Programación Paralela

Teórica 01 - Introducción

Segundo Semestre, 2025

Introducción

Como regla general queremos que nuestros algoritmos sean realizables y por *realizables* queremos decir la capacidad de resolver las instancias deseadas de un problema dentro de nuestros recursos disponibles. En la práctica, la factibilidad es muy dependiente del contexto y no es particularmente 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 son exponenciales 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. [1]

Los microprocesadores se basan en una unidad central de procesamiento (CPU) que ejecuta un cierto número de *threads* (hilos) en paralelo. Estas CPUs se fueron llevando a límites de rendimiento donde gracias a la miniaturización de los componentes, la mayor cantidad de núcleos, la mayor velocidad, 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.

Históricamente, la mayor parte de las aplicaciones se vieron beneficiadas de estos avances de hardware para incrementar la velocidad de las propias aplicaciones donde, esencialmente, el mismo software funcionaba 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.

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. CUDA C extiende el lenguaje de programación C con una nueva sintaxis e interfaces que hace que los programadores puedan trabajar con sistemas que tengan tanto CPUs como GPUs masivamente paralelas.

Si bien la escalabilidad vertical seguirá siendo importante y la cantidad de núcleos posiblemente siga aumentando mejorando el paralelismo de las CPUs, hay aplicaciones que podrían ver un incremento de *performance* (rendimiento) mucho mayor si se utilizaran programas que se ejecutan en paralelo. Si bien esta idea no es nueva [2], estos programas paralelos estaban limitados a supercomputadoras y clusters de computadoras.

El *ratio* de rendimiento entre una CPU y una GPU puede ser de 1:10 se debe a que las CPUs están optimizadas para ejecutar código secuencial de forma *performante* relizando un paralelismo interno de instrucciones, pero dando la impresión de una ejecución secuencial. Poseen internamente grandes cachés que les permiten manejar la típica latencia del acceso a dispositivos externos lentos. Las GPUs por el otro lado tienen cachés mucho más pequeñas, pero están optimizadas 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.

¿Por qué no se utilizan GPUs para todo?

Las GPUs están diseñadas para la ejecución paralela y de alto *throughput* de operaciones matemáticas. Sin embargo, las CPUs son mucho mejores para realizar tareas de baja latencia donde sorbepasan holgadamente a la *performance* de una GPU. Por eso es que las computadoras de escritorio traen CPUs ya que muchas de las tareas de interacción con los usuarios tienen que ser de baja latencia.

¿Qué es CUDA?

CUDA es un modelo de programación desarrollado por NVIDIA que permite la utilización de ambas unidades de procesamiento (CPU y GPU) a la vez para ejecutar una aplicación. Con esto entonces podemos aprovechar la potencia de cómputo de varias unidades de procesamiento a la vez.

Sin embargo no hay que ser ingénuos y pensar que la única razón por la que se desarrolla esto es para aumentar la velocidad. En principio el factor económico es uno de los factores más importantes, ya que lo que se desea es utilizar la base instalada de procesadores, ya que el costo del desarrollo de software sólo se ve justificado por una población grande de clientes. Este había sido un problema grande ya que la creación de supercomputadoras masivamente paralelas, sólo había estado al alcance de pocos. Pero dado que el mercado de GPUs es enorme casi que podemos decir que prácticamente todas las PCs tienen algún tipo de GPUs, lo cual hace que existan más de 1000 millones de GPUs en el mundo que pueden ser utilizadas con CUDA.

¿Por qué paralelizar?

Hay aplicaciones que trabajan con datos que pueden ser procesados en paralelo que pueden ser masivamente paralelizadas ejecutando en varias GPUs. Por ejemplo en biología donde a veces las observaciones están limitadas por las observaciones que se pueden hacer a nivel molecular, es posible crear modelos que simulen las actividades subyacentes de estas moléculas a través de modelos computacionales.

Sin embargo no todas las aplicaciones pueden ser paralelizadas, sino que normalmente un porcentage del tiempo de ejecución de un aplicación puede ser paralelizado. Sin embargo la ley de Amdahl nos dice que el rendimiento obtenida por la optimización de una parte de un programa está limitada por la cantidad de tiempo que esa parte está en uso. Con lo cual, cuantas más secciones de un software podamos paralelizar, más rápido podremos lograr que se ejecuten, llegando a mejoras de $100\times$ o más en el rendimiento de un programa.

Hay que tener en cuenta, de todas formas, que la CPU es muy eficiente para correr aplicaciones secuenciales, con lo cual la GPU es un complemento de la CPU y no un reemplazo.

Desafíos en la programación paralela

Se podría pensar entonces que la programación paralela es una solución a todos los problemas de rendimiento.

Sin embargo, por un lado es complicado diseñar algoritmos paralelos que tengan el mismo nivel de complejidad computacional que los algoritmos secuenciales. Por el otro lado la programación paralela tiene límites de velocidad de acceso a memoria. La performance de los algoritmos paralelos son muy sensibles a los datos de entrada, ya que esencialmente la parelización se basa en el procesamiento de datos de manera paralela. Por último, pero no menos importante, los algoritmos paralelos suelen expresarse de manera de recurrencias, lo cual requiere que se piensen los diferentes problemas de formas no intuitivas.

Pero los problemas antes mencionados sólo son problemas técnicos que se podrían resolver con el tiempo y la suficiente dedicación, pero podríamos preguntarnos si **¿todos los problemas son paralelizables?**.

Los límites de la programación paralela

Una rama de la teoría de la complejidad computacional se dedica a identificar los problemas más difíciles en la clase P, que son los problemas que pueden ser resueltos en tiempo polinomial. Estos problemas son de interés porque parecen carecer de soluciones altamente paralelas. Es decir, los diseñadores de algoritmos han fallado en encontrar algoritmos NC, soluciones paralelas altamente factibles que toman tiempo polinomial en el logaritmo del tamaño del problema mientras usan sólo un número polinomial de procesadores. 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 ser rota por toda la clase de problemas P-completo. Esto se puede escribir como $P = NC$.

En la teoría de complejidad computacional, P es una clase 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 [1] usando un número polinomial de procesadores. Esta clase se conoce como **NC** (por *Nick's class*). [3]

La 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

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

subconjunto de P porque los cálculos paralelos polilogarítmicos se pueden simular mediante cálculos secuenciales polinomiales. Pero no se sabe si NC = 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, hay ciertos problemas que son intrínsecamente secuenciales y no se pueden paralelizar.

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

[2] 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.

[3] 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.