Índice general

Resumen **IX**

1. Introducción **1**
1.1. Motivación . . . 1
1.2. Objetivo . . . 1
1.3. Metas . . . 1
1.4. Organización de la tesis . . . 1

2. Paralelismo **3**
2.1. Motivación . . . 3
2.2. Tipos de arquitectura . . . 3
2.2.1. Sistemas *SISD* . . . 3
2.2.2. Sistemas *SIMD* . . . 3
2.2.3. Sistemas *MIMD* . . . 4
2.3. Tipos de Problemas . . . 4
2.4. Métricas de desempeño . . . 5
2.5. Resumen . . . 5

3. Sistemas de memoria compartida **7**
3.1. Motivaciones . . . 7
3.2. CPUs . . . 7
3.3. GPUs . . . 7
3.4. TPUs . . . 7
3.5. Resumen . . . 7

4. Julia **9**
4.1. Principios y características . . . 9
4.1.1. Sistema de tipos . . . 9
4.1.2. Técnicas de optimización . . . 9
4.2. Resumen . . . 9

5. Optimización sobre gráficas **11**
5.1. Definiciones básicas . . . 11
5.2. Distancia . . . 12
5.2.1. Sin pesos . . . 12
5.2.2. Con pesos . . . 13
5.2.3. Con pesos negativos . . . 14
5.3. Flujos . . . 14
5.3.1. Método de Ford-Fulkerson . . . 14
5.4. $\mathcal{NP}$-Completez . . . 14
5.4.1. Ejemplos de problemas . . . 14
5.4.2. Métodos evolutivos . . . 14
5.5. Resumen . . . 14

6. Conclusiones y Trabajo Futuro **15**

A. Especificación de Julia **17**
A.1. Sintaxis 17
A.2. Estructuras básicas 17
A.3. Sistema de tipos 17

Referencias **21**

Índice de figuras

Índice de cuadros

Lista de algoritmos

1. BFS 13
2. Dijsktra 13

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

Dado un conjunto de elementos de cualquier tipo, se puede abstraer las relaciones entre ellos con diagramas. Cada elemento podría ser un punto que está unido por una línea a los demás puntos con los que está relacionado. Estos elementos podrían ser relaciones de amistad en una red social, vías de comunicación entre centros de abastecimiento, o conxiones de red entre servidores.

Estas abtracciones suelen ser suficientes para resolver problemas importantes en estos contextos. Por ejemplo, saber la ruta más corta para distribuir productos entre varios lugares, saber la mínima cantidad de servidores a retirar para colapsar una red o saber como maximizar el flujo de energía en una red eléctrica.

Muchos de estos problemas se pueden plantear como problemas de optimización. Varias técnicas, tanto algorítimicas como heurísticas, se han desarrollado para resolver estos tipos de problemas. A continuación se presentar algunas de ellas.

5.1. Definiciones básicas

Para los propósitos de este trabajo, se usarán las definiciones de [Die17]. En estas, se toma una *gráfica* G como una tupla $\langle V, E\rangle$, donde V es un conjunto no vacío y E es una familia de subconjuntos de V, todos de cardinalidad dos. Además, se pide que V y E sean disjuntos. Se suele denotar estos conjuntos como $V(G)$ y $E(G)$ para evitar ambigüedad.

Los elementos de V se denominan *vértices*, y los de E *aristas*. Para simplificar notación, una arista $\{u, v\}$ se suele denotar como uv. La cantidad de vértices en una gráfica se llama el *orden* y la cantidad de aristas el *tamaño*. Una arista e es *incidente* a un vértice v si $v \in e$. Si existe una arista uv, entonces u, v son *adyacentes*. El conjunto de vértices adyacente a un vértice v se denomia su *vecindad* y se denota como $N_G(v)$ o $N(v)$ si no hay ambigüedad.

Luego, sean G, H dos gráfica, y $\phi : V(G) \to V(H)$ una función que preserva adyacencias. Es decir, si $uv \in E(G)$, entonces $\phi(x)\phi(y) \in E(H)$. ϕ es un *homomorfismo*. Si ϕ^{-1} también es un homomorfismo, entonces ϕ es *isomorfismo*. En este caso se dice que G y H son isomorfas y se denota como $G \cong H$. En general no se hace distinción entre gráficas isomorfas.

Dadas dos gráficas G, H, si tanto los vértices como las aristas de H son subconjuntos de los vértices y aristas de G, H es una *subgráfica* de G y G es una *supergráfica* de H, que se denota como $H \subseteq G$.

5.2. Distancia

Intuitivamente, una trayectoria es una manera de llegar de un vértice a otro a través de las aristas. Formalmente, una *trayectoria* 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 en el camino.

Una uv-trayectoria es una subgráfica 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, que es 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.

5.2.1. Sin pesos

En su forma más simple, se puede considerar que todas las aristas representan la misma distancia. Entonces, lo que se busca es la trayectoria con menos aristas.

Esto se puede hacer iniciando en u y revisando todos los vértices vecinos, que serían los vértices a nivel uno. Luego, se revisan a los vértices en el nivel 2, que serían los vecinos de los vecinos que no hayan sido revisados ya. Esto se realiza sucesivamente hasta que no haya más vértices por revisar. Al final, el nivel de v corresponde a su distancia desde u. Esto no es tan obvio y se justificará formalmente más adelante.

Este tipo de acciones se conocen como búsquedas, pues se busca cierta propiedad de los vértices, en este caso el nivel de v. En particular, una búsqueda donde se revisan todos los vértices a una nivel dado 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. Inductivamente, si se procesan los vértices en orden de exploración, entonces todos los vértices del nivel k estarían en la sala de espera antes de que siquiera se explore alguno del nivel $k + 1$.

Una estrucutura que respete el orden de llegada, FIFO por sus siglas en inglés, 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, el más antigüo , de la cola. Los detalles del proceso están descritos en el algoritmo 1.

Intuitivamente, si un vértice v está a distancia k de r, existe una rv-trayectoria $p = (x_0 = r, x_1, \ldots, x_k = v)$ de logitud k. El nivel de r es cero. Luego, el nivel de x_1 es a lo más 1, pues es adyacente a r. Y no puede ser cero, pues entonces existiría una rv-trayectoria más corta, así que $l(x_1) = 1 = d(r, x_1)$. Este argumento se puede usar inductivamente para todos los vértices de p, y concluir que $l(x_i) = i = d(r, x_i)$, y en particular $l(v) = k = d(r, v)$. Formalmente, esto se enuncia en el teorema 5.1.

Teorema 5.1. *La función de nivel $l(v)$ devuelta por el algoritmo 1 corresponde a la distancia entre r y v. Es decir, para todo vértice v, $l(v) = d(r, v)$.*

Demostración. Sea $l(v) = bfs(G, r)$ la función de nivel y v un vértice de G. Cada vértice en nivel $k + 1$ es adyacente a un vértice en el nivel k, por lo que para cualquier vértice, se puede construir una rv-trayectoria de longitud $l(v)$ siguiendo estas adyacencias, así que $l(v) \geq d(r, v)$.

Para la otra desiguladad, se prosigue por inducción sobre $d(r, v)$. Con $d(r, v) = 0$, v es r y $l(r) = 0$. Para $d(r, v) > 0$, sea $(x_0 = r, \ldots, x_k = v)$ una rv-trayectoria de longitud mínima. Por hipótesis de inducción, $d(r, x_{k-1}) \geq l(x_{k-1})$. Luego, como x_{k-1} y v son adyacentes,

Algoritmo 1 Distancias usando búsqueda en amplitud(BFS)

Entrada G gráfica, r vértice
Salida Función de nivel l

```
1: funcion bfs(G, r)
2:     Q ← []
3:     push(Q, r)
4:     l(r) = 0
5:     mientras Q ≠ [] hacer
6:         x ← pop(Q)
7:         para todo v_x ∈ N(x) hacer
8:             si v_x no ha sido visitado entonces
9:                 l(v_x) = l(x) + 1
10:                marcar v_x como visitado
11:                push(Q, v_x)
12:            fin si
13:        fin para
14:    fin mientras
15:    devolver l
16: fin funcion
```

por la línea 8, hay dos posibilidades para $l(v)$. Primero, puede ser que v ya haya sido visitado. Como las visitas se hacen por niveles, entonces el nivel del padre de v es a lo más el nivel actual $l(x_{k-1})$. Así

$$l(v) \leq l(x_{k-1}) + 1 \leq d(r, x_{k-1}) + 1 = d(r, v)$$

En otro caso, v no ha sido visita, así que por la línea 9 se tiene que

$$l(v) = l(x_{k-1}) + 1 \leq d(r, x_{k-1}) + 1 = d(r, v)$$

En cualquier caso, se tiene que $l(v) \leq d(r, v)$. Y como se tiene la desigualdad opuesta, se puede concluir que $l(v) = d(r, v)$. □

Ahora, que pasaría si la distancia que representa cada arista no fuera uniforme. En este caso, el algoritmo BFS no necesariamente daría la trayectoria más corta. Este tipo de gráficas se llaman gráficas con pesos.

5.2.2. Con pesos

Para extender la definición de [Die17], se usarán conceptos de [BM08]. Se define a una gráfica con pesos como una tupla $\langle G, w \rangle$ donde G es una gráfica y w es una función que asigna pesos a las aristas $w : E \to \mathbb{R}$. Por ahora, digamos que los pesos son positivos.

Algoritmo 2 Distancia con pesos usando el algoritmo de Dijkstra

Entrada G gráfica, r vértice raíz
Salida Función de distancia $l(v) = d_G(r, v)$

5.2.3. Con pesos negativos

5.3. Flujos

5.3.1. Método de Ford-Fulkerson

5.4. $\mathcal{NP}$-Completez

5.4.1. Ejemplos de problemas

Trayectorias hamiltoneánas

Clanes

Conjuntos independientes

5.4.2. Métodos evolutivos

Algoritmos genéticos

Enjambre de partículas

Colonia de hormigas

5.5. 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

adyacentes, 11
aristas, 11

búsqueda en amplitud, 12

cola, 12
CPU, 3

distancia, 12

GPU, 3
gráfica, 11

homomorfismo, 11

ILP, 3
incidente, 11
isomorfismo, 11

longitud, 12

MIMD, 3
multiple issue, 3

orden, 11

paralelismo de datos, 3
pipelining, 3

SIMD, 3
SISD, 3
subgráfica, 11
supergráfica, 11

tamaño, 11
trayectoria, 12

vecindad, 11
vértices, 11

Referencias

[BM08] J.A. Bondy y U.S.R Murty. *Graph Theory*. 1st. Springer Publishing Company, Incorporated, 2008. isbn: 1846289696.

[Die17] Reinhard Diestel. *Graph Theory*. 5th. Springer Publishing Company, Incorporated, 2017. isbn: 3662536218. doi: 10.1007/978-3-662-53622-3. url: https://doi.org/10.1007/978-3-662-53622-3.

[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.