Sistemas Distribuidos y Paralelos

Trabajo práctico promoción

29/05/2024

Profesor:

- Pousa, Adrián

Alumnos:

- Pacheco, Nicolas 01640/9
- Llamocca, Brian 02037/9

1.Introducción

En el siguiente informe se presenta la solución y sus diferentes implementaciones junto a sus métricas propuesta por el grupo 14 de la clase sistemas distribuidos y paralelos, al problema propuesto por la cátedra utilizando librerías como pthreads y MPI del lenguaje de programación C.

2.Consigna

1. Dados dos arreglos desordenados de N números enteros realizar un algoritmo que retorne si los dos arreglos contienen los mismos elementos. Pueden existir elementos repetidos y deben coincidir en el valor y cantidad.

Pautas:

- Asumir que inicialmente los dos arreglos son iguales, aunque desordenados, y si o si deben incluir elementos repetidos.
- Probar para valores de N = 2^{24}, 2^{25}, 2^{26} y 2^{27}
- Resolver con Pthreads y ejecutarlo en el XeonPHI para 8, 16, 32 y 64 hilos.
- Comparar la solución Pthreads con la solución secuencial ejecutada en el XeonPHI.
- Resolver con MPI y ejecutarlo en el Blade para las siguientes configuraciones:
  - 4 cores: usando 2 máquinas, 2 procesos en cada máquina
  - 8 cores: usando 2 máquinas, 4 procesos en cada máquina.
  - 16 cores: usando 2 máquinas, 8 procesos en cada máquina
- Comparar la solución MPI con la solución secuencial ejecutada en el Blade.

3.Material utilizado

Para las pruebas locales se utilizaron computadoras personales con las siguientes especificaciones:

- Intel i3 7100u : 16Gb de ram ddr4L 2133Mhz , 2 cores, 4 hilos a 2.40 GHz, 3MB cache
- Intel i9 10900k : 16Gb de ram ddr4 3600Mhz, 10 cores, 20 hilos, 3.7GHz, 20MB cache

Para los resultados contemplados en el informe se utilizaron las siguientes particiones del cluster brindadas por la facultad de informática de la Universidad Nacional de La Plata:

- **Intel Manycore** Intel Xeon Phi 7230: Posee 128GB de ram y 64 cores que operan a 1.3GHz, 32MB cache L2

Como cada core tiene caché L1 de instrucciones y datos, se asume máximo 256 KB de caché de datos, por lo que los bloques de trabajo no deberían superar este tamaño para el rendimiento óptimo.

- **Cluster Multicore** (partición Blade): Conformado por 16 nodos, donde cada nodo posee 8GB de RAM y 2 procesadores Intel Xeon E5405 de 4 cores cada uno que operan a 2.0GHz.

4.Resolución del problema

4.1. Estrategia resolución Secuencial

Para el abordaje del problema propuesto se decide partir el problema en 2 sub-problemas de resolución conocida, ordenar ambos vectores de números, y luego comparar los vectores ordenados posición por posición.

En el segmento de ordenamiento se opta por ordenar con el método merge-sort iterativo que consiste en dividir el arreglo en porciones reducidas, ordenarlas e ir mezclando las porciones ordenadas . El código desarrollado presenta varias mejoras respecto a otras versiones:

- El método merge-sort teóricamente es más eficiente que el ordenamiento burbuja, ya que presentan diferencias en el orden, siendo de $O(n * log_2 n)$ y $O(n^2)$ respectivamente.
- El merge-sort secuencial presenta mejor tiempo de ejecución que la versión recursiva, ya que no consume tantas llamadas a funciones.
- Al momento de hacer el merge, se utiliza una variable auxiliar inicializada al inicio del programa para no crear temporalmente arreglos dinámicos, en la cual iría el resultado del merge entre las porciones.
- Se utilizan bucles while en lugar de bucles for dentro del merge para disminuir la cantidad de operaciones a realizar en cada iteración (3 vs 5 respectivamente)
- Micro-optimizaciones que aprovechan operaciones de bajo nivel de c (+=, ++, *=, manejo de punteros en lugar de índices para reducir la cantidad de operaciones)
- Al finalizar el merge se opta por usar swap de punteros de vectores, intercambiando el puntero del vector original con el puntero temporal para evitar la asignación de valores del temporal hacia el vector original luego de cada llamada al merge.

Una vez finalizado el ordenamiento de los vectores se llama al método de comparación, en el cual los arreglos se comparan posición por posición. Como ambos vectores están ordenados, si se encuentra alguna posición en la que ambos tengan valores diferentes se activa un flag (en este caso llamado diferencia) con el que se asegura que no son iguales. En el peor caso se recorre todas las posiciones del vector y se verifica que ambos vectores son iguales, con orden $O(n)$.

Tiempo de ejecución

En la siguiente tabla quedan contemplados los tiempos de ejecución para los distintos tamaños del vector.

Tiempo de ejecución [*seg*]	**Tamaño de problema (N)**			
Arquitectura de Ejecución	2^{2^4}	2^{2^5}	2^{2^6}	2^{2^7}
Xeon PHI	15.066302	31.164180	64.465414	133.166936
Blade	7.956265	16.317448	33.458753	68.609852

Tabla 1: *Tiempo de Ejecución secuencial en ambas arquitecturas según tamaño de datos*

4.2. Estrategia resolución Paralelo

El abordaje del problema es similar al secuencial: primero se ordena ambos vectores, (uno a la vez), y luego se comparan los vectores ordenados posición por posición.

El método de ordenamiento para maximizar el paralelismo es utilizar la misma estrategia de ordenamiento por niveles: merge-sort iterativo con swap, en el cual se divide el vector en partes iguales para cada hilo de ejecución, y cada hilo ordena de forma independiente (para minimizar la sincronización). Cuando todos los segmentos de vector están ordenados se selecciona la mitad de los hilos que se encargan de mezclar su parte del vector con la del hilo contiguo, mientras que la otra mitad se queda esperando a que los hilos terminen. Este proceso se repite en cada nivel, el cual será el resultado de $log_2(T)$, con T la cantidad de hilos creados.

La lógica sigue hasta que queda 1 solo hilo mezclando los últimos 2 segmentos. Este algoritmo para T hilos es de $O(n*\log_2(n) / T)$.

Para la implementación en paralelo del algoritmo mergesort se tuvo en consideración ciertas cuestiones:

- Se ordena de a 1 vector a la vez para que los accesos a memoria sean lo más consistentes y cercanos posibles, lo que disminuye los fallos de caché.
- El diezmado de los vectores no se hace de forma lineal, sino que los hilos activos toman la carga del hilo siguiente, para que la caché L1 del hilo vuelva a acceder a los mismos elementos ya ordenados con los del hilo siguiente, disminuyendo los fallos de cache.
- Los hilos nunca se eliminan sino que se duermen en barreras, para no tener que destruir hilos varias veces durante la ejecución, disminuyendo el overhead.

Para la comparación de los vectores ordenados se trabaja de manera aislada para minimizar la sincronización y comunicaciones, de manera que cada hilo tiene un segmento de trabajo equitativo para chequear. Se maneja un flag global el cual todos los hilos chequean en cada iteración, y si algún hilo encuentra una diferencia en su segmento de comparación, prende el flag y finaliza la búsqueda. En el peor de los casos, cada hilo recorre completo su segmento asignado, resultando con un algoritmo de orden O(n/T).

Tiempo de ejecución

En la siguiente tabla se mostrarán los tiempos de ejecución de la solución utilizando pthreads ejecutadas en el cluster ManyCore utilizando XeonPHI:

Tiempo de ejecución	**Tamaño de problema (N)**			
Unidades de procesamiento	2^{24}	2^{25}	2^{26}	2^{27}
Secuencial	15.036685	31.156600	64.465414	133.166936
8 hilos	2.625788	5.390475	10.826396	22.327974
16 hilos	1.939083	3.918018	7.951043	15.376282
32 hilos	1.593795	3.090203	6.910788	13.703501
64 hilos	1.421963	2.787120	5.943731	12.712958

Tabla 2: *Tiempo de ejecución para Xeon PHI según tamaño de datos y número de hilos*

Para el tiempo secuencial se tiene en cuenta los tiempos obtenidos con el algoritmo secuencial mencionado en el capítulo anterior.

Speedup

Speedup	**Tamaño de problema (N)**			
Unidades de procesamiento	2^{24}	2^{25}	2^{26}	2^{27}
8 hilos	3,030049	3,027089	3,090479	3,072820
16 hilos	4,103107	4,164720	4,208096	4,462057
32 hilos	4,992025	5,280381	4,841525	5,006739
64 hilos	5,595269	5,854591	5,629251	5,396844

Tabla 3: *Speedup para Xeon PHI según tamaño de datos y número de hilo*

Eficiencia

Eficiencia	Tamaño de problema (N)			
Unidades de procesamiento	2^{2^4}	2^{2^5}	2^{2^6}	2^{2^7}
8 hilos	0,378756	0,378386	0,386310	0,384103
16 hilos	0,256444	0,260295	0,263006	0,278879
32 hilos	0,156001	0,165012	0,151298	0,156461
64 hilos	0,087426	0,091478	0,087957	0,084326

Tabla 4: *Eficiencia para Xeon PHI según tamaño de datos y número de hilo*

Escalabilidad

Como se puede apreciar en las tablas realizadas, la escalabilidad para el algoritmo de ordenamiento con merge-sort iterativo optimizado con swap, para esta arquitectura no es escalable. Lo que se puede notar al chequear cualquier columna, que se compare 8 hilos, contra las potencias superiores, la eficiencia decae bruscamente, lo que indica que la solución no es fuertemente escalable.
También se debe observar que sobre la diagonal, al duplicar la potencia de cómputo y el tamaño del problema, la eficiencia también decae fuertemente, lo que indica que el problema NO ES débilmente escalable. Por lo la solución aunque está muy optimizada respecto a la solución “ordenamiento burbuja”, “ordenamiento merge-sort recursivo”, y es mejor que el “ordenamiento merge-sort iterativo sin permuta de vectores”. Pero la solución no es escalable, como se puede esperar de una solución $O(n \log_2(n))$, cuando duplica los datos, la potencia de cómputo necesaria para igualar el tiempo de ejecución supera al doble .

4.3. Estrategia resolución Distribuido

El abordaje del problema es similar al secuencial y el paralelo : primero se ordena ambos vectores, y luego se comparan los vectores.
Como se dispone de 2 máquinas independientes, se optimiza la comunicación, y la sincronización que agrega overhead innecesario. Una máquina actuará como maestro y la otra como esclavo. En cada máquina tendrá 1 solo vector, y su vector auxiliar para el ordenamiento. Ambas máquinas ordenarán bajo el mismo algoritmo merge-sort iterativo con swap del ejercicio anterior. **Una observación que notamos para 2 cores con varios procesos es que al momento de enviar y recibir mensajes, se necesitan varios canales para la comunicación en lugar de 1 solo canal. En este caso asignamos un canal para cada proceso que hace el merge entre su porción de vector y el subvector del proceso contiguo** Cuando ambas tengan su vector ordenado, el maestro le enviará paquetes de datos para que el esclavo compare de forma paralela y verifique si hay alguna diferencia, con cada confirmación del maestro se enviará un nuevo paquete de datos, hasta encontrar alguna

diferencia en los vectores, o recorrer ambos vectores y corroborar que son iguales. El orden de la solución de ordenamiento es $O(\frac{n*log_2 n}{T})$ y la solución de comparación es $O(\frac{n}{T})$.

Se busca un balance en el tamaño del bloque, si es muy pequeño hay muchas comunicaciones, lo que introduce mucho overhead. pero si el bloque es muy grande, el esclavo tardará demasiado en comenzar a trabajar, y será similar a enviar el vector de datos entero (sin saber si la diferencia esta al inicio del vector)

Como optimización se usará carga especulativa, asumiendo que se van a enviar varios bloques, estos bloques se enviarán de forma especulativa para optimizar el tiempo muerto de comunicación.

Tiempo de ejecución

En la siguiente tabla se mostrarán los tiempos de ejecución de la solución utilizando MPI ejecutadas en el cluster multicore utilizando Blade:

Tiempo de ejecución [*seg*]	**Tamaño de problema (N)**			
Unidades de procesamiento	$\mathbf{2^{24}}$	$\mathbf{2^{25}}$	$\mathbf{2^{26}}$	$\mathbf{2^{27}}$
Secuencial	7.956265	16.317448	33.458753	68.609852
4 cores	4.018233	8.113246	19.925542	33.336395
8 cores	3.151561	6.342399	12.833736	25.833787
16 cores	2.772356	5.564337	11.234331	39.842411

Tabla 5: *Tiempo de ejecución para Blade según tamaño de datos y número de procesos*

Speedup

Speedup	**Tamaño de problema (N)**			
Unidades de procesamiento	$\mathbf{2^{24}}$	$\mathbf{2^{25}}$	$\mathbf{2^{26}}$	$\mathbf{2^{27}}$
4 cores	1.980041	2.011211	1.679189	2.058107
8 cores	2.524547	2.572756	2.607094	2.655819
16 cores	2.869857	2.932505	2.978259	1.722031

Tabla 6: *Speedup para Blade según tamaño de datos y número de procesos*

Eficiencia

Eficiencia	Tamaño de problema (N)			
Unidades de procesamiento	2^{2^4}	2^{2^5}	2^{2^6}	2^{2^7}
4 cores	0.495010	0.502803	0.419797	0.514527
8 cores	0.315568	0.321595	0.325887	0.331977
16 cores	0.179366	0.183282	0.186141	0.107627

Tabla 7: *Eficiencia para Blade según tamaño de datos y número de procesos*

Escalabilidad

Como se puede apreciar en las tablas 7, los resultados de eficiencia carecen de la misma tendencia que la solución implementada en Pthreads, ya que la eficiencia decae por columna, y por diagonal. Como es de esperar ya que el algoritmo utilizado en esta resolución es igual que el de pthreads pero implementado en memoria distribuida, lo que hace mantener la tendencia de la escalabilidad. La solución No es escalable

Observaciones del Problema

El proceso de merge-sort lineal es altamente eficiente para ordenar grandes listas de datos de manera secuencial, a la hora de paralelizar luego de que cada proceso ordene de forma lineal su trozo de datos, estos se deben mezclar, esta mezcla se debe agregar barreras de sincronización, o enviar grandes trozos del vector, lo que hace que la 2da etapa del ordenamiento tenga más overhead, y por este efecto reducción del problema , en cada iteración de la combinación se usan la mitad de hilos que la anterior, lo reduce el paralelismo irremediablemente. Por estas razones el problema nunca va a llegar al speedup ideal en el merge-sort iterativo paralelizado o distribuido.

Un caso diferente es el método de detectar diferencias, ya que dependiendo de donde esté la diferencia se puede ahorrar mucho tiempo de procesamiento al paralelizar, o igualarlo.
Aunque no se midió de manera directa, en las pruebas todas se hicieron con 0 diferencias, para que el chequeo se recorra completo y tener una cota de tiempo estable. En el caso de haber 1 sola diferencia en la última dirección del vector, el speedup sería prácticamente perfecto. Pero si la diferencia estuviera en la primera posición del vector, ambos chequeos siempre tardan lo mismo (o incluso peor el paralelo, ya que consume tiempo de crear las tareas o hilos). Pero si la diferencia estuviera en el primer segmento del último bloque, la solución paralela resolvería la comparación en 1 acceso, y la versión secuencial tendría que recorrer todos los bloques hasta llegar al último bloque, lo que haría un speedup superlineal.

Como el problema es tan dependiente de la ubicación y la cantidad de diferencias, para ser comparaciones de tiempo justas, se utilizaron vectores sin diferencias, y con la misma semilla aleatoria para que los números de los vectores de prueba sean aleatoriamente idénticos.

5. Parámetros de Test

Para ejecutar y probar los programas de la manera más justa posible, se fijó la semilla de los números aleatorios en “2000” para que los números aleatorios sean estrictamente iguales en todas las pruebas.
Todas las pruebas se hicieron sin valores diferentes, para que los recorridos sean completos en todos los casos.

La compilación de los scripts se hizo en el frontend y luego se mandó a ejecutar los batch con las diferentes configuraciones de parámetros.
Se adjuntan 3 scripts utilizados de ejemplo para testear el programa en secuencial, paralelo y distribuido.
Cada script tiene versiones alternativas con parámetros hardcodeados con diferente cantidad de núcleos, hilos, o tamaño de la tarea.
Las carpetas se crearon previamente desde la interfaz del explorador de archivos del clúster.
Los archivos comprimidos en el “SDP-G14.zip” son la versión final utilizada el dia de la fecha de entrega

6. Conclusiones

El problema propuesto tiene varias implementaciones posibles, pero la lógica de resolución más eficiente siempre divide el problema en 2 segmentos, el segmento de ordenar, y de comparar. Aunque la comparación de datos puede dar valores muy dispersos incluso superlineal cuando hay 1 solo diferencia. La carga computacional del ordenamiento es muy superior a la comparación, por lo que para todas las implementaciones la comparación es despreciable en la tendencia de rendimiento y eficiencia paralela.

Los resultados son concluyentes y consistentes con la escalabilidad del ordenamiento numérico, despreciando el aporte de la comparación lineal de vectores. Que en ambas implementaciones de la solución, no sea escalable le da consistencia a la conclusión de que la lógica de resolución del ordenamiento no es escalable, sea cual sea la implementación y la arquitectura.

Para que una solución sea al menos débilmente escalable, debe tener un O(n) o superior.
Otra conclusión que se observó durante el desarrollo del proyecto, y la ejecución de los programas en las computadoras personales, fue el costo extra de la comunicación en memoria distribuida, contra memoria compartida. Específicamente para las implementaciones utilizadas, en Pthreads no había consumo mover grandes estructuras de datos, ya que se compartían con punteros, y en la implementación de MPI, las comunicaciones de los procesos, involucra buffers de gran tamaño.

No se ven reflejados los números en las tablas propuestas, pero hay un rendimiento extra de la ejecución de la solución con memoria compartida sobre la de memoria distribuida para la misma máquina, bajo las mismas condiciones. Aunque queda al descubierto que las arquitecturas de memoria distribuida son económicamente más escalables que las de memoria compartida.

6. Bibliografía:

Arquitectura XeonPhi :

- https://ark.intel.com/content/www/xl/es/ark/products/series/75557/intel-xeon-phi-processors.html

Recomendaciones respecto al uso de MPI_Send:

- https://www.mcs.anl.gov/research/projects/mpi/sendmode.html
- https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf