Índice general

Índice de figuras

Índice de cuadros

Resumen

Capítulo 1

Introducción

1.1. Motivación

1.2. Objetivo

El objetivo general de este trabajo consiste en:
Los objetivos particulares consisten en:

1.3. Metas

Se establecieron como metas:
Las contribuciones de este trabajo son:

1.4. Organización de la tesis

Este trabajo se desarrolla en .. capítulos, incluyendo esta introducción, y una sección adicional de apéndices que complementan la investigación.

En el Capítulo 2, se presentan los **Antecedentes**. Donde se hace...

En el Capítulo 3, **Capítulo...**, se presentan ...

...

Se concluye el trabajo con una sección de **Conclusiones y Trabajo Futuro**, capítulo..., en donde se exponen .... los comentarios finales y se sugiere el trabajo a relizar en próximas investigaciones.

Se anexan en la sección **Apéndices** los apartados: ...

Capítulo 2

Paralelismo

2.1. Motivación

2.2. Tipos de arquitectura

Tradicionalmente se dividen los tipos de sistemas en base a como tratan el flujo de datos y de instrucciones. Esto se conoce como taxonomía de Flynn [Pac11]. Están las siguientes categorías

- *SISD* (Single Instruction Single Data): sistemas sin paralelismo.
- *SIMD* (Single Instruction Multiple Data): sistemas que pueden aplicar una operación a vectores o arreglos de datos en una unidad de tiempo.
- *MIMD* (Multiple Data Multiple Instructions): sistemas donde cada unidad de procesamiento es independiente.

2.2.1. Sistemas *SISD*

2.2.2. Sistemas *SIMD*

En estos sistemas hay una única unidad de control con varias unidades aritmético lógicas (*ALU*). Al recibir la instrucción, la unidad de control avisa a todas las *ALUs* para que apliquen la operación al dato correspondiente. Este tipo de paralelismo se denomina **paralelismo de datos** .

Deben tener registros capaces de guarda vectores y operaciones optimizadas para leer y escribir de los elementos en estos registros. Todas las operaciones son síncronas y totalmente uniformes. Además de operaciones sobre los elementos, suelen haber operaciones que actúan sobre los vectores sin tener que acceder a cada elementos, como obtener la longitud del vector.

A pesar de solo ser factibles en operaciones muy restringidas, estos sistemas no suelen introducir problemas nuevos a los programas y su uso es bastante directo. Además, la mejora en rendimiento al usarlos suele ser aceptable.

Hoy en día, cada núcleo de un *CPUs* cuentan con instrucciones que permiten este tipo de operaciones. No son un sistema exclusivamente *SIMD*, ya que tener varios núcleos le dan características *MIMD*.

Las unidades de procesamiento gráfico (*GPU*) también tienen este tipo de funcionalidad. Además, como las imágenes suelen ocupar mucha memoria, tiene memoria de gran tamaño optimizada para el manejo con hilos a nivel de hardware . Curiosamente, esto hace que tenga mal desempeño en problemas pequeños. Al igual que los *CPUs*, tampoco son exclusivamente un sistema *SIMD*, ya que suelen tener más de un núcleo.

2.2.3. Sistemas *MIMD*

2.3. Tipos de Problemas

Pancake [Pan96] propone algunas reglas que a grandes rasgos permitan determinar si vale la pena dar una solución paralela a un problema.

1. Si el problema es de paralelismo perfecto, probablemente sea fácil de paralelizar y se obtengan ganancia considerable en rendimiento.

2. Si el problema es de paralelismo de *pipeline*, sólo valdría la pena paralelizarlo si se puede balancear la carga de trabajo entre las etapas.

3. Si el problema es de paralelismo totalmente síncrono, valdría la pena paralelizar dependiendo de que tan uniformemente se puede distriubir la carga.

4. Si el problema es de paralelismo vagamente síncrono, solo valdría la pena paralelizarlo si hay muy pocas interacciones entre procesos.

5. Un problema de paralelismo perfecto podría implementarse en un sistema *MIMD*, pero sería problemático hacerlo en un sistema *SIMD*.

6. Un problema de paralelismo de *pipeline* probablemente se desempeñe mejor en un sistema de memoria compartida o en un SMP (siempre y cuando cada etapa queda en una unidad de procesamiento). Podría funcionar decentemente en un sistema distribuido si la conexión entre etapas es lo suficientemente rápida.

7. Un problema totalmente síncrono probablemente se desempeñe mejor en un sistema SIMD. Pero esto requeriría que todas las operaciones sean uniformes. De no ser así, una opción decente sería un sistema de memoria compartida.

8. Para un problema vagamente síncrono la mejor opción sería un sistema de memoria compartida. Un sistema distribuido funcionaría siempre y cuando haya muchas operaciones entre cada interacción de los procesos.

9. Lenguaje

10. El tiempo de ejecución de una versión secuencia del programa se puede usar para estimar el desempeño del programa paralelo.

11. Debido a las restricciones de las secciones secuenciales del programa, probablemente no valga la pena paralelizar un programa con una fracción paralelizable menor a 0.95.

12. Hay que estimarlo con cuidado como cambia el desempeño del programa al cambiar la entrada. Este depende de la naturaleza del problema.

13. El desempeño final siempre será peor al estimado.

14. Aunque se puede intentar resolver pérdidas de tiempo surgidas por la concurrencia, en general dependerá de la naturaleza del problema y del equipo usado.

15. Un problema de baja granuralidad tendrá un desempeño decente en cualquier sistema. Un sistema de granuralidad media/alta probablemente solo sea decente en un sistema SIMD.

16. Para sistemas distribuidos, es útil estimar la equivalencia de mensaje.

2.4. Métricas de desempeño

2.5. Resumen

Capítulo 3

Sistemas de memoria compartida

3.1. Motivaciones

3.2. CPUs

3.3. GPUs

Kernels Indices Huesped y devices

3.4. TPUs

3.5. Resumen

Capítulo 4

Julia

4.1. Principios y características

4.1.1. Sistema de tipos

4.1.2. Técnicas de optimización

4.2. Resumen

Capítulo 5

Optimización sobre gráficas

5.1. Motivación

5.2. Distancia

Intuitivamente, una trayectoria es una manera de llegar de un vértice a otro a través de las aristas. Formalmente, un camino es una gráfica tal que existe un orden lineal de sus vértices

$$p = (x_0, x_1, \ldots, x_{k-1}, x_k)$$

de tal manera que dos vértices son adyacentes si y solo si son consecutivos en el orden. La longitud $l(p) = k$ del camino es la cantidad de arista que tiene.

Una trayectoria es un camino que no repite vértices, y una uv-trayectoria es una gráfica que sea trayectoria, que inicie en u y que termine en v. Es claro que entre un par de vértices puede haber más de una trayectoria. La longitud de la más corta se denomia la distancia $d(u, v)$ de los vértices, o infinito si no existe ninguna trayectoria.

Un problema básico en gráfica es que dada una gráfica G y dos de sus vértices u, v, encontrar su distancia. En su forma más simple, esto se puede hacer iniciando en u y revisando todos los vértices a distancia 1, que serían sus vecinos. Si no se encuentra v entonces habría que revisar a los vértices a distancia 2, que sería los vecinos de los vecinos que no hayan sido revisados ya. Esto se realiza sucesivamente hasta encontrar a v o hasta que no haya más vértices por revisar.

Este tipo de acciones se conocen como búsquedas, pues se busca un vértice que cumpla algo, en este caso que sea v. En particular, una búsqueda donde se revisan todos los vértices a una distancia dada antes de avanzar se conoce como una búsqueda en amplitud, o BFS por sus siglas en inglés.

Para decidir el siguiente vértice a revisar se requiere algún tipo sala de espera, donde se guarden los vértices vecinos al vértice actual que deberán ser visitados eventualmente. Para garantizar que se visiten a todos los vértices a distancia k antes de visitar alguno a distancia $k + 1$, basta notar que todo vértice a distancia $k + 1$ es adyacente a un vértice a distancia k. Así que basta con respetar el orden de exploración. Es decir, revisar siempre el vértice que lleva más tiempo en la sala de espera.

Esta estrucutura se conoce como una cola, y tiene dos operaciones. *push* que añade un elemento al final de la cola y *pop* que saca el primer elemento de la cola. Los detalles están en el algoritmo 1.

Proposición 5.1. *El algoritmo 1 devuelve la distancia entre dos vértices.*

Demostración. Sea □

Algorithm 1 Distancia usando BFS

Require: G gráfica, $u, v \in V(G)$
Ensure: Encontrar $d_G(u, v)$

```
Q ← []
i ← 0
push(Q, u)

while Q ≠ [] do
    x ← pop(Q)
    for v_x ∈ N(x) do
        if x = v then
            Return(i)
        end if

        if v_x no ha sido visitado then
            marcar v_x como visitado
            push(Q, v_x)
        end if

        i ← i + 1
    end for
end while

Return(∞)
```

5.2.1. Algoritmo de Dijkstra

5.2.2. Algoritmo de Bellman-Ford

5.3. Ciclos hamiltoneanos

5.3.1. Algoritmos genéticos

5.3.2. Enjambre de partículas

5.3.3. Colonia de hormigas

5.4. Flujos

5.4.1. Método de Ford-Fulkerson

5.5. Conjuntos independientes

5.6. Clanes

5.7. Resumen

Capítulo 6

Conclusiones y Trabajo Futuro

Apéndice A

Especificación de Julia

A.1. Sintaxis

A.2. Estructuras básicas

A.3. Sistema de tipos

Índice alfabético

Referencias

[Pac11] Peter Pacheco. *An Introduction to Parallel Programming*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. isbn: 9780123742605. doi: 10.1016/c2009-0-18471-4. url: https://doi.org/10.1016/c2009-0-18471-4.

[Pan96] C. Pancake. «Is parallelism for you?» En: *IEEE Computational Science and Engineering* 3.2 (1996), págs. 18-37. doi: 10.1109/99.503307.