Reporte Final: Multiprocesadores

Eduardo Larios Fernández

Instituto Tecnológico de Estudios Superiores de Monterrey

A00569364 – 08/11/2018

Contenido

Resumen

En el reporte siguiente se presentan los resultados obtenidos de la comparación de múltiples lenguajes de programación y sus respectivos frameworks para paralelismo y concurrencia. El algoritmo analizado fue “Quicksort”, uno de los algoritmos más comunes de ordenamiento.

Quicksort es un algoritmo de divide y conquista que realiza múltiples particiones a un arreglo a partir de un pivote precalculado. La estrategia de partición es una de las fuentes más significativas de varianza en su desempeño cuando es utilizado en un solo hilo de ejecución de manera recursiva.

He intentado utilizar el método de partición de Hoare en la mayoría de las implementaciones, el cual indica con apuntadores el inicio y el fin del arreglo. Estos dos apuntadores se acercan hasta que detectan una inversión de valores que se encuentra en el lugar erróneo.

Este método de partición generalmente realiza menos “swaps” que otros esquemas de partición. Por otro lado, la desventaja de este esquema es la reducción de rendimiento a $O(n^2)$ cuando el arreglo ya se encuentra ordenado, otra posible desventaja es que no mantiene un ordenamiento estable, sin embargo, este garantiza que el lado izquierdo es menor al pivote y este en si es menor al lado derecho (low < pivot < high), por lo que no hay riesgo de recursión infinita, reduciendo así la complejidad de paralelización.

Introducción

Para realizar una comparación objetiva de las tecnologías se intentó mantener el algoritmo lo más similar posible, descartando los cambios requeridos para su ejecución en paralelo, manteniendo las secciones de llenado del arreglo, partición del arreglo y distribución de los valores.

Las tecnologías utilizadas en este reporte incluyen el lenguaje de programación *C* con el framework de OpenMP y el compilador GCC. Para la sección de Java se utilizó la versión 10.0.2 del JDK de Oracle para su compilación. Para la sección de multithreading se utilizó la implementación normal de java.lang.Thread para su ejecución paralela regular y java.util.concurrent.RecursiveAction para la implementación utilizando el Fork-Join Framework.

Se incluyó adicionalmente una implementación de C# utilizando .NET Core 2.1.4 y System.Threading.Tasks para su paralelización con Parallel.Invoke y su comparación con Java.

Finalmente se realizó una implementación paralela con múltiples GPU de NVIDIA utilizando CUDA v8.0.61 con NVCC. Esta implementación fue la única no ejecutada en Windows 10 1809.

Tecnologías

OpenMP – Lenguaje: C

El venerable lenguaje de programación C cuenta con múltiples tecnologías que permiten aprovechar a fondo el rendimiento del hardware moderno, así como los casi 50 años de avances en técnicas de compilación y optimización.

La implementación de C no utiliza threads nativos con la librería lpthreads, si no el framework open source de OpenMP. Este es un API incluido con la mayoría de los compiladores de C, como por ejemplo GCC y Clang. Esta tecnología permite el uso de memoria compartida en la mayoría de los sistemas operativos y plataformas de cómputo. La ventaja principal de OpenMP es la simpleza de su modelo y la posibilidad de integrarlo al gusto junto con otras tecnologías.

El ambiente de ejecución fue el siguiente:

OS:	Windows 10 Pro 1809 de 64-bits
Compilador:	GCC v6.3 -O3
CPU:	Intel Core i7 7700HQ 4C8T @ 3.80 GHz 6MB Cache L3
RAM:	16 GB DDR4 @ 2400 MHz
GPU:	NVIDIA GTX 1050 4096 MB GDDR5

Los resultados fueron los siguientes:

N = 300,000 **Iteraciones = 10**	**Single-Thread**	**Multi-Thread**
Execution Time	8552.7953 ms	1661.3366 ms
Processor Use	10% - 17%	94% - 97%
Memory Use	1.8 MB	1.9 MB
Speedup	1.0x	5.1487x

Conclusión:

Mientras que el uso de memoria se mantuvo básicamente estático entre ambas implementaciones, inmediatamente se puede notar una mejora significativa en el tiempo de ejecución al utilizar #omp tasks con OpenMP. La utilización del procesador fue bastante alta a llegar incluso al 97% de un procesador con 4 núcleos y 8 hilos. OpenMP además de ser bastante eficiente es muy sencillo de implementar, por lo que es un framework bastante efectivo y útil para quienes están familiarizándose con la programación paralela.

Threads – Lenguaje: Java

Java es un lenguaje de programación orientado a objetos que se ejecuta sobre una máquina virtual (JVM) lo que le permite tener código portable que se ejecuta de manera idéntica en múltiples plataformas sin la necesidad de portar el código con ajustes en diferentes arquitecturas y sistemas operativos. El uso de threads es considerablemente senillo en Java, ya que estos solo requieren de una clase que implemente la interfaz Runnable o que extienda la clase Thread. En este caso se utilizó el JDK más reciente de Oracle v10.0.2.

La dificultad de escalar la versión básica del algoritmo fue considerablemente mayor ya que el intentar crear un Thread cada que se ejecutaba la recursión era inviable debido al gran número de hilos que eran creados, y su dificultad de sincronizarse para el resultado final.

El ambiente de ejecución fue el siguiente:

OS:	Windows 10 Pro 1809 de 64-bits
Compilador:	JDK v10.0.2
CPU:	Intel Core i7 7700HQ 4C8T @ 3.80 GHz 6MB Cache L3
RAM:	16 GB DDR4 @ 2400 MHz
GPU:	NVIDIA GTX 1050 4096 MB GDDR5

Los resultados fueron los siguientes:

N = 300,000 **Iteraciones = 10**	**Single-Thread**	**Multi-Thread**
Execution Time	1432.200 ms	848.500 ms
Processor Use	17%	26%
Memory Use	27.1 MB	27.4 MB
Speedup	1.0x	1.6886x

Conclusión:

El uso de Java Threads no fue muy sencillo debido a la complejidad de la partición no fue posible implementar más threads en la versión regular del algoritmo. El speedup utilizando solamente dos hilos fue bastante cercano al doble de velocidad, con una mejora de casi 70% sobre la versión de un solo hijo.

Sin embargo, este tipo de problema que depende de la reducción del problema de manera dinámica con el paradigma de divide y conquista es mucho más sencillo de manera recursiva, el cual será el siguiente caso por analizar con el framework Fork-Join de Java, el cual mostró un rendimiento superior a la implementación paralela regular.

Fork-Join – Lenguaje: Java

Java es un lenguaje de programación orientado a objetos que se ejecuta sobre una máquina virtual (JVM) lo que le permite tener código portable que se ejecuta de manera idéntica en múltiples plataformas sin la necesidad de portar el código con ajustes en diferentes arquitecturas y sistemas operativos.

En la implementación siguiente se utilizó el Fork-Join framework de la librería estándar de Java. Este esquema es considerablemente más sencillo de utilizar, ya que permite el uso de particiones que son asignadas a un pool de threads de manera automática, y deja de asignarlos cuando llega a un límite inferior.

El ambiente de ejecución fue el siguiente:

OS:	Windows 10 Pro 1809 de 64-bits
Compilador:	JDK v10.0.2
CPU:	Intel Core i7 7700HQ 4C8T @ 3.80 GHz 6MB Cache L3
RAM:	16 GB DDR4 @ 2400 MHz
GPU:	NVIDIA GTX 1050 4096 MB GDDR5

Los resultados fueron los siguientes:

N = 300,000 Iteraciones = 10	**Single-Thread**	**Multi-Thread ForkJoin**
Execution Time	1432.200 ms	26.500 ms
Processor Use	17%	13.6 %
Memory Use	27.1 MB	173 MB
Speedup	1.0x	54.03x

Conclusión:

Con el uso del Fork-Join framework se evita mucho trabajo repetido y se mantiene una cantidad adecuada de threads para el trabajo. Al ser la división de trabajos más exacta y con menos redundancia se reduce el tiempo de creación de los hilos, al estar estos ya en el pool. Gracias a la versatilidad de este framework se logra un increíble speedup de 54 veces más rápido que la versión seria.

El rendimiento de este no debe ser subestimado, ya que es incluso superior al alcanzado con OpenMP o incluso CUDA. Honestamente es impresionante la optimización posible por la máquina virtual de Java. El único “pero” de este método es el uso incrementado de memoria, sin embargo, en el caso en el que se busque el mayor speedup posible en un algoritmo, el uso de threads y memoización siempre debe ser considerado.

CUDA Runtime – Lenguaje: C/C++

CUDA son las siglas de Compute Unified Device Architecture (Arquitectura Unificada de Dispositivos de Cómputo) que hace referencia a una plataforma de computación en paralelo, incluyendo el compilador de NVCC y un conjunto de herramientas de desarrollo creadas por NVIDIA que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en GPU de NVIDIA.

En la implementación siguiente se utilizó CUDA para paralelizar el algoritmo de Quicksort con el esquema de partición de Hoare. La dificultad de paralelizar el algoritmo en este caso es resultado del esquema de memoria de las GPU, el cual es lineal y cuenta con múltiples bloques con hilos de ejecución.

El ambiente de ejecución fue el siguiente:

OS:	Ubuntu 16.04.5 LTS
Compilador:	NVCC 8.0.61
CPU:	Intel Core i7 7700K 4C8T @ 3.80 GHz 6MB Cache L3
RAM:	Unknown Amount of RAM
GPU:	(2x) NVIDIA GTX 1080 8GB GDDR5X

Los resultados fueron los siguientes:

N = 300,000 **Iteraciones = 10**	**Single-Thread**	**Multi-Thread Fork-Join**
Execution Time	5130.8263 ms	209.0601 ms
Processor Use	Stats not available	Stats not available
Memory Use	Stats not available	Stats not available
Speedup	1.0x	24.5454x

Conclusión:

El uso de CUDA para la paralelización del algoritmo de Quicksort fue increíblemente efectiva. El algoritmo tiene en este caso un esquema de partición distinto. El algoritmo hace uso de la recursión regular para la partición en el caso de un arreglo de tamaño mayor al threshold, de otra manera utiliza un selection sort que en casos de arreglos chicos escala mejor con múltiples threads.

Haciendo uso de este esquema doble de ordenamiento se alcanza un speedup de hasta 24x veces más rápido que la versión serial. Algo a mencionar es que el uso de NVCC para compilar el código serial de C puede no ser tan efectivo como el uso de un compilador estándar como GCC, el cual puede contar con mejores optimizaciones debido al tiempo que tiene en el mercado.

Referencias:

1. https://en.wikipedia.org/wiki/OpenMP
2. https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html
3. https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html
4. https://es.wikipedia.org/wiki/CUDA
5. https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.invoke?view=netframework-4.7.2
6. https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080/
7. Repositorio de Código: https://github.com/EduardoLarios/QuicksortComparison