Programación Paralela

Teórica 01 - Introducción

Segundo Semestre, 2025

Introducción

En esta materia vamos a estudiar una técnica de programación llamada **programación paralela**, que nos permite escribir programas que pueden ser ejecutados en diferentes unidades de procesamiento al mismo tiempo.

Los microprocesadores que tenemos en nuestras computadoras personales y celulares se basan en una unidad central de procesamiento (CPU) que ejecuta un cierto número de *threads* (hilos) en paralelo, que ejecutan un código secuencial de instrucciones. A lo largo de la historia, estas CPUs se fueron llevando a límites de rendimiento cada vez mayores, donde gracias a la miniaturización de los componentes, la mayor cantidad de núcleos, la mayor velocidad del reloj, mejores formas de enfriamiento y la mejora en la eficiencia energética, se logró aumentar la cantidad de procesamiento que se podía hacer en un solo chip.

Esto produjo que, la mayor parte de las aplicaciones, se vieran beneficiadas de estos avances de hardware para incrementar la velocidad de las propias aplicaciones donde, esencialmente, el mismo software funcionaba mucho más rápido a medida que se iban realizando mejoras en estas unidades de procesamiento secuenciales. Sin embargo esto muchas veces se lo conoce como escalabilidad vertical , donde se busca mejorar el rendimiento simplemente agregando más recursos. El problema con este enfoque es que eventualmente se llega a un límite de la cantidad de recursos que se pueden agregar y debemos encontrar otras formas de optimización.

Para ilustrar los conceptos básicos de la programación paralela y escalable, necesitamos elegir un lenguaje simple de programación que soporte paralelismo masivo. En este curso vamos a utilizar CUDA C para nuestros ejemplos y ejercicios, que no es más que una extensión del lenguaje de programación C con una nueva sintaxis e interfaces que hace que el software generado, no sólo pueda trabajar con sistemas que tengan CPUs, sino también con GPUs, como unidades de procesamiento masivamente paralelas. El modelo de programación CUDA fue desarrollado por NVIDIA que permite aprovechar la potencia de cómputo de varias unidades (GPU) de procesamiento a la vez.

La idea de parlelizar no es nueva, pero históricamente los clusters de procesamiento paralelo estaban limitados a supercomputadoras y clusters bajo la órbita de gobiernos y grandes empresas que podían costearlos (Sutter and Larus [1]). Lo cual no implica un reemplazo de las CPUs sino un complemento ya que la escalabilidad vertical seguirá siendo importante ya la cantidad de núcleos en la CPU posiblemente siga aumentando, lo cual redundará en un paralelismo interno de las CPUs.

El *ratio* de rendimiento entre una CPU y una GPU puede ser de 1:10 (o más) para ciertas operaciones, ya que las CPUs están optimizadas para ejecutar código secuencial de forma *performante* relizando, en la medida de lo posible, un paralelismo interno de instrucciones que es transparente para el usuario. Las CPUs, además, poseen internamente grandes cachés que les permiten manejar la inherente latencia del acceso a dispositivos externos lentos, mientras que las GPUs por el otro lado tienen cachés mucho más pequeñas ya que sólo necesitan tener una gran optimización para mover grandes cantidades de datos tanto *in* como *out* de la DRAM

(*Dynamic Random Access Memory*) de la GPU porque fueron creadas para la industria de los juegos donde el *frame buffering* es un requerimiento crítico.

Complejidad algorítmica

En materias anteriores (programación o algoritmos) se estudiaron algoritmos secuenciales y la importancia de la complejidad algorítmica. Esta complejidad algorítmica, es un parámetro con el cual podemos medir la eficiencia de un algoritmo en términos de la cantidad de recursos que utilizaar ya sea **tiempo** de CPU y o **espacio** de memoria requeridos para cada solución algorítmica. Esto nos permite comparar los diferentes algoritmos independientemente del lenguaje en que los vayamos a implementar.

Cuando diseñamos un algoritmo, es importante tener en cuenta la cantidad de recursos que se van a utilizar, tratando de estimar cuál será su complejidad en tiempo y en espacio de acuerdo a la solución que hayamos propuesto. Hay diferentes métricas de estimación, aunque las más comunes son estimar **el caso promedio** y, aún más importante, **el peor caso posible**. Por ejemplo, agregar un elemento al final de un *array* dinámico en cualquier lenguaje de programación de alto nivel, tiene una **complejidad promedio** constante, ya que cuando se crea un array dinámico, el lenguaje lo crea de un tamaño dado (aún si el usuario no lo especifica). Por el otro lado, si el *array* dinámico estuviera lleno, estaríamos en el *peor caso* posible, ya que el lenguaje, tendría que conseguir más memoria para añadir ese nuevo elemento y luego copiar todos los elementos a la nueva posición de memoria.

En esta materia vamos a ver sólo la notación de complejidad para el peor caso, que se representa con la notación **Big O**. En palabras simples, lo que representa la complejidad algorítmica en el peor caso es, la cantidad de tiempo de CPU o cantidad de memoria que se requiere para resolver un problema en función del tamaño de la entrada. Esta función **Big O** nos da una cota asintótica superior de la cantidad de recursos que se necesitan para resolver un problema en función del tamaño de la entrada (fig: 1).

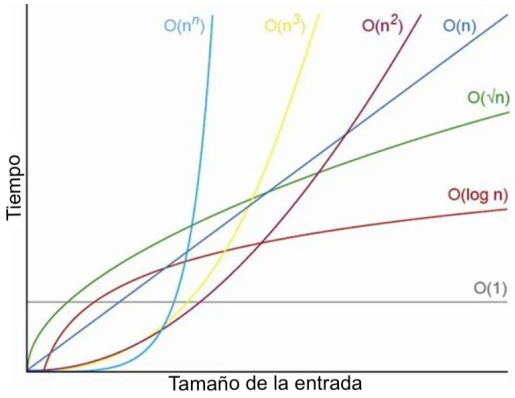

Figure 1: Big O - Comparación de complejidades

Diferentes implementaciones de un mismo algoritmo que resuelve un problema dado pueden tener diferentes complejidades, Por ejemplo, para hay varios algoritmos que resuelven el problema de ordenar un *array* (arreglo) de elementos de tamaño n. El más popular por su facilidad de implementación es el algoritmo de burbujeo que, si bien ordena los elementos, tiene una complejidad de $O(n^2)$, pero si utilizamos un algoritmo de ordenamiento más eficiente, nos encontramos con el *merge sort* que tiene una complejidad de $O(n \cdot \log\ n)$. Ambos algoritmos resuelven el mismo problema (ordernar un array), pero uno es mucho más eficiente que el otro. A veces, descubrir un algoritmo más eficiente no es tan fácil.

El n es la cantidad de elementos que tenemos como input de nuestro algoritmo. Con lo cual, un algoritmo que tiene una entrada de tamaño n y se ejecuta con una cierta función de tiempo $f(n)$, diremos que el algoritmo se comporta con una complejidad asintótica de $O(g(n))$ si existe una constante C y un valor n_0 tal que el valor absoluto de $f(n)$ es menor o igual a C por el valor absoluto de $g(n)$ para todo n mayor a n_0. (Wilf [2]). Formalmente:

$$f(n) = O(g(n))\ (x\ \rightarrow\ \infty)\ si\ \exists\ C, n_0\ /\ |f(n)| \leq C\,|g(n)|\ (\forall\ n > n_0)$$

Las diferentes complejidades algorítmicas deberán ser realizables; y por *realizables* queremos decir que la capacidad de resolver las instancias deseadas de un problema estén dentro de nuestros recursos disponibles. En la práctica, la factibilidad es muy dependiente del contexto y no es particularmente *portable* (portátil) entre diferentes problemas y situaciones. Un principio común se mantiene en casi todas las situaciones: **una tasa de crecimiento exponencial en el consumo de algún recurso limita la aplicación del uso de ese método a todas las instancias, excepto a las más pequeñas**. En otras palabras, si nuestros algoritmos tienden a tener complejidades exponenciales (o más) en el tamaño de la entrada, no importa cuántos recursos tengamos, no podremos resolver instancias grandes del problema. Por lo tanto, la factibilidad ha llegado a significar que la tasa de crecimiento del recurso está acotada por un una complejidad polinomial en el tamaño de la entrada. Esto nos da la noción común de que un problema tiene una solución secuencial factible sólo si tenemos un algoritmo de tiempo polinomial, es decir, sólo si cualquier instancia de tamaño n del problema se puede resolver en tiempo $n^{O(1)}$. Aunque ampliamente reconocido como muy simplista, la dicotomía entre algoritmos polinomiales y no polinomiales ha demostrado ser un discriminador poderoso entre aquellos cálculos que son factibles en la práctica y aquellos que no lo son (Greenlaw, Hoover, and Ruzzo [3]).

Lógicamente hay problemas para los cuales no se han encontrado algoritmos polinomiales, por ejemplo, la factorización entera de un número muy grande es un problema que no puede resolverse en tiempo polinomial con las CPUs y GPUs actuales. Por eso es que la seguridad de muchos sistemas de encriptación se basan en ello. Si bien excede lárgamente el contenido de esta materia, hay algoritmos cuánticos que pueden resolver este problema en tiempo $O((\log n)^{O(1)})$ ([4]).

¿Por qué no se utilizan GPUs para todo?

En la se mencionó que el *ratio* de rendimiento entre una CPU y una GPU puede ser de 1:10 (e incluso mayor), entonces una pregunta válida sería **¿por qué no se utilizan GPUs para todo?**. La respuesta a esto no es única, por un lado las GPUs están diseñadas para tener alto *throughput* de operaciones matemáticas y de transferencia a memoria y suelen ser buenas para tareas de alta latencia que requieren pasajes grandes de información desde y hacia la memoria (óptimos para el procesamiento paralelo). Por el otro lado las CPUs están optimizadas para realizar tareas de baja latencia y tareas inherentemente secuenciales donde sorbepasan holgadamente a la *performance* de una GPU. Por esto es que estos sistemas se llaman híbridos, ya que hay un hilo conductor secuencial y otros que se ejecutan en paralelo (a demanda).

La parelelización es importante ya que hay aplicaciones que trabajan con datos que pueden ser procesados en paralelo utilizando varias GPUs. Por ejemplo en biología donde a veces las observaciones están limitadas por las observaciones que se pueden hacer a nivel molecular con lo cual, creando modelos que simulen las actividades subyacentes de estas

moléculas a través de modelos computacionales se pueden hacer simulaciones que permitan hacer predicciones sobre el comportamiento de estas moléculas.

Como veremos en la sección siguiente, no todas las aplicaciones pueden ser paralelizadas, y en muchas aplicaciones, sólo un porcentage del tiempo de ejecución de una aplicación puede ser paralelizado, lo cual implica que es el algoritmo utilizado quien define la mejora de velocidad, no necesariamente el número de procesadores; en algún momento llegaremos a un límite donde finalmente por más que tengamos más recursos no se podrá paralelizar más el algoritmo. A esto se lo conoce como ley de Amdahl.

Al desarrollar CUDA C, NVIDIA no sólo ha permitido paralelizar aplicaicones, sino que también ha permitido bajar el costo del desarrollo del software paralelo que estaba restringido a supercomputadoras masivamente paralelas. Esto se debe a que el mercado actual de GPUs es enorme ya que casi todas las PCs actuales tienen algún tipo de GPU instalada, habiendo más de 1000 millones de GPUs en el mundo que pueden ser utilizadas con CUDA, con lo cual al facilitar el desarrollo de aplicaciones paralelas, NVIDIA logra hacer que el desarrollo de software paralelo sea más accesible para todos.

Desafíos en la programación paralela

Podríamos pensar entonces que la programación paralela puede ser una solución general para resolver todos los problemas de *performance* y debería utilizarse siempre. Aunque como vimos con la ley de Amdahl, no todos los problemas son 100% paralelizables, y en general sólo podremos intentar paralelizar una parte de la aplicación.

El primer problema, es que es difícil diseñar algoritmos paralelos, ya que requiere pensar los problemas de formas anti-intuitivas que, muchas veces, no son sencillas de resolver para que tengan el mismo nivel de complejidad computacional que los algoritmos secuenciales. La performance de los algoritmos paralelos son muy sensibles a los datos de entrada, ya que esencialmente la paralelización se basa en el procesamiento de datos de manera limitada por la velocidad de acceso a memoria. Pero aún si salváramos estos problemas que son, méramente técnicos, nos encontraríamos con la pregunta más profunda: **¿todos los problemas son teóricamente paralelizables?**.

Los límites de la programación paralela

En la teoría de complejidad computacional, P es una clase [1] de complejidad que contiene todos los problemas de decisión que pueden ser resueltos por una máquina de Turing determinista en tiempo polinomial. Sin embargo, Nicholas Pippenger realizó una investigación exhaustiva sobre circuitos con profundidad polilogarítmica y tamaño polinomial y sugirió una interesante clase de complejidad a estudiar que sería la clase de problemas solucionables por máquinas paralelas en tiempo polilogarítmico [2] usando un número polinomial de procesadores. Esta clase se conoce como **NC** (por *Nick's class*). (Stockmeyer [5])

Los problemas $P-completos$ son de interés porque parecen carecer de soluciones altamente paralelas. Es decir, los diseñadores de algoritmos han fallado en encontrar algoritmos *NC*. En consecuencia, la promesa de la computación paralela, es decir, que aplicar más procesadores a un problema puede acelerar en gran medida su solución, parece estar ser imposible en toda la

[1] En matemáticas, una clase es una colección de conjuntos (u otros objetos matemáticos) que pueden ser definidos unívocamente por una propiedad compartida por todos los miembros de ese conjunto

[2] Un problema es polilogarítmico si su complejidad es $O((\log n)^k)$ para algún k

clase de problemas $P - completos$. Dejando abierta la siguiente pregunta: (Greenlaw, Hoover, and Ruzzo [3])

$$P \stackrel{?}{=} NC$$

Esta clase P se puede pensar como los problemas tratables (tesis de Cobham), por lo que NC se puede pensar como los problemas que se pueden resolver eficientemente en una computadora paralela. NC es un subconjunto de P porque los cálculos paralelos polilogarítmicos se pueden simular mediante cálculos secuenciales polinomiales. Pero no se sabe si $NC \stackrel{?}{=} P$, pero **la mayoría de los investigadores sospechan que esto es falso**, lo que significa que probablemente hay algunos problemas tratables que son **"intrínsecamente secuenciales"** y no se pueden acelerar significativamente utilizando paralelismo. Así como la clase NP-completo se puede pensar como "probablemente intratable", así que **la clase P-completo, cuando se utilizan reducciones NC, se puede pensar como "probablemente no paralelizable" o "probablemente *intrínsecamente secuencial*"**.

Con lo cual, la respuesta a la pregunta de si todos los problemas son paralelizables es que **NO, no todos los problemas son paralelizables ya que hay ciertos problemas que son intrínsecamente secuenciales**.

Modelos y Lenguajes de Programación Paralela

En el pasado hubo muchos lenguajes de programación que fueron propuestos para abordar el problema de la programación paralela. Algunos de estos lenguajes son:

- **OpenMP**: es una API de programación en paralelo que se basa en directivas de compilador y funciones de biblioteca para permitir la paralelización de aplicaciones en sistemas de memoria compartida.
- **MPI**: es una biblioteca de paso de mensajes que permite la comunicación entre procesos en un sistema distribuido.
- **OpenACC**: Es un estándar de programación para la computación paralela desarrollado por Cray, CAPS, Nvidia y PGI. El estándar está diseñado para simplificar la programación paralela de sistemas heterogéneos CPU/GPU.
- **OpenCL**: Es un estándar abierto para la programación de sistemas heterogéneos que consta de CPUs, GPUs y otros dispositivos de cómputo.
- **CUDA**: Es una plataforma de computación paralela y un modelo de programación desarrollado por NVIDIA para sus GPUs.

References

[1] H. Sutter and J. Larus, "Software and the concurrency revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry.," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005, ISSN: 1542-7730. DOI: 10.1145/1095408.1095421. [Online]. Available: https://doi.org/10.1145/1095408.1095421.

[2] H. S. Wilf, *Algorithms and complexity*. AK Peters/CRC Press, 2002.

[3] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to parallel computation: P-completeness theory*. Oxford university press, 1995.

[4] A. Ekert and R. Jozsa, "Quantum computation and shor's factoring algorithm," *Reviews of Modern Physics*, vol. 68, no. 3, p. 733, 1996.

[5] L. Stockmeyer, "Classifying the computational complexity of problems," *The Journal of Symbolic Logic*, vol. 52, no. 1, pp. 1–43, 1987. DOI: 10.2307/2273858.