Link to Repo:
https://github.com/pipecadav/
SO-Lab3-20241/

Informe práctica 3

Paralelización función SAXPY iterativa

Autores:
Felipe Cadavid
Jaime Alexis Herrera Ruiz
Marfa Alejandra Kleber Sierra

1. 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 $Y_i = a * X_i + Y_i$, donde **a** es un escalar y $\mathbf{X_i}$ y $\mathbf{Y_i}$ 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.

2. Método de Paralelización

Cambios globales

Link to Repo:
https://github.com/pipecadav/
SO-Lab3-20241/

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:

Link to Repo:
https://github.com/pipecadav/
SO-Lab3-20241/

	• 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

3. 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
```

Link to Repo:
https://github.com/pipecadav/
SO-Lab3-20241/

```
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:

Vector X	0.840188	0.783099	0.911647	0.335223	0.277775	0.477397	0.364784	0.95223	0.635712	0.141603	
Vector Y	0.394383	0.79844	0.197551	0.76823	0.55397	0.628871	0.513401	0.916195	0.717297	0.606969	
a	**0.016301**										**Y_avgs**
it1 (a*x) + y	0.4080789046	0.8112052968	0.2124117577	0.7736944701	0.5584980103	0.6366530485	0.519347344	0.9317173012	0.7276597413	0.6092772705	0.6188543145
it2 (a*x) + y	0.4217748092	0.8239705936	0.2272725155	0.7791589402	0.5630260206	0.644435097	0.525293688	0.9472396025	0.7380224826	0.611585541	0.628177929
it3 (a*x) + y	0.4354707138	0.8367358904	0.2421332732	0.7846234104	0.5675540308	0.6522171455	0.531240032	0.9627619037	0.7483852239	0.6138938115	0.6375015435
it4 (a*x) + y	0.4491666184	0.8495011872	0.256994031	0.7900878805	0.5720820411	0.659999194	0.5371863759	0.9782842049	0.7587479652	0.616202082	0.646825158
it5 (a*x) + y	0.4628625229	0.862266484	0.2718547887	0.7955523506	0.5766100514	0.6677812425	0.5431327199	0.9938065062	0.7691107066	0.6185103525	0.6561487725
it6 (a*x) + y	0.4765584275	0.8750317808	0.2867155465	0.8010168207	0.5811380617	0.675563291	0.5490790639	1.009328807	0.7794734479	0.620818623	0.665472387
it7 (a*x) + y	0.4902543321	0.8877970776	0.3015763042	0.8064812909	0.5856660719	0.6833453395	0.5550254079	1.024851109	0.7898361892	0.6231268935	0.6747960015
it8 (a*x) + y	0.5039502367	0.9005623744	0.316437062	0.811945761	0.5901940822	0.691127388	0.5609717519	1.04037341	0.8001989305	0.625435164	0.684119616
it9 (a*x) + y	0.5176461413	0.9133276712	0.3312978197	0.8174102311	0.5947220925	0.6989094365	0.5669180959	1.055895711	0.8105616718	0.6277434345	0.6934432306
it10 (a*x) + y (Fi	0.5313420459	0.926092968	0.3461585775	0.8228747012	0.5992501028	0.706691485	0.5728644398	1.071418012	0.8209244131	0.630051705	0.7027668451
	p = 10										
	s = 1										
	max_iters = 10										

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.

Link to Repo:
https://github.com/pipecadav/
SO-Lab3-20241/

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

Link to Repo:
https://github.com/pipecadav/
SO-Lab3-20241/

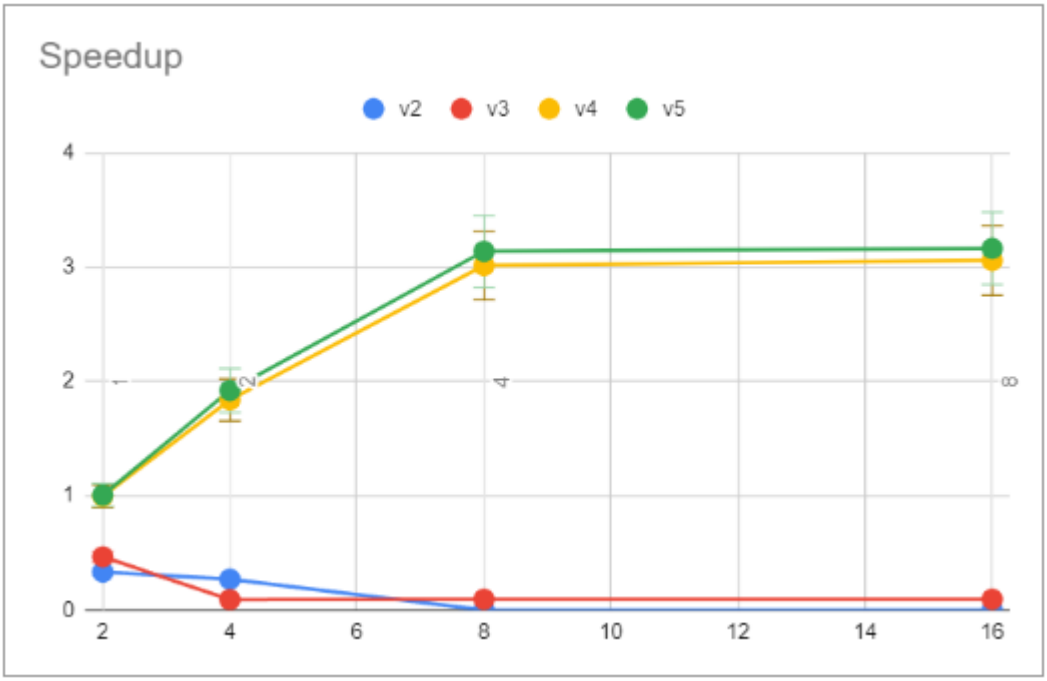

Ilustración 2 Resultado del Speedup por cada versión

4. 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.

Link to Repo:
https://github.com/pipecadav/
SO-Lab3-20241/

	1 thread	2 threads	4 threads	8 threads	16 threads
v2 0	79928.805	86143.946	0	0	0
v2 1	87668.172	90946.724	0	0	0
v2 2	83383.776	97554.766	0	0	0
v2 3	75276.686	109964.963	0	0	0
v2 4	79172.918	103829.373	0	0	0
v2 5	79057.545	100593.810	0	0	0
v2 6	78887.543	117457.341	0	0	0
v2 7	87974.821	97934.759	0	0	0
v2 8	76431.827	87443.596	0	0	0
v2 9	76928.443	101399.243	0	0	0
v3 0	57361.177	294093.881	249379.474	287797.575	357660.085
v3 1	56852.205	294278.371	287967.428	277357.561	358445.962
v3 2	58868.156	294296.409	288288.236	287745.131	357739.789
v3 3	56587.765	294091.349	286846.525	286592.980	357827.049
v3 4	57246.985	294177.804	287422.437	283497.230	357703.952
v3 5	58375.138	294117.955	287346.062	282396.212	358621.988
v3 6	57296.275	293759.12	287805.74	277457.493	357530.893
v3 7	57558.156	293136.051	288240.41	273757.967	357761.808
v3 8	56972.245	293782.927	287562.48	287953.345	357986.366
v3 9	58008.936	291559.12	287631.805	282296.523	357498.359
v4 0	26951.919	14909.857	8853.689	8671.55	9459.739
v4 1	26903.372	14627.80	9031.666	8546.657	9456.677
v4 2	26810.938	14324.677	8691.875	9035.475	9424.64
v4 3	26913.06	14638.133	8906.522	8751.227	9539.86
v4 4	26963.21	14705.825	8750.767	8663.079	9523.524
v4 5	26895.518	14542.471	8761.153	8659.062	9426.297
v4 6	26806.39	14662.085	8872.628	8763.099	9415.609
v4 7	26900.69	14595.134	9067.202	8671.976	9431.66
v4 8	26956.706	14561.559	8858.583	9135.001	9543.616
v4 9	26833.907	14109.617	9045.851	8661.75	9411.319
v5 0	26480.987	13760.451	8366.743	8469.968	8818.219
v5 1	26719.902	13977.188	8759.892	8480.095	8885.195
v5 2	26552.158	14113.187	8268.806	8407.962	8784.387
v5 3	26567.839	14261.455	8404.089	8518.55	8849.192
v5 4	26540.14	13756.76	8813.165	8372.578	8761.706
v5 5	27673.213	13882.58	8905.554	8414.799	8877.293
v5 6	26569.799	13996.852	8338.07	8372.339	8852.795
v5 7	25998.356	14120.985	8704.078	8526.036	8945.89
v5 8	26639.496	13701.952	8509.734	8535.263	8900.437

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.

5. 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.

Link to Repo:
https://github.com/pipecadav/
SO-Lab3-20241/

- 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.

6. 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