Universidad Nacional
de La Plata

Facultad de Informática

Sistemas Paralelos

Trabajo Práctico 2
Programación en memoria compartida

Alumnos:
Torrico, Fabrizio Benjamín
Castro, Lautaro Germán

Grupo 8

Índice

1. Detalles de experimentación

1.1. Hardware utilizado

Este trabajo se realizó sobre la **Partición XeonPhi** del cluster proporcionado por la cátedra. El equipo cuenta con las siguientes características:

- Procesador Manycore Intel Xeon PHI KNL (Knights Landing) 7230.
- 128GB de RAM.
- 64 cores que operan a 1.3GHz.
- Caché L1 con 32KB para datos y 32KB para instrucciones por core.

Debido a que las experimentaciones requieren pruebas con hasta 8 hilos ejecutando paralelamente, los requerimientos del equipo descrito son suficientes.

1.2. Pruebas realizadas

En este trabajo el problema a probar es una simple operatoria con matrices:

$$R = \frac{maxA * maxB - minA * minB}{PromA * PromB} \times [A \times B] + [C \times D]$$

Presentando tres algoritmos: secuencial, paralelo con pthreads y paralelo con openMP. Donde las pruebas realizadas cuentan con las siguientes variaciones:

- Tamaño de problema = { 512, 1024, 2048, 4096}
- Cantidad de hilos = { 2, 4, 8}

2. Soluciones planteadas

2.1. Optimizaciones generales

Dado que todos los algoritmos fueron escritos a partir de la versión secuencial, en todos se encontrarán las siguientes optimizaciones:

- **Localidad de los datos**
  - Las matrices fueron organizadas internamente por filas o columnas de acuerdo a como debe procesarse respecto a la operación aplicada. También se utilizó un algoritmo de multiplicación por bloques para el producto de matrices. Haciendo un uso óptimo de la localidad espacial.
  - Cada recorrido de matriz fue aprovechado incluyendo la mayor cantidad de operatoria posible dentro del recorrido (localidad temporal). Por ejemplo el mínimo, máximo y promedio se obtienen en un único recorrido por cada matriz.
- **Optimización de caché:** Para el tamaño de bloque en la multiplicación se buscó un número que permita que los bloques o submatrices entraran en la caché L1 del hardware utilizado. Para ello se utilizó el siguiente cálculo:

$$2 * (\text{tamañoDeBloque})^2 * \text{tamañoPalabra} = \text{tamañoDeCache}$$

Siendo que tamaño de palabra es 8 bytes porque usamos matrices con datos de tipo double, y el tamaño de caché L1 es 32KB, el cálcula da un resultado de 45 aproximadamente. Si bien, por temas de alineación en memoria parecería ser más adecuado utilizar un bloque de 32 ya que es la potencia de 2 más cercana, las pruebas realizadas, cuyos resultados se expresan en la Tabla 3.1, denotaron que el tamaño óptimo es de 64.

- **Expresiones de cómputo repetido:** Se evitaron utilizar expresiones de cómputo que se repitan en cada iteración de un loop. Por ejemplo, expresiones como i*N o j*N para acceder a una fila o columna de una matriz para una operatoria, son realizadas de manera previa a entrar al loop que las utiliza.
- **Optimización del compilador:** El compilador gcc ofrece varios niveles de optimización. Todos los algoritmos fueron compilados con un nivel -O3, excepto el algoritmo de pthreads que fue compilado con un nivel -O2, debido a que en las experimentaciones el nivel -O3 causaba fallos en el algoritmo, lo cual probablemente se debe a los cambios agresivos que realiza este nivel para optimizar el código.

2.2. Algoritmos Paralelos

2.2.1. Idea general

Como se nombró anteriormente, este trabajo presenta una operatoria con matrices, donde se puede articular muy facilmente una **descomposición de datos**, en el que cada hilo va realizar el mismo cómputo (cómputo regular) sobre un subconjunto del conjunto de datos original. Donde luego, debe tenerse en cuenta las barreras pertinentes de acuerdo a la precedencia de operaciones.

Teniendo en cuenta lo expresado anteriormente, la solución general puede resumirse de la siguiente manera:

1) Creación de hilos

2) Realizar operación

   I. Distribuir el conjunto de datos

   II. operar sobre el subconjunto

3) Esperar en un barrera si es necesario, sino paso (2).

2.2.2. Algoritmo Pthreads

De acuerdo a la idea general, la implementación con esta herramienta presenta tres incógnitas a resolver:

- **Distribución de datos:** Esto se resuelve generalmente con dividr el espacio total por la cantidad de hilos, y luego asignar una sección de acuerdo al id de cada hilo.
- **Barrera:** Pthreads proporciona un tipo de dato `pthread_barrier_t` con una interfaz simple de usar.
- **Exclusión mutua:** Se utiliza la librería `semaphore.h` que proporciona una interfaz simple para el uso de semáforos.

Optimizaciones realizadas

Sumado a las optimizacionez generales:

- **Sobrecarga creación/destrucción de hilos:** Debido al cómputo regular del problema, se decidió realizar una única vez la creación de hilos y sincronizar con barreras las operaciones

pertinentes. De esta manera se evita la sobrecarga que llevaría crear/destruir los hilos por cada operación.

- **Sobrecarga en secciones críticas:** Se evitó toda acción innecesaria y se redujo los accesos a las secciones críticas. Por ejemplo, al obtener los mínimos, máximos y promedios, cada hilo primero trabaja con variables locales, y luego una vez analizada su porción de los datos, se realiza las comparaciones/operatoria necesaria con los datos compartidos.

Decisiones

Se describrem decisiones sobre factores con poco peso sobre el rendimiento final, pero que de igual manera consideramos correcto aclararlas:

- El cálculo del $factor = \frac{maxA*maxB-minA*minB}{PromA*PromB}$ para la multiplicación escalar es realizado por cada hilo. Debido a que este cálculo se realiza luego de obtener los mínimos de A y B, consideramos que es un cálculo muy simple como para armar un mecanismo de sincronización donde solo un hilo realiza el cálculo, lo cual a nivel intuitivo pareciera tener mas costo computacional que ese simple cálculo.
- Por cada operatoria (como está modularizada), se repite el cálculo de los índices para el subconjunto de datos que recorrera cada hilo. Estos cálculos aproximadamente se repiten tres veces en todo el programa (una por cada llamada repetida a un módulo). Si bien esto puede solucionarse sin complejidad alguna, la legibilidad del código empeora, por lo cual como la ganancia son un par de operaciones aritméticas, decidimos no modificar el algoritmo, ya que su impacto en rendimiento es muy bajo.

2.2.3. Algoritmo OpenMP

Similar a las problematicas principales de pthreads, la implementación en OpenMP presenta las mismas incognitas, sin embargo a diferencia de pthreads, OpenMP proporciona una interfaz de mas alto nivel que pthreads basado en directivas.

Controlandose el procesamiento paralelo a través de directivas de alto nivel:

- **Distribucion de datos** Se utiliza la directiva parallel for que distribuye automaticamente los hilos.
- **Barrera:** se encuentra de manera implicita al final de cada directiva paralela, se puede eliminar mediante la clausula nowait.
- **Exlusión mutua:** se realiza mediante diferentes directivas especificas para el proposito, como reduction o sum.

Decisiones de Diseño

- **Uso de `nowait`**: Se utiliza `nowait` en las secciones donde no hay dependencias de datos entre iteraciones para permitir que los hilos continúen ejecutándose sin esperar a los demás, lo que puede mejorar la utilización de los recursos y la eficiencia del paralelismo.
- **Uso de `schedule(static)`**: Se utiliza `schedule(static)` para distribuir equitativamente el trabajo entre los hilos, lo que puede ayudar a lograr una carga de trabajo balanceada y evitar desequilibrios de ejecución entre los hilos.
- **Reducción de Variables Compartidas**: Se utilizan cláusulas de reducción (`reduction`) para garantizar que las variables compartidas utilizadas en cálculos paralelos, como el mínimo, máximo y promedio, se manejen correctamente y se eviten problemas de concurrencia.

3. Pruebas de tiempo

3.1. Tamaño de bloque óptimo

N	Secuencial (bloque 16)	Secuencial (bloque 32)	Secuencial (bloque 64)
	Tiempo	Tiempo	Tiempo
512	0.6573	0.4185	0.3426
1024	5.3341	3.8149	2.8003
2048	43.8128	30.8074	23.7526
4096	373.6544	264.2447	216.0979

3.2. Rendimiento de todos los algoritmos

N	Secuencial	Pthread (2 hilos)			OMP (2 hilos)		
	Tiempo	Tiempo	Speedup	Eficiencia	Tiempo	Speedup	Eficiencia
512	0.3426	0.1976	1.7338	0.8669	0.2012	1.7027	0.8513
1024	2.8003	1.5278	1.8328	0.9164	1.4975	1.8699	0.9349
2048	23.7526	13.0906	1.8144	0.9072	12.4584	1.9065	0.9532
4096	216.0979	116.6434	1.8526	0.9263	110.9717	1.9473	0.9736

N	Secuencial	Pthread (4 hilos)			OMP (4 hilos)		
	Tiempo	Tiempo	Speedup	Eficiencia	Tiempo	Speedup	Eficiencia
512	0.3426	0.1797	1.9065	0.4766	0.1108	3.0920	0.7730
1024	2.8003	0.9816	2.8527	0.7131	0.7650	3.6605	0.9151
2048	23.7526	6.6064	3.5953	0.8988	6.3862	3.7193	0.9298
4096	216.0979	58.5076	3.6935	0.9233	58.7067	3.6809	0.9202

N	Secuencial	Pthread (8 hilos)			OMP (8 hilos)		
	Tiempo	Tiempo	Speedup	Eficiencia	Tiempo	Speedup	Eficiencia
512	0.3426	0.1463	2.3417	0.2927	0.0658	5.2066	0.6508
1024	2.8003	1.1007	2.5441	0.3180	0.3936	7.114	0.8892
2048	23.7526	4.4141	5.3810	0.6726	3.1610	7.5142	0.9392
4096	216.0979	31.8659	6.7814	0.8476	28.9039	7.4764	0.9345

4. Conclusiones

4.1. Secuencial vs Paralelo

Rendimiento

Como puede observarse, en la Sección 3.2, en términos de rendimiento, la diferencia es evidente, la paralelización es sumamente importante para mejorar el rendimiento, haciéndose cada vez más significativo, en magnitud, el tiempo ocupado cuando el tamaño del problema incrementa, además del speedup y eficiencia que reflejan valores altos. Por lo cual se vuelve una herramienta fundamental explotar el paralelismo cuando se busque la optimización de un algoritmo.

Complejidad y facilidad de implementación

En términos de implementación, la paralelización presenta mayor complejidad, ya que además de pensar la solución directa al problema como se hace en un algoritmo secuencial, deben tenerse en cuenta múltiples aspectos como:

- Administración de hilos
- Sincronización de procesos
- Exclusiones mutuas
- etc.

Donde además de aquellos aspectos, verificar la correctitud es más complejo debido a la naturaleza no determínistica de las soluciones.

4.2. Pthreads vs OpenMP

Rendimiento

En términos de rendimiento, se observaron resultados similares entre ambas implementaciones en la mayoría de los casos de prueba. Ambos enfoques lograron aprovechar eficientemente los recursos disponibles en el hardware utilizado, mostrando tiempos de ejecución comparables para tamaños de problema y cantidades de hilos similares.

Complejidad y Facilidad de Implementación

Pthreads

La implementación utilizando Pthreads requirió un mayor nivel de detalle en cuanto a la gestión de hilos y la sincronización de operaciones concurrentes. Fue necesario manejar explícitamente la creación y destrucción de hilos, así como también implementar mecanismos de exclusión mutua y barreras para garantizar la coherencia en los resultados.

OpenMP

Por otro lado, la implementación con OpenMP resultó ser más sencilla y menos propensa a errores debido a su enfoque basado en directivas. La mayoría de las tareas de gestión de hilos y sincronización fueron manejadas de manera transparente por el compilador, lo que redujo la complejidad del código y facilitó su mantenimiento.