL~***

**Integrantes:**
-David Felipe Martinez Castiblanco
-Miguel Ángel Castillo Espitia

Fecha de entrega: 18 de noviembre del 2018

Universidad del Rosario

## RESUMEN EJECUTIVO:

El problema principal del proyecto fue mejorar el rendimiento de algunos algoritmos, ya que, detectamos que en estos algoritmos cuando el input es demasiado grande, el tiempo de ejecución se ralentiza y hace que incluso un algoritmo con baja complejidad computacional llegue a parecer ineficiente.

La solución que encontramos a este problema fue el uso de programación paralela, ejecución simultánea de multi-hilos(multi-threads); para esto debemos mirar cómo inicializar un hilo, es decir cuando se crea, cuando está en ejecución, si entra en espera, cuando acaba, etc. y así mismo en el caso de C++ familiarizarse con los métodos que incluye la estructura de datos <thread> en la librería estándar y cómo interactúa con la memoria Heap, Stack y demás.

Luego de estar familiarizado con los conceptos procedimos a implementar una pequeña demostración sobre el tiempo de ejecución de distintos algoritmos con diferente complejidad computacional.

## FUNCIONALIDAD DE LA HERRAMIENTA:

- Ejecuta un algoritmo que suma dos vectores, pasando por referencia el primero y sumando entrada a entrada con el segundo.
- Ejecuta un algoritmo que calcula el producto punto y lo suma a una variable global denominada “producto punto”.
- Ejecuta un algoritmo que calcula el n-ésimo número armónico, sumando desde derecha a izquierda y viceversa.
- Ejecuta un algoritmo que calcula la cantidad de números primos desde 1 hasta n.
- Ejecuta un algoritmo de la sucesión de fibonacci hasta el n.
- Permite definir el números de hilos que se usarán para correr estas operaciones.
- Obtiene el tiempo de ejecución de cada algoritmo.

## DESCRIPCIÓN DE LA HERRAMIENTA:

Primero hicimos un código secuencial, es decir, que use un sólo hilo, ejecutando algoritmos O(N), O($N^2$) y recursivos, tales como la suma de dos vectores, el producto punto entre dos vectores, el cálculo de los números primos hasta un número n, la sucesión de fibonacci, etc.
Calculamos el tiempo de ejecución y lo mostramos en la consola. Luego, usamos la estructura de datos <thread> de la librería estándar de C++ para crear los mismos algoritmos (suma de vectores, producto punto, etc.), pero ahora usando

multi-threads. Adicionalmente implementamos la clase <future> de C++, para ejecutar el código del cálculo del n-ésimo no. armónico, ya que con esta clase podemos obtener lo que retorna esa función.

Cada algoritmo es medido por aparte, esto se hizo para sacar una medición más precisa a la hora de comparar 1, 2, 4 y 8 threads principalmente. Además se mide el tiempo de ejecución de todos los algoritmos a la vez con mismo objetivo de la comparación.

La división de tareas de cada algoritmo es proporcional al número de threads usados, es decir, por ejemplo al sumar dos vectores, suponiendo que usaremos 4 threads, la suma se dividirá en cuatro sumas parciales, dividiendo el vector en 4 partes y así que cada thread se ocupe de cada suma.

Generalmente los algoritmos recursivos hacen el llamado a la misma función por lo tanto si queremos usar multithread para estos algoritmos lo mejor es no incluirlos dentro de la función, sino primeramente dividir correctamente el problema y que se corra pocas veces el algoritmo recursivo.de otra forma dentro de la recursión la llamada a crear nuevos hilos ocasiona mucho gasto de memoria y tiempo de ejecución lo que concluye en una implementación extremadamente ineficiente.

## CONCLUSIONES:

Luego de haber revisado varias veces los tiempos de ejecución, con distintos algoritmos, en distintas arquitecturas de computadores y demás nos dimos cuenta que la implementación de threads no debe ser realizada a la ligera ya que en un principio uno intuitivamente podría llegar a pensar que el tiempo de ejecución es inversamente proporcional a la cantidad de threads, es decir, mientras más threads menos tiempo de ejecución, pero esto no es del todo así pues se deben tener en cuenta varios factores como el número de threads que usaremos vs el número que es conveniente usar en cada computador, e incluso mirar casos en los que estos acceden al mismo espacio en memoria (shared memory).

En cuanto tiempo de ejecución claramente se reduce, por ejemplo en un computador que recomienda usar 8 threads obtuvimos los siguientes resultados:

**Algoritmo que suma de vectores de tamaño 1 x 10^8:**

-Usando 1 hilo : 0.937 seg.

-Usando 2 hilos: 0.503 seg.

-Usando 4 hilos: 0.285 seg.

-Usando 8 hilos: 0.201 seg.

**********

**Miguel Castillo y David Martinez**

**Algoritmo que hace el producto punto entre dos vectores de tamaño**

**1 x 10^8:**

-Usando 1 hilo: 28.885 seg.

-Usando 2 hilos: 16.633 seg.

-Usando 4 hilos: 11.156 seg.

-Usando 8 hilos: 10.425 seg.

**Algoritmo calcula la cantidad de números primos desde 1 hasta 2 x 10^5:**

-Usando 1 hilo : 76.537 seg.

-Usando 2 hilos: 58.345 seg.

-Usando 4 hilos: 33.655 seg.

-Usando 8 hilos: 18.642 seg.

Notamos también que al usar la clase de C++ <future> incurrimos en programación asincrónica, es decir procesos que corren en segundo plano pero a diferencia de threads estos van hacia una cola con los demás Async para ser ejecutados en orden y retornar un valor determinado, por esta razón y debido a nuestra implementación, con cualquier número de threads el tiempo de ejecución de Async será aproximadamente el mismo:

**Algoritmo que calcula el n-ésimo número armónico. Con n = 5x10^8:**

-Usando 1 hilo : 6.379 seg.

-Usando 2 hilos: 6.413 seg.

-Usando 4 hilos: 6.477 seg.

-Usando 8 hilos: 6.45 seg.

Por último el objetivo propuesto fue completado correctamente con distintas complejidades computacionales, a futuro pensamos indagar mucho más sobre esta librería de c++ para mejorar su implementación y aprovechar al máximo el uso de memoria y procesamiento paralelo.

**********

**Miguel Castillo y David Martinez**