# Informe práctica 3

# Paralelización función SAXPY iterativa

Autores:

Felipe Cadavid

Jaime Alexis Herrera Ruiz

Marfa Alejandra Kleber Sierra

## Introducción

El código proporcionado implementa una operación SAXPY **(Single-precision A\*X Plus Y)** de forma iterativa utilizando múltiples hilos en paralelo. La operación SAXPY es comúnmente utilizada en computación numérica y se define como ***Yi =a∗Xi +Yi*** *,* donde **a** es un escalar y **Xi** y **Yi** son vectores de tamaño **p**. La versión presentada aquí busca optimizar el rendimiento utilizando múltiples hilos para realizar los cálculos de manera concurrente.

El código comienza analizando los argumentos de línea de comandos para determinar el tamaño del vector, la semilla aleatoria, el número de hilos a crear y el número máximo de iteraciones. Luego, inicializa los vectores *X* e *Y*con valores aleatorios, así como el escalar **a**. Posteriormente, crea múltiples hilos para ejecutar la función *saxpy\_thread*, que realiza la operación SAXPY en paralelo en partes del vector ***Y***. Finalmente, calcula los promedios de **Y** después de cada iteración y muestra los resultados, incluido el tiempo de ejecución y los últimos valores de ***Y*** y los promedios de ***Y****.*

El código hace uso de la biblioteca estándar de C para manejar hilos (*pthread.h*) para crear y controlar múltiples hilos de ejecución. Además, utiliza la función *getopt* para analizar los argumentos de línea de comandos de forma eficiente. El tiempo de ejecución se mide utilizando las funciones *gettimeofday* para calcular el tiempo transcurrido entre el inicio y el final de la ejecución del programa.

## Método de Paralelización

Cambios globales

Para mejorar el rendimiento usando múltiples hilos, se realizaron varias modificaciones al código original de forma global que se verán reflejadas a lo largo de todas las iteraciones con ciertas variaciones:

* Inclusión de *pthreads*: Se añadió la biblioteca *pthread.h* para manejar los hilos.
* Se definió una estructura *thread\_data\_t* para pasar datos a cada hilo. Esta estructura incluye:
  + Identificador del hilo (*thread\_id)*.
  + Número total de hilos (*n\_threads*).
  + Tamaño del vector (p).
  + Número máximo de iteraciones (*max\_iters*).
  + El escalar a.
  + Punteros a los vectores *X, Y* y *Y\_avgs*.
* Función SAXPY para hilos: Se creó una función *saxpy\_thread* que realiza la operación SAXPY en una porción de los vectores *X* y *Y* correspondiente al hilo. Cada hilo procesa una parte específica del vector, calculada en función de su identificador y el número total de hilos.
* Creación y gestión de hilos: En la función principal, se inicializaron los hilos y se pasó la estructura *thread\_data\_t* a cada uno.
* Se crearon y esperaron los hilos usando *pthread\_create* y *pthread\_join*.
* Normalización de *Y\_avgs:* Después de que todos los hilos completaran su ejecución, se normalizaron los valores de *Y\_avgs* dividiendo cada suma acumulada por p.

Sin embargo, se crearon diferentes versiones donde buscamos intentar mejorar la concurrencia.

A continuación, se explican a grandes rasgos los cambios que se realizaron en las diferentes versiones:

|  |  |
| --- | --- |
| **Versión** | **Cambios Implementados** |
| saxpyv2 | Se incluyó la división de trabajo por hilos de manera dinámica.  Se siguen utilizando funciones de la biblioteca *pthread*. |
| saxpyv3 | Además de los mencionados anteriormente se incluyó lo siguiente:   * Implementación de *Mutex*: Se añadió un *mutex* (*pthread\_mutex\_t mutex*) para asegurar actualizaciones seguras al vector *Y\_avgs* entre hilos y para probar si esto mejora el rendimiento * Actualización Segura: Se usa un *mutex* para asegurar que las actualizaciones al vector *Y\_avgs* sean consistentes entre los hilos, esto porque en la versión 2 del código, los resultados de los promedios no estaban siendo consistentes con la prueba de escritorio. * Partición del Array: Se decidió que cada hilo operara en una porción específica de los vectores de *X* y *Y*. |
| saxpyv4 | En esta iteración se realizó lo siguiente:   * División del Vector y cálculo de los promedios: En esta versión, estos cálculos se hacen antes y después de ejecutar los hilos. Esto para evitar los *locks* frecuentes y costosos para el rendimiento que fueron causados por el *mutex* * Estructura *thread\_data\_t:* Se añadió un puntero *double*\* *Y\_part\_sum* a la estructura *thread\_data\_t* para almacenar las sumas parciales de Y para cada hilo e iteración. * Actualización de Sumas Parciales: En lugar de actualizar directamente el vector Y\_avgs, se actualiza la suma parcial de *Y\_part\_sum* por cada hilo respectivamente. * Cálculo Final en el Hilo Principal: Después de que todos los hilos terminan, la función principal se calcula el vector Y\_avgs final sumando las sumas parciales de todos los hilos y haciendo el promedio basado en el tamaño de los vectores y la cantidad de iteraciones |
| saxpyv5 | * Cada hilo calcula la suma parcial de Y la almacena en Y\_part\_sum, un vector para contener las sumas parciales de todos los hilos y todas las iteraciones. * Al hacer que cada hilo opere en su porción de Y\_part\_sum, no hay necesidad de sincronización durante la operación SAXPY. * Después de que todos los hilos terminan, el hilo principal suma las sumas parciales de todos los hilos y calcula los promedios dividiéndolas por p para calcular el Y\_avgs final. * Se intento implementar *posix\_memalign* durante la creación de los vectores de cada estructura, pero al final fue retirado del código debido a errores de segmentación |

## Resultados

**Condiciones de experimentación:** A continuación, se muestran las especificaciones técnicas del dispositivo que se utilizó para realizar las ejecuciones:

Architecture: x86\_64

CPU op-mode(s): 32-bit, 64-bit

Address sizes: 48 bits physical, 48 bits virtual

Byte Order: Little Endian

CPU(s): 16

On-line CPU(s) list: 0-15

Vendor ID: AuthenticAMD

Model name: AMD Ryzen 7 5700X 8-Core Processor

CPU family: 25

Model: 33

Thread(s) per core: 2

Core(s) per socket: 8

Socket(s): 1

Stepping: 2

BogoMIPS: 6787.26

Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr\_opt pdpe1gb r dtscp lm constant\_tsc rep\_good nopl tsc\_reliable nonstop\_tsc cpuid extd\_apicid pni pclmulqdq ssse3 fma cx16 sse4\_1 sse4\_2 movbe popcnt ae s xsave avx f16c rdrand hypervisor lahf\_lm cmp\_legacy cr8\_legacy abm sse4a misalignsse 3dnowprefetch osvw topoext ibrs ibpb stibp vmmcall fsgsbase bmi1 avx2 smep bmi2 erms rdseed adx smap clflushopt clwb sha\_ni xsaveopt xsavec xgetbv1 xsaves clzero xsaveerptr arat umip vaes vpclmulqdq rdpid fsrm

Virtualization features:

Hypervisor vendor: Microsoft

Virtualization type: full

Caches (sum of all):

L1d: 256 KiB (8 instances)

L1i: 256 KiB (8 instances)

L2: 4 MiB (8 instances)

L3: 32 MiB (1 instance)

Vulnerabilities:

Gather data sampling: Not affected

Itlb multihit: Not affected

L1tf: Not affected

Mds: Not affected

Meltdown: Not affected

Mmio stale data: Not affected

Retbleed: Not affected

Spec rstack overflow: Mitigation; safe RET, no microcode

Spec store bypass: Vulnerable

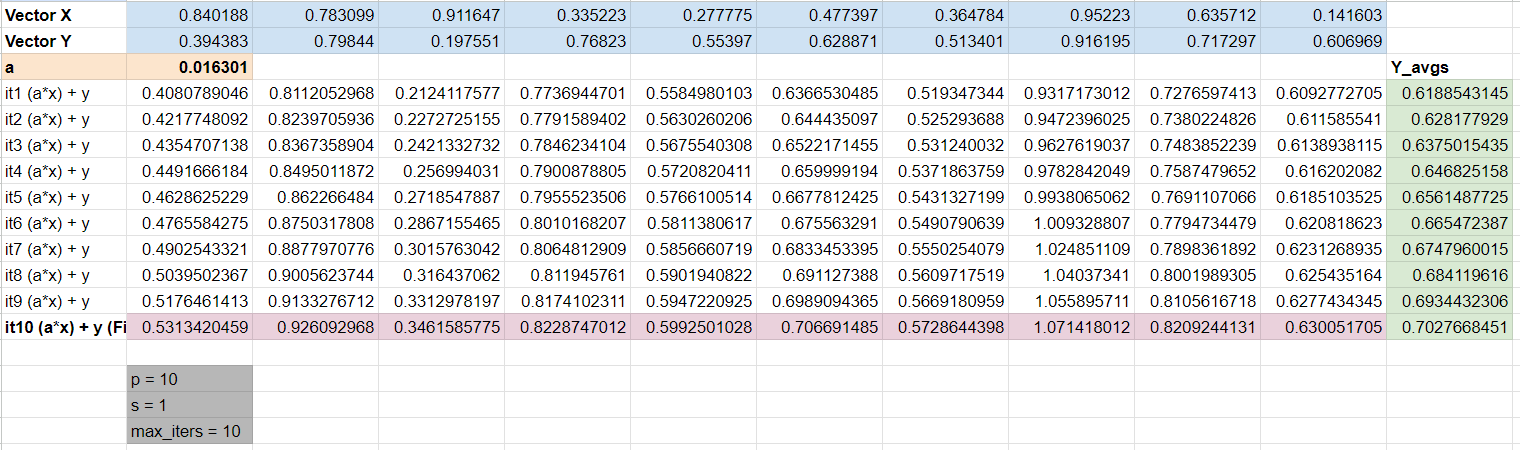
Spectre v1: Mitigation; usercopy/swapgs barriers and \_\_user pointer sanitization

Spectre v2: Mitigation; Retpolines, IBPB conditional, IBRS\_FW, STIBP conditional, RSB filling, PBRSB-eIBRS Not affected

Srbds: Not affected

Tsx async abort: Not affected

**Validación de las versiones**: Para poder validar correctamente las versiones fue necesario establecer una prueba de escritorio que nos permitiera identificar que los cálculos se estaban realizando correctamente tras cada iteración. Para esto se creó la siguiente tabla, cuya fuente original fueron los resultados de la ejecución secuencial del código proporcionado por la pauta del laboratorio:

*Ilustración 1 Prueba de escritorio*

Esta prueba de escritorio consiste en calcular cada iteración de *Y* suponiendo que los vectores son de 10 elementos y las iteraciones son 10.

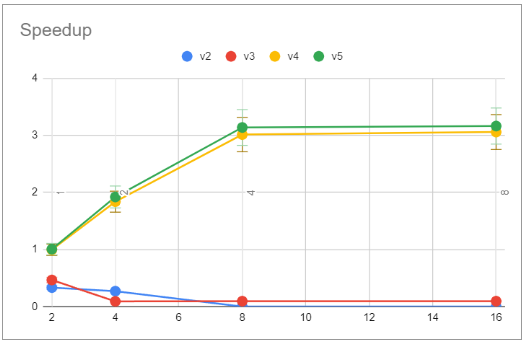
**Hallazgos de la experimentación por versión:**

|  |  |
| --- | --- |
| **Versión** | **Hallazgos** |
| saxpyv2 | * Extremadamente lento * *Segmentation Fault* error a partir de los 8 hilos * Los datos no fueron consistentes cuando se realizó la prueba de escritorio |
| saxpyv3 | * La implementación de mutex demostró ser muy ineficiente * Los tiempos de ejecución empeoraron dramáticamente * Se concluyo que usar bloqueos (*locks*) y mutex no son la solución para me * Los tiempos de ejecución empeoraron con cada incremento de hilos |
| saxpyv4 | * Dividir el vector y calcular los promedios por fuera de los hilos mejoraron drásticamente los tiempos de ejecución * Hacer los cálculos del vector *Y\_avgs* al final también mejoro drásticamente el rendimiento * Dada las condiciones del dispositivo utilizado, no hubo mejoras significativas entre los 8 y los 16 hilos de ejecución |
| saxpyv5 | * Partir el cálculo de *Y* para que fuera de formar parcial por hilo no tuvo un impacto significativo en el mejoramiento del rendimiento. * Se intento implementar *posix\_memalign* durante la creación de los vectores de cada estructura, pero al final fue retirado del código debido a errores de segmentación * Dada las condiciones del dispositivo utilizado, no hubo mejoras significativas entre los 8 y los 16 hilos de ejecución excepto que durante la ejecución con 16 hilos se bloqueó el dispositivo en algunas ocasiones |

Tras realizar las pruebas de cada versión, fue evidente que el *speedup* a partir de la versión 4 incremento drásticamente. Posteriormente para la versión 5 aunque hubo cierta mejora, el cambio no fue significativo e incluso causaba problemas con la operación del dispositivo.

Adicionalmente, en todas las versiones fue evidente un aumento en el tiempo de ejecución en un solo hilo en comparación con la ejecución secuencial del código proporcionada por la pauta del laboratorio.

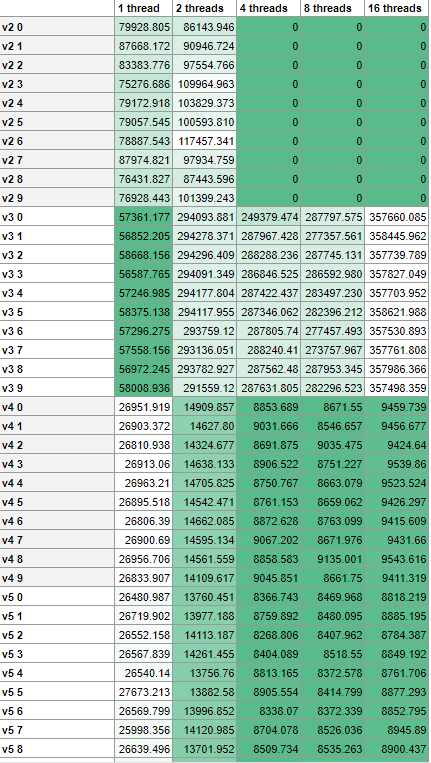
A continuación, se muestra la gráfica del *speedup* con cada una de las versiones que se ejecutaron donde se evidencia como a partir de la versión 4 mejora el rendimiento. Para calcular el *speedup* se tomó como base el tiempo de ejecución secuencial del código proporcionado inicialmente en la pauta del laboratorio. Los cálculos pueden también ser consultados en este link: [SAXPY NEW Results](https://docs.google.com/spreadsheets/d/10J7eSvNG48xHWUcsrZexXAKFikoeyYWydVloOJN0tJU/edit#gid=0)



*Ilustración 2 Resultado del Speedup por cada versión*

## Resultados con Optimización

A continuación, se muestra una tabla donde se realizó el registro de las ejecuciones por cada iteración. Esta tabla pretende demostrar como cambiaron los tiempos de ejecución en cada versión.



*Ilustración 3 Tiempos de elección de cada versión de código*

En La tabla anterior se muestran los tiempos de ejecución de cada versión de código, se visualizan 10 resultados por cada ejecución tal como lo pide el reto, además se resaltan los valores con mejor tiempo a excepción de los valores que se encuentran en cero, ya que se presentó error de segmentación a partir de la ejecución de 4 hilos.

## Conclusiones

* La implementación de hilos siempre genera una sobrecarga que solo es justificable cuando se requiere la ejecución de procesos de forma concurrente, se llega a esta conclusión tras identificar que el código base proporcionado por la pauta del laboratorio es más rápido que su equivalente de ejecución con un solo hilo
* Cada versión del código *SAXPY* incluye mejoras progresivas para abordar el problema de performance en la ejecución de código, por ello, el reto nos llevó a realizar varias versiones como en la versión 2 que carece de consistencia de datos debido a la ausencia del mecanismo de sincronización.
* En la versión 3 se introdujo mutex para garantizar la consistencia de datos, lo que mejoró potencialmente, pero introdujo sobrecarga de sincronización aumentando drásticamente el tiempo de ejecución lo que nos llevó a realizar otra versión y mejorar el performance.
* En la versión 4, se reduce la sobrecarga del mutex pre-calculando la división y los promedios del vector, pero se agrega el uso de memoria para sumas parciales lo que nos llevó a mejorar solo un poco en la versión 5, utilizando sumas parciales locales a cada hilo, lo que hizo que mejorara el rendimiento, aunque hizo que se utilizara memoria adicional para *Y\_part\_sum*.
* Para cumplir con el reto del profesor de mejorar el performance, es utilizar la versión 5 por su menor sobrecarga de rendimiento.

## Referencias bibliográficas

* <https://man7.org/linux/man-pages/man3/posix_memalign.3.html>
* <https://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf>
* <https://ptgmedia.pearsoncmg.com/images/9780201633924/samplepages/0201633922.pdf>
* <https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf>
* <https://www.ibm.com/docs/en/zos/3.1.0?topic=functions-posix-memalign-reserve-aligned-storage-block>
* <https://ferestrepoca.github.io/paradigmas-de-programacion/paralela/paralela_teoria/index.html#four>
* <https://learn.microsoft.com/es-es/dotnet/csharp/language-reference/statements/lock>
* <https://pages.cs.wisc.edu/~remzi/OSTEP/threads-sema.pdf>
* <https://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf>
* <https://www.um.es/earlyadopters/actividades/a3/PCD_Activity3_Session1.pdf>
* <https://www.ibm.com/docs/es/aix/7.3?topic=programming-using-mutexes>