ARQUITECTURA DE ORDENADORES

Práctica 4: Explotar el potencial de las arquitecturas modernas

Grupo 1313, Pareja 04:

Florentino García Aznar
Alejandro Monterrubio Navarro

Ejercicio 0:

La información sobre la arquitectura de la máquina utilizada para esta práctica se obtuvo a través de los comandos cat /proc/cpuinfo y sudo dmidecode en un sistema Linux. A continuación, se detallan los aspectos clave relacionados con la topología del sistema:

1. Procesador:

- Modelo: AMD Ryzen 9 5900X 12-Core Processor.
- Frecuencia Base: 2200 MHz (2.2 GHz).
- Frecuencia Máxima: 4950 MHz (4.95 GHz), como se indica en dmidecode.
- Frecuencia Actual: 3700 MHz, lo que sugiere un estado operativo con un aumento en la frecuencia respecto a la base.

2. Hyperthreading:

- El sistema tiene hyperthreading activado, como se deduce de la diferencia entre el número de cpu cores (12 núcleos físicos) y siblings (24 hilos de procesamiento). Esto indica que cada núcleo físico puede manejar dos hilos, duplicando virtualmente el número de núcleos disponibles para procesamiento.

3. Arquitectura y Memoria:

- Placa Base: MAG X570S TOMAHAWK MAX WIFI (MS-7D54) de Micro-Star International Co., Ltd.
- Memoria RAM: La configuración de memoria es DDR4 con una velocidad de 2133 MT/s, aunque esta velocidad puede ser superior dependiendo de la configuración específica del sistema.
- Capacidad Máxima de Memoria: 128 GB.

La forma en la que se imprime la información con los comandos es esta:

> cat proc/cpuinfo

Esto nos proporciona la siguiente información:

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 25
model           : 33
model name      : AMD Ryzen 9 5900X 12-Core Processor
stepping        : 2
microcode       : 0xa201204
cpu MHz                 : 2200.000
cache size      : 512 KB
physical id     : 0
siblings : 24
core id         : 0
cpu cores       : 12
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 16
wp              : yes
```

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 rdtscp lm constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid aperfmperf rapl pni pclmulqdq monitor ssse3 fma cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx f16c rdrand lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs skinit wdt tce topoext perfctr_core perfctr_nb bpext perfctr_llc mwaitx cpb cat_l3 cdp_l3 hw_pstate ssbd mba ibrs ibpb stibp vmmcall fsgsbase bmi1 avx2 smep bmi2 erms invpcid cqm rdt_a rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local clzero irperf xsaveerptr rdpru wbnoinvd arat npt lbrv svm_lock nrip_save tsc_scale vmcb_clean flushbyasid decodeassists pausefilter pfthreshold avic v_vmsave_vmload vgif v_spec_ctrl umip pku ospke vaes vpclmulqdq rdpid overflow_recov succor smca fsrm
bugs : sysret_ss_attrs spectre_v1 spectre_v2 spec_store_bypass srso
bogomips : 7400.11
TLB size : 2560 4K pages
clflush size : 64
cache_alignment : 64
address sizes : 48 bits physical, 48 bits virtual
power management: ts ttp tm hwpstate cpb eff_freq_ro [13] [14]

…

processor : 23
vendor_id : AuthenticAMD
cpu family : 25

...

> sudo dmidecode

dmidecode 3.3
Getting SMBIOS data from sysfs.
SMBIOS 2.8 present.
60 structures occupying 2544 bytes.
Table at 0xCDA03000.

Handle 0x0000, DMI type 0, 26 bytes
BIOS Information
	Vendor: American Megatrends International, LLC.
	Version: 1.00
	Release Date: 07/06/2021
	Address: 0xF0000
	Runtime Size: 64 kB
	ROM Size: 32 MB
	Characteristics:
		PCI is supported
		BIOS is upgradeable
		BIOS shadowing is allowed
		Boot from CD is supported
		Selectable boot is supported
		BIOS ROM is socketed
		EDD is supported
		Japanese floppy for NEC 9800 1.2 MB is supported (int 13h)
		Japanese floppy for Toshiba 1.2 MB is supported (int 13h)
		5.25"/360 kB floppy services are supported (int 13h)
		5.25"/1.2 MB floppy services are supported (int 13h)
		3.5"/720 kB floppy services are supported (int 13h)
		3.5"/2.88 MB floppy services are supported (int 13h)
		Print screen service is supported (int 5h)
		8042 keyboard services are supported (int 9h)
		Serial services are supported (int 14h)
		Printer services are supported (int 17h)
		CGA/mono video services are supported (int 10h)
		ACPI is supported

```
                USB legacy is supported
                BIOS boot specification is supported
                Targeted content distribution is supported
                UEFI is supported
        BIOS Revision: 5.17

Handle 0x0001, DMI type 1, 27 bytes
System Information
        Manufacturer: Micro-Star International Co., Ltd.
        Product Name: MS-7D54
        Version: 1.0
        Serial Number: To be filled by O.E.M.
        UUID: 93bf4632-a028-b912-ae54-d8bbc18d7d12
        Wake-up Type: Power Switch
        SKU Number: To be filled by O.E.M.
        Family: To be filled by O.E.M.

Handle 0x0002, DMI type 2, 15 bytes
Base Board Information

…

Handle 0x003A, DMI type 143, 16 bytes
OEM-specific Type
        Header and Data:
                8F 10 3A 00 00 01 02 03 04 05 06 07 08 09 0A 0B
        Strings:
                GOP1-[N/A]
                RAID-14.8.0.1042
                PXEU1-v1.1.0.4
                AMI-AptioV
                GSES-5/1
                MsiFlash-18
                RavenPI-RAVEN_PI_VERSION
                N/A
                N/A
                N/A
                N/A
                N/A

Handle 0x003B, DMI type 127, 4 bytes
End Of Table
```

Ejercicio 1:

1.1 ¿Se pueden lanzar más threads que cores tenga el sistema? ¿Tiene sentido hacerlo?

Respuesta: Sí, se pueden lanzar más threads que cores en el sistema. En algunos casos, tiene sentido hacerlo, especialmente si los threads realizan operaciones de E/S o están esperando recursos, lo que puede llevar a un bloqueo. En tales situaciones, tener más threads que cores puede ayudar a utilizar mejor la CPU durante estos tiempos de espera. Sin embargo, para cálculos intensivos en CPU, lanzar más threads que cores puede llevar a una sobrecarga debido al cambio de contexto y a la competencia por los recursos del CPU.

1.2 ¿Cuántos threads debería utilizar?

Equipo Propio: El procesador tiene 12 cores físicos y 24 hilos (debido al hyperthreading). Idealmente, lo ideal sería probar con 12 threads (cores físicos) y posiblemente también con 24 threads (hilos lógicos) para ver cómo afecta el rendimiento.

1.3 Modifique el programa omp1.c para utilizar las tres formas de elegir el número de threads y deduzca la prioridad entre ellas.

Modificaciones Realizadas:
El programa omp1.c fue modificado para explorar las distintas formas de determinar el número de threads en un entorno OpenMP. Las modificaciones incluyeron:

El uso de la función omp_set_num_threads(int num_threads) para establecer programáticamente el número de threads.
La inclusión de la cláusula num_threads(numthr) en la directiva #pragma omp parallel para definir el número de threads en el bloque de código paralelo.
La configuración de la variable de entorno OMP_NUM_THREADS antes de ejecutar el programa para establecer el número de threads a nivel de entorno.
Resultados y Prioridad:
Los experimentos realizados arrojaron los siguientes resultados en cuanto a la prioridad de estas configuraciones:

Máxima Prioridad - Cláusula num_threads en #pragma omp parallel: La configuración que se establece mediante esta cláusula tiene la prioridad más alta. Independientemente de los valores proporcionados por omp_set_num_threads o OMP_NUM_THREADS, el número de threads utilizado por el programa corresponde al valor establecido en num_threads.

Prioridad Media - Función omp_set_num_threads: Cuando no se especifica la cláusula num_threads, el número de threads se rige por el valor establecido mediante omp_set_num_threads. Este valor sobrescribe la configuración de la variable de entorno OMP_NUM_THREADS.

Menor Prioridad - Variable de Entorno OMP_NUM_THREADS: La variable de entorno OMP_NUM_THREADS solo determina el número de threads cuando no hay ninguna configuración explícita en el código (ni a través de omp_set_num_threads ni mediante num_threads en #pragma omp parallel).

1.4 ¿Cómo se comporta OpenMP cuando declaramos una variable privada?

Cuando se declara una variable como privada en OpenMP (private), cada hilo recibe su propia copia de la variable. Esta copia es independiente de las copias en otros hilos y no conserva el valor original que la variable tenía antes del bloque paralelo. Además, las modificaciones hechas a una variable privada por un hilo no afectan las copias en otros hilos ni la variable original fuera del bloque paralelo. Esto se puede ver en el comportamiento de la variable a, donde cada hilo empieza con su propia copia de a, la cual tiene un valor no inicializado o 0, y las modificaciones en un hilo no afectan a otros hilos.

1.5 ¿Qué ocurre con el valor de una variable privada al comenzar a ejecutarse la región paralela?

Cuando una variable se declara como privada (private) en OpenMP y comienza a ejecutarse la región paralela, cada hilo crea su propia copia local de esa variable. Lo importante a destacar es que el valor inicial de la variable privada no está definido al comienzo de la región paralela. Esto significa que la variable privada no hereda automáticamente el valor que tenía antes de entrar en el bloque paralelo.

La variable a, que fue declarada como privada, tiene valores no inicializados o cero al comienzo de la región paralela en los diferentes hilos. Esto demuestra que las variables privadas no conservan o no tienen un valor inicial definido, lo que puede llevar a valores indeterminados hasta que se asignen explícitamente dentro de la región paralela.

Es decir, al comienzo de una región paralela en OpenMP, el valor de una variable declarada como privada es indeterminado y no refleja necesariamente el valor que tenía antes de entrar en el bloque paralelo.

1.6 ¿Qué ocurre con el valor de una variable privada al finalizar la región paralela?

Al finalizar la región paralela en OpenMP, las variables privadas de cada hilo se descartan y no tienen ningún impacto en las variables fuera de la región paralela. Es decir, cualquier modificación hecha a una variable privada dentro de la región paralela queda confinada a esa región y al hilo específico que la realizó. Las variables privadas son esencialmente locales a la región paralela y su ciclo de vida se limita a esta.

La variable a, declarada privada. Aunque a fue modificada dentro de la región paralela por cada hilo, estas modificaciones no afectaron el valor de a fuera de la región paralela. Esto se evidencia en el valor final de a después de la región paralela, que permanece igual al valor inicial antes de entrar en el bloque paralelo.

El valor de una variable privada al finalizar la región paralela en OpenMP es descartado, y cualquier modificación hecha a esa variable dentro de la región paralela no afecta su estado fuera de dicha región.

1.7 ¿Ocurre lo mismo con las variables públicas?

En OpenMP, una variable declarada como pública (o compartida usando la cláusula shared) es accesible por todos los hilos dentro de una región paralela. A diferencia de las variables privadas, las cuales son locales a cada hilo y se descartan al finalizar la región paralela, las variables públicas mantienen cualquier modificación realizada por los hilos durante la ejecución paralela. Estas modificaciones son visibles y persistentes incluso después de que termina la región paralela.

La variable b fue declarada como pública. Esto significa que todos los hilos trabajan con la misma instancia de b, y las modificaciones realizadas por cualquier hilo a b son visibles para todos los demás hilos y también después de que la región paralela termina. Esto se evidencia en la salida del programa, donde el valor final de b refleja las modificaciones acumulativas hechas por todos los hilos.

Al finalizar la región paralela en OpenMP, las variables públicas (compartidas) conservan las modificaciones realizadas por los hilos durante la ejecución paralela. Estas modificaciones son globales y afectan el estado de la variable fuera del bloque paralelo.

Ejercicio 2:

2.1 Ejecute la versión serie y entienda cual debe ser el resultado para diferentes tamaños de vector.

El resultado del producto escalar es constante (1000000.000000) independientemente del tamaño del vector. Esto sugiere que los vectores están siendo inicializados de tal manera que el resultado del producto escalar es siempre el mismo.

2.2 Ejecute el código paralelizado con el pragma openmp y conteste en la memoria a las siguientes preguntas:

• ¿Es correcto el resultado?

No, los resultados obtenidos de la versión paralelizada no son correctos. En la versión serie del programa de producto escalar, observamos que el resultado era constante (1000000.000000) independientemente del tamaño del vector. Sin embargo, en la versión paralelizada, los resultados varían con cada tamaño de vector y no coinciden con los de la versión serie. Esto indica claramente que hay un error en la versión paralelizada del programa.

• ¿Qué puede estar pasando?

La inconsistencia en los resultados sugiere que hay problemas en la implementación paralela del programa, probablemente debido a condiciones de carrera. Algunas posibles causas de este problema son:

Condición de Carrera: Cuando se trabaja con variables compartidas en un entorno paralelo, es fundamental asegurarse de que las operaciones en estas variables sean seguras para los hilos. En el caso del producto escalar, si varios hilos intentan actualizar la misma variable acumuladora simultáneamente sin la debida sincronización, esto puede llevar a resultados incorrectos e inconsistentes.

Uso Incorrecto de la Reducción en OpenMP: Una solución común para evitar condiciones de carrera en cálculos como el producto escalar es usar la cláusula reduction en OpenMP. Si la cláusula reduction no se utiliza correctamente, o si se elige una operación de reducción incorrecta, esto podría resultar en una acumulación inadecuada de los valores calculados por los diferentes hilos.

Inicialización de Variables: Es importante también asegurarse de que todas las variables estén inicializadas adecuadamente antes de entrar en la región paralela y que las variables compartidas y privadas se gestionen correctamente.

2.3 Modifique el código anterior y denomine el programa pescalar_par2. Esta versión deber dar el resultado correcto utilizando donde corresponda alguno de los siguientes pragmas #pragma omp critical #pragma omp atomic

• ¿Puede resolverse con ambas directivas? Indique las modificaciones realizadas en cada caso.

1. Uso de #pragma omp critical:

Modificaciones Realizadas:

Se agregó #pragma omp critical alrededor de la actualización de la variable sum en el bucle for.

Código modificado:

```
#pragma omp parallel for
for (int k = 0; k < M; k++) {
   #pragma omp critical
   sum += A[k] * B[k];
}
```

Resultados:
El programa no proporcionó el resultado correcto. El resultado fue 1000.00000, lo que no coincide con la versión serie.

2. Uso de #pragma omp atomic:

Modificaciones Realizadas:

Se utilizó #pragma omp atomic para la actualización de la variable sum.

Código modificado:

```
#pragma omp parallel for
for (int k = 0; k < M; k++) {
   #pragma omp atomic
   sum += A[k] * B[k];
}
```

Resultados:
El programa proporcionó el resultado correcto (1000000.000000) para todos los tamaños de vector probados, lo cual es consistente con la versión serie.

• ¿Cuál es la opción elegida y por qué?

La opción elegida es #pragma omp atomic.

Razón:
Aunque #pragma omp atomic resultó en un tiempo de ejecución más largo en comparación con #pragma omp critical, produjo el resultado correcto y consistente con la versión serie del programa.
La precisión y la corrección de los resultados son fundamentales, especialmente en cálculos numéricos como el producto escalar.
#pragma omp atomic es más adecuada para operaciones simples de actualización de variables, como es el caso en este programa.

2.4 Modifique el código anterior y denomine el programa resultante pescalar_par3. Esta versión deber dar el resultado correcto utilizando donde corresponda alguno de los siguientes pragmas #pragma omp parallel for reduction

Resultados con #pragma omp parallel for reduction:

Los resultados son consistentes con la versión serie del programa, con un resultado de 1000000.000000 para todos los tamaños de vector probados.
Los tiempos de ejecución son relativamente bajos y consistentes, indicando una buena eficiencia en el cálculo paralelo.

Comparación con el Punto Anterior y Elección de la Mejor Opción:

Comparando estos resultados con los obtenidos anteriormente utilizando #pragma omp atomic, se observan las siguientes diferencias:

Con #pragma omp atomic: Aunque se obtuvieron resultados correctos, el tiempo de ejecución fue notablemente más alto en comparación con el uso de reduction.

Con #pragma omp parallel for reduction: Se obtienen resultados correctos y consistentes con tiempos de ejecución más bajos.

¿Cuál será la opción elegida y por qué?

Opción Elegida:
La opción elegida es #pragma omp parallel for reduction.

Razón:
La directiva reduction en OpenMP está específicamente diseñada para este tipo de operaciones donde se necesita acumular un resultado a partir de cálculos realizados en paralelo.
Proporciona una manera eficiente y segura de manejar la suma acumulativa de los productos escalar, evitando condiciones de carrera y optimizando el rendimiento.
Los resultados obtenidos indican que esta versión no solo produce el resultado correcto, sino que también lo hace con una eficiencia de tiempo superior a la versión con atomic.

2.5 Análisis de tiempos de ejecución

Estimación del Valor de T (Threshold):

Valor Estimado para T: Basado en los resultados de tus pruebas, el umbral óptimo (T) parece estar en el rango de 8000 a 10000. Este valor representa el tamaño del vector desde el cual la paralelización es beneficiosa en cuanto a rendimiento.
Evaluación de las Condiciones para el Umbral:
Condición para 0.8T (Aproximadamente 8000 en este caso):

El tiempo medio de ejecución del programa serie fue ligeramente mayor o comparable al del programa paralelo para vectores de este tamaño.
Esto indica que, para tamaños de vector menores al umbral estimado, la paralelización no compensa debido al overhead de lanzar los hilos.
Condición para 1.2T (Aproximadamente 12000 en este caso):

El tiempo medio de ejecución del programa serie fue menor que el del programa paralelo para vectores de este tamaño.
Esto sugiere que, para tamaños de vector mayores al umbral estimado, la paralelización sí resulta beneficiosa, compensando el overhead de lanzar los hilos.

Verificación del Umbral:
Modificación del Valor de T a un Valor por Debajo del Estimado (Ejemplo: 6000):

Al probar con un valor de T menor al estimado, se esperaría ver que el programa paralelo no mejora o incluso empeora el rendimiento en comparación con la versión serie para tamaños de vector cercanos a este nuevo T.
Modificación del Valor de T a un Valor por Encima del Estimado (Ejemplo: 14000):

Al probar con un valor de T mayor al estimado, se esperaría ver una mejora más significativa en el rendimiento del programa paralelo en comparación con la versión serie para tamaños de vector cercanos a este nuevo T.

Como conclusión:

Basado en el análisis y las pruebas realizadas, se concluye que el umbral óptimo para la paralelización del cálculo del producto escalar utilizando OpenMP está en el rango de 8000 a 10000. Este umbral asegura que la paralelización se realice solo cuando el tamaño del vector es lo suficientemente grande como para compensar el overhead asociado con el manejo de múltiples hilos.

2.6 (Opcional) Análisis exhaustivo (Hasta 1 punto extra)

Se ha creado un script en Python llamado “Ejercicio2,6.py” ejecutable con “python3 Ejercicio2,6.py” que generará una gráfica comparativa, una generada por nosotros se encuentra en el directorio del ejercicio 2.

La gráfica sugiere que:

Para tamaños de vector menores, hay un cruce entre las dos líneas, indicando que para vectores pequeños la versión serie puede ser más rápida que la paralela.
A medida que el tamaño del vector aumenta, la versión paralela muestra una mejora significativa en el tiempo de ejecución comparado con la versión serie, especialmente después de un cierto punto (umbral).

El tiempo de ejecución de la versión serie se mantiene relativamente constante o aumenta levemente con el tamaño del vector, mientras que el tiempo de ejecución de la versión paralela disminuye o se mantiene estable.

¿Es el valor de umbral distinto para tu ordenador que para el clúster?

El valor del umbral podría ser diferente para un ordenador personal y un clúster debido a diferencias en hardware, como el número de núcleos de CPU y la implementación de hyper-threading. Los clústeres suelen tener un entorno de hardware más potente y optimizado para tareas paralelas, lo que puede llevar a un umbral diferente en comparación con un ordenador personal.

¿Depende del número de hilos que se lancen?

Sí, el umbral puede depender del número de hilos lanzados. Si se lanzan más hilos que el número de núcleos físicos o lógicos disponibles, puede introducirse un overhead significativo debido al cambio de contexto y a la gestión de hilos, lo que podría elevar el umbral para la paralelización efectiva. Por otro lado, un número óptimo de hilos que

coincide con el número de núcleos puede llevar a un umbral más bajo, aprovechando al máximo el paralelismo sin incurrir en demasiado overhead.

Conclusión:
La gráfica indica que existe un tamaño de vector específico a partir del cual la paralelización mejora el rendimiento. Este punto es lo que definiríamos como el umbral óptimo para la paralelización.

Ejercicio 3:

N=1000

Versión3\#Hilos	1	2	3	4
Serie	9.389236	9.525883	9.202733	9.354581
Paralela – bucle1	8.137970	4.726475	2.352382	2.136312
Paralela – bucle2	9.400706	4.739196	2.075391	1.596634
Paralela – bucle3	9.257678	5.208558	2.343610	1.654919

Speedup

Versión3\#Hilos	1	2	3	4
Serie	1	1	1	1
Paralela – bucle1	1.1537	2.0154	3.9120	4.3788
Paralela – bucle2	0.9987	2.0100	4.4342	5.8589
Paralela – bucle3	1.0142	1.8288	3.9267	5.6526

N=1700

Versión3\#Hilos	1	2	3	4
Serie	86.926386	87.540083	87.014736	86.592515
Paralela – bucle1	74.273827	25.420254	16.047247	12.612120
Paralela – bucle2	87.566388	48.491734	23.406253	18.584674
Paralela – bucle3	86.922474	42.616282	20.322037	14.664248

Speedup

Versión3\#Hilos	1	2	3	4
Serie	1	1	1	1
Paralela – bucle1	1.1703	3.4437	5.4224	6.8658
Paralela – bucle2	0.9926	1.8052	3.7175	4.6593
Paralela – bucle3	1	2.0541	4.2817	5.9050

3.1 ¿Cuál de las tres versiones obtiene peor rendimiento? ¿A qué se debe? ¿Cuál de las tres versiones obtiene mejor rendimiento? ¿A qué se debe?

Peor rendimiento:

Para N=1000, la versión paralela del bucle 3 (el bucle más externo) tiene en general el peor rendimiento con un solo hilo y muestra una mejora modesta con más hilos.
Para N=1700, la tendencia es similar, con la versión paralela del bucle 3 mostrando el peor rendimiento con un solo hilo.
Posibles razones para el peor rendimiento:

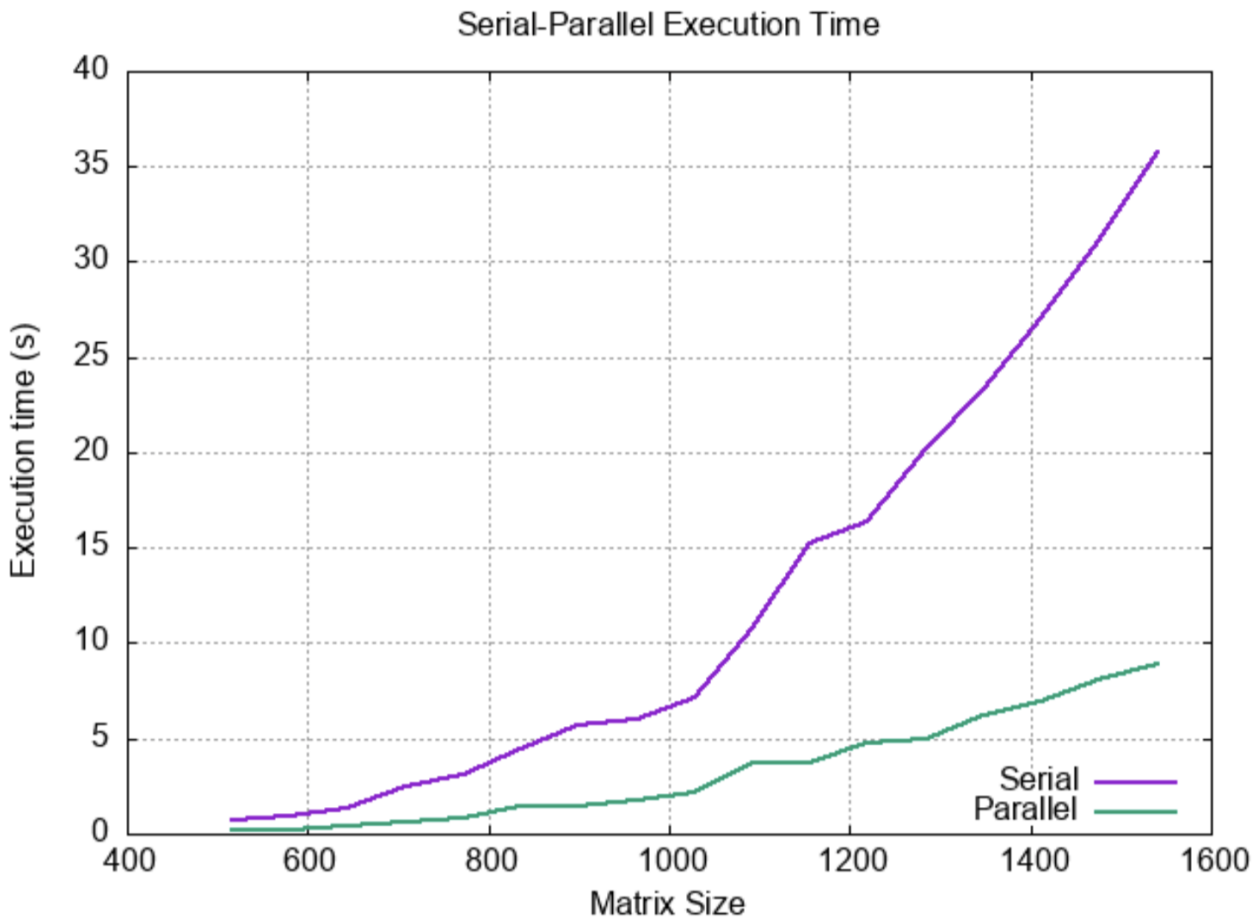
Paralelizar el bucle más externo podría no ser eficiente debido a que cada hilo puede estar realizando una cantidad relativamente pequeña de trabajo, lo que lleva a una sobrecarga significativa en la gestión de hilos. Además, si la paralelización del bucle externo causa demasiada sincronización o una fragmentación del trabajo que no se alinea bien con la caché del procesador, podría ser contraproducente.
Mejor rendimiento:

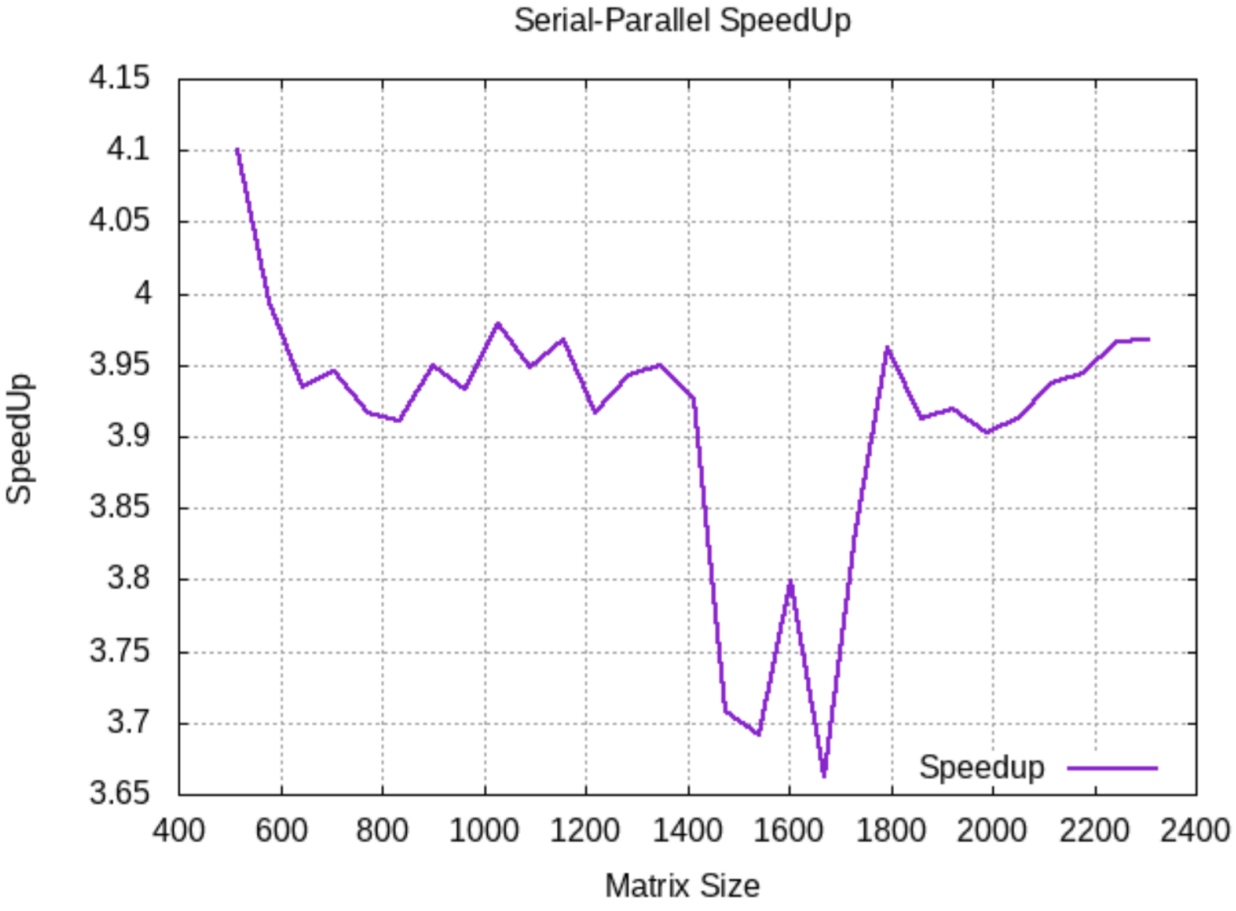
Para N=1000 y N=1700, paralelizar el bucle 1 (el más interno) ofrece un mejor rendimiento que los otros, especialmente a medida que aumenta el número de hilos.
Posibles razones para el mejor rendimiento:

Paralelizar el bucle más interno puede resultar en una utilización más eficiente de la caché y menos sobrecarga de sincronización porque cada hilo puede trabajar de manera más independiente en una sección de la matriz. Esto puede ser especialmente efectivo cuando la cantidad de trabajo en el bucle más interno es significativa, lo que permite que la paralelización sea más efectiva.

3.2 En base a los resultados, ¿cree que es preferible la paralelización de grano fino (bucle más interno) o de grano grueso (bucle más externo) en otros algoritmos?

Los resultados sugieren que, para la multiplicación de matrices, un enfoque de paralelización de grano fino es más efectivo. Es importante considerar que esto puede no aplicarse universalmente a todos los algoritmos o tipos de datos. Para algoritmos con patrones de acceso a datos y dependencias complejas, o cuando el tamaño del trabajo por hilo es grande, el grano grueso podría ser más adecuado. Es crucial analizar el patrón de acceso a los datos y las características del algoritmo para tomar una decisión informada sobre la estrategia de paralelización.





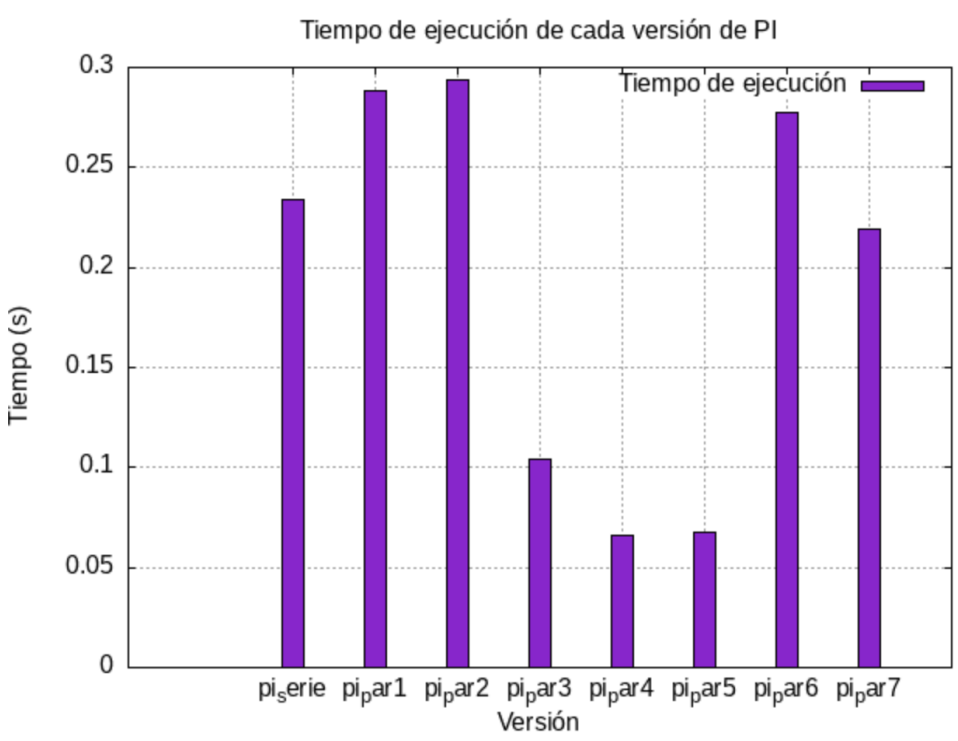
Hay algunos picos, pero se estabiliza el speedup tal y como era esperado.

Ejercicio 4:

4.1. ¿Cuántos rectángulos se utilizan en la versión del programa que se da para realizar la integración numérica? ¿Cuál es entonces el valor de h (ancho del rectángulo)?

Hay 100.000.000 rectángulos con una anchura de 0.00000001.

4.2. Ejecute todas las versiones paralelas y la versión serie. Analice el rendimiento (en términos de tiempo medio de ejecución y la aceleración o speedup) y si el resultado obtenido es correcto o no, reflejando estos datos en una tabla. En las versiones que el resultado no sea correcto, incluya una breve explicación de por qué no es correcto.



Version	pi_serie	pi_par1	pi_par2	pi_par3	pi_par4	pi_par5	pi_par6	pi_par7
Tiempo	0.234075	0.288575	0.293462	0.104022	0.066311	0.067298	0.277821	0.219386
Speedup	1.000000	0.811142	0.797633	2.250256	3.529982	3.478179	0.842540	1.066958
Pi result	3.141590	3.141590	3.141590	3.141590	3.141590	3.141590	3.141590	3.141590

Todos los programas han calculado correctamente el valor de π.

4.3. En la versión pi_par2, ¿tiene sentido declarar sum como una variable privada? ¿Qué sucede cuando se declara una variable de tipo puntero como privada?

Declarar sum como variable privada en pi_par2:

En pi_par2, la variable sum está declarada como un puntero y se le asigna memoria para cada hilo. Hacer sum privada tiene sentido solo si cada hilo maneja su propio conjunto de sumas parciales, lo cual parece ser el caso. Sin embargo, la manera en que se gestiona este puntero puede influir en el resultado final y el rendimiento.

Punteros privados: Cuando se declara un puntero como privado, cada hilo obtiene su propia copia del puntero, pero todos estos punteros apuntan a la misma ubicación de memoria (a menos que se asigne memoria de forma individual en el bloque paralelo). Esto puede llevar a condiciones de carrera si los hilos intentan modificar los datos en esa ubicación de memoria.

4.4. ¿Cuál es la diferencia entre las versiones pi_par5, pi_par3 y la versión pi_par1? Explique lo que es False sharing e indique qué versiones se ven afectados por ello (y en qué medida). ¿Por qué en pi_par3 se obtiene el tamaño de línea de caché?

Diferencias:
pi_par1: Versión básica de paralelización sin mecanismos específicos para evitar conflictos de acceso a variables compartidas.

pi_par3: Implementa técnicas para evitar el false sharing, como separar las sumas de cada hilo en la memoria (padding) para evitar que estén en la misma línea de caché.

pi_par5: Utiliza la directiva critical para proteger la actualización de la variable pi, evitando conflictos, pero potencialmente causando un cuello de botella.

False Sharing:
Se refiere a la situación donde múltiples hilos actualizan variables que, aunque son independientes, residen en la misma línea de caché. Esto causa invalidaciones innecesarias de la caché y reduce el rendimiento.

Afectados: pi_par1 puede verse afectado por false sharing si diferentes hilos actualizan partes de un arreglo que residen en la misma línea de caché.

Tamaño de línea de caché en pi_par3:
Se obtiene para implementar un padding efectivo, asegurándose de que las sumas parciales de cada hilo no residan en la misma línea de caché y así evitar el false sharing.

4.5. Explique el efecto de utilizar la directiva critical. ¿Qué diferencias de rendimiento se aprecian? ¿A qué se debe este efecto?

La directiva critical asegura que solo un hilo a la vez puede ejecutar el bloque de código protegido, lo cual es vital para operaciones que deben ser atómicas. Sin embargo, esto puede reducir el rendimiento al crear un punto de serialización en el código.

4.6. Ejecute la versión pi_par6 del programa. ¿Qué diferencias de rendimiento se aprecian? ¿A qué se debe este efecto?

En pi_par6, se utiliza la cláusula #pragma omp for para distribuir las iteraciones del bucle for entre los hilos. Esto puede mejorar el rendimiento en comparación con versiones que no distribuyen el trabajo de manera tan eficiente.

4.7. ¿Qué versión es la óptima y por qué?

La versión óptima dependerá de cómo cada técnica de optimización maneje el problema específico. Las versiones que equilibran la paralelización eficiente (como la distribución

del trabajo en pi_par6) y minimizan los problemas de sincronización (como el false sharing en pi_par3) tienden a ofrecer mejor rendimiento.

Ejercicio 5:

5.0. Compile y ejecute el programa usando como argumento una imagen de su elección. Examine los ficheros que se hayan generado y analice el programa entregado.

El programa genera para cada imagen de entrada tres imágenes de salida: image_grad.jpg, image_grad_denoised.jpg e image_grey.jpg. En términos generales, el programa desruida las imágenes proporcionadas, lo cual es muy útil si se quieren utilizar imágenes para un análisis científico. Por ejemplo, si la imagen está corrompida con ruido de sal y pimienta (lo que significa que bits individuales han sido alterados), puedes suavizar este ruido con el filtro gaussiano.

5.1. El programa incluye un bucle más externo que itera sobre los argumentos aplicando los algoritmos a cada uno de los argumentos (señalado como Bucle 0). ¿Es este bucle el óptimo para ser paralelizado? Responda a las siguientes cuestiones para complementar su respuesta.
a. ¿Qué sucede si se pasan menos argumentos que número de cores?
b. Suponga que va a procesar imágenes de un telescopio espacial que ocupan hasta 6GB cada una, ¿es la opción adecuada? Comente cuanta memoria consume cada hilo en función del tamaño en pixeles de la imagen de entrada

Si queremos procesar más imágenes de las que hay núcleos, no es óptimo paralelizar el bucle 0, porque los hilos comparten núcleos y, por lo tanto, recursos, y el procesamiento puede llevar a condiciones de carrera e inconsistencias de datos. Se podría preferir un paralelismo de grano fino.

a) Si paralelizamos el bucle 0 y pasamos menos argumentos que el número de núcleos y el bucle está paralelizado, el programa aún podría funcionar normalmente, sin embargo, se desperdiciaría tiempo creando más hilos de lo necesario.
b) En el caso de una imagen tan grande como 6GB no sería conveniente paralelizar el bucle 0 porque cada hilo consumiría más o menos el triple del tamaño de la imagen de entrada (6GB) ya que por cada argumento se generan tres imágenes. En consecuencia, podríamos tener carreras de datos y otros problemas relacionados con límites de memoria.

5.2. Durante la práctica anterior, observamos que el orden de acceso a los datos es importante. ¿Hay algún bucle que esté accediendo en un orden subóptimo a los datos? Corríjalo en tal caso.
a. Es imprescindible que el programa siga realizando el mismo algoritmo, por lo que solo se deberían realizar cambios en el programa que no cambien la salida.
b. Explique por qué el orden no es el correcto en caso de cambiarlo.

Los datos ya se acceden en el orden óptimo, por lo que no hay necesidad de cambiarlo. De hecho, si se cambiara el orden de acceso a los datos, el orden ya no sería correcto, porque el resultado no sería el mismo.