Índice general

Capítulo 1

Introducción

1.1. Motivación

Gran cantidad de problemas como simulación del clima, análisis de genomas, investigación en nuevas energía o aprendizaje de máquina[Pac11] requieren gran poder computacional para ser resueltos. Por lo que mayor poder de cómputo disponible aumenta la cantidad de problemas tratables. En el pasado, esto se lograba aumentando la velocidad y densidad de transistores en los procesadores. Sin embargo, esta técnica está llegando a un límite físico.

Así que en los últimos años, para obtener un aumento en el rendimiento se usan sistemas paralelos. Estos pueden variar desde un procesador con varios núcleos hasta redes de computadores. El tener varias unidades de cómputo permite obtener mejoras de rendimiento a pesar de no tener procesadores más veloces.

Pero hacerlo introduce problemas concernientes a la coordinación de todas las partes. En este trabajo se hará el análisis necesario para paralelizar la solución a un problema particular, prestando atención a que las técnicas usadas pueda ser útiles en problemas que requieran otra infraestructura.

1.2. Metas

Existe una gran variedad de técnicas de optimización paralelizables, varios métodos de paralelización, cada uno con ventajas y desventajas, y aún más variedad de problemas que podrían beneficiarse de esas optimización. Por restricciones externas, en este trabajo se considera la heurística de Optimización por Enjambre de Partículas, paralelización por medio de GPUs y como problema de estudio la logística de cadenas de suministros.

Aún así, la meta principal de este trabajo es que el proceso documentado aquí sea útil para replicar el resultado en problemas similares.

Las contribuciones de este trabajo son:

1.3. Objetivo

El objetivo general de este trabajo consiste en analizar que tanto se puede mejorar el rendimiento de una heurística de optimización dada por medio de paralelización y analizar su desempeño en un problema en particular.

Los objetivos particulares consisten en

- Implementar secuencialmente la heurística

- Determinar los puntos de paralelización de la técnica existente
- Implementar la heurística de forma paralela
- Analizar las mejoras en tiempos de la implementación paralela
- Analizar las mejoras en calidad de la implementación paralela

1.4. Organización

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

En capítulo 2 se presentan los diferentes tipos de paralelismo, así como sus ventajas, restricciones y problemáticas. Además se estudian algunas técnicas comunes para paralelizar algoritmos secuenciales. En capítulo 3 se presenta la heurística a paralelizar y se usan los criterios presentados en capítulo 2 para proponer una versión paralelizada. En capítulo 4, se presenta el problema de cadenas de suministros. Se incluye su formalización como problema de optimización, se mencionan las dificultades para obtener soluciones óptimas y las técnicas usadas para resolverlo, incluyendo BPSO. En capítulo 5 se realiza una comparación cualitativa entre los recursos y la calidad de las soluciones tanto de la solución secuencial como la solución paralela. Se concluye el trabajo con el capítulo 6, donde se exponen los comentarios finales y se sugiere el trabajo a realizar en próximas investigaciones.

Se anexan además varios apéndices. En el apéndice A se describe la arquitectura de las tarjetas CUDA de Nvidia, así como conceptos básicos de CUDA y `nvcc`. El apéndice B es una guía de alto nivel del lenguaje Julia, así como algunas bibliotecas útiles. En el apéndice C se da un breve repaso de programas lineales y del método Simplex.

Capítulo 2

Paralelismo

En este capítulo se van a explorar diferentes clasificaciones de arquitecturas paralelas y sus peculiaridades, además de dar algunos criterios para clasificar los problemas y decidir la mejor manera de crear una solución paralela a un problema en particular.

2.1. 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*

- *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.

2.1.1. Sistemas *SISD*

Contrario a su nombres, estos modelos pueden tener ciertos mecanismos paralelos. Se suelen implementar como una optimizaciones a nivel de hardware, por lo no se tiene el control a nivel del software. Por esto no se consideran sistemas paralelos en la taxonomía de Flynn.

Sin embargo, paralelizar secciones de código previamente secuenciales, altera el orden de ejecución del programa. Esto se tiene que tomar en cuenta al diseñar sistemas paralelos, se tenga o no el control sobre el mecanismo.

Instruction Level Paralelism

ILP consiste en paralelizar la ejecución de las instrucciones de un mismo programa.

- *pipelining*

  Consiste en dividir una instrucción en pasos atómicos. Por ejemplo sacar los datos de memoria, revisar el tipo de operación, aplicar la operación, guardar los datos resultantes, entre otros.

  Cada paso se realiza en una sección de la unidad de control. Cuando una sección termina de procesar su paso, empieza a procesar la siguiente instrucción, independientemente si la instrucción anterior se haya terminado de ejecutar, simulando una

línea de ensamblaje. Si el procesamiento se divide en *k* pasos, se podrían procesar hasta *k* instrucciones a la vez.

- *multiple issue*

  La unidad de control posee varias copias de las mismas secciones (i.e. *ALUs*) que pueden ser usadas en paralelo. Para coordinar esas secciones, es necesario tener algún tipo de planificación. Esto se suele hacer como una optimización al compilar un programa o durante la ejecución a nivel de hardware.

En todos casos, no siempre es posible saber la secuencia de instrucciones, pues la siguiente instrucción puede depender del resultado de la actual. Para esto usan métodos de *especulación*. Basado en análisis estadísticos, durante la planificación se adivina la siguiente instrucción. En caso se ser errónea, se desecha el resultado y se calcula la versión correcta.

Thread Level Paralelism

Una alternativa a especular es usar una abstracción más alta. *TLP* mantiene en ejecución diferentes programas, alternando qué programa usa el procesador.

El mecanismo para cambiar el programa ejecutándose se denomina *hardware parallelism* (paralelismo de hardware). Para que se note una mejora en el desempeño, el costo fijo de cambiar de proceso debe ser menor a la ganancia dada por el paralelismo, que incrementa entre mayor sean los datos a procesar. Esto hace que estas optimizaciones no sean útiles en problemas pequeños.

2.1.2. Sistemas *SIMD*

Tienen una única unidad de control con varias unidades aritmético lógicas (*ALU*). Al recibir la instrucción, la unidad de control reparte los datos para que se aplique la operación en paralelo. 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.

No suelen requerir modificaciones sustanciales a código existente, por lo que su uso se puede automatizar, y suelen resultar en mejoras notables. Por otra parte, tienen una variedad limitada de operaciones disponibles, usar datos irregulares provoca que unidades desperdicien ciclos de trabajo, y no es una arquitectura escalable.

En general, cada núcleo de un *CPU* cuenta con instrucciones que permiten este tipo de operaciones. Sin embargo, los *CPUs* no se consideran sistemas *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 *paralelismo de hardware*. Los *GPUs*, tampoco son exclusivamente un sistema *SIMD*, ya que suelen tener decenas de núcleos.

2.1.3. Sistemas *MIMD*

Cada unidad de procesamiento es completamente independiente. Las operaciones no son uniformes ni síncronas. Para coordinar sus acciones necesitan, las unidades necesitan comunicarse. Los dos métodos más comunes son usar bloques de memoria compartida, que se usa implícitamente para comunicar las acciones a otros procesos, o explícitamente mandar mensajes por medio de un sistema de red.

Sistemas de memoria compartida

Se suele compartir memoria de dos maneras. Se puede tener un bloque de memoria global y cada unidad tiene acceso a ella. Esto es simple pero potencialmente lento. Este tipo se sistemas se denominan *UMA* (Universal Memory Access). Una alternativa es que cada unidad tenga un bloque de memoria, y las demás unidades tenga acceso a ese bloque. Esto hace que la memoria *local* de cada proceso sea de rápido acceso, aunque acceder a la memora de otra unidad sea más costoso. Esto se denomina *NUMA* (Non Universal Memory Access).

- *UMA* (Universal Memory Access)

  Los sistemas *UMA* son más predecibles, por lo que son más sencillos de programar. Pero no son escalables, pues hay un límite para el tamaño de la memoria global.

  Sistemas como estos son los *CPUs* multi-núcleo. Cada núcleo es unidad independiente que comparten la *RAM* o algún espacio de disco duro.

- *NUMA* (Non Universal Memory Access)

  Los sistemas *NUMA* son más baratos y escalables, pues para añadir más memoria basta con añadir una nueva unidad. Son menos predecibles al tener tiempo variable de acceso a memoria.

En ambos casos, el mecanismo para compartir la memoria puede ser un cuello de botella.

La topología más sencilla para esto se conoce como un *bus*. Todas las unidades están conectadas a un canal de comunicación principal. Solo una unidad puede acceder a la memoria compartida al mismo tiempo. Esto es restrictivo pero barato. Para sistemas pequeños puede ser lo más eficiente. Claramente no es escalable, pues entre más unidades haya mayor la probabilidad de contención por en *bus* aumenta.

Para sistemas más grandes se suele usar un *crossbar*. Esto requiere *switches*. Cada unidad de procesamiento y de memoria cuenta con un *bus*. Y cada *bus* está conectado a los demás por medio de un *switch*, de tal forma que cada unidad de procesamiento solo esté conectada a una unidad de memoria a la vez. Así, solo hay contención si dos unidades quieren acceder al mismo bloque de memoria al mismo tiempo. Esto es eficiente, pero los *switches* agregan un costo fijo que puede ser demasiado para sistemas pequeños.

Independientemente de las conexiones, un problema en los sistemas compartidos es la *coherencia de caché*. El *caché* es memoria de alta velocidad dentro de las unidades de procesamiento. Sirve para almacenar temporalmente los datos que se están usando para evitar consultas a la memoria global. Es una optimización a nivel de *hardware*, por lo que no se tiene control sobre como se usa.

El problema en sistemas paralelos llega cuando varias unidades procesan los mismo datos en memoria. Al cada uno tener una copia de los datos en su *caché*, estos pueden quedar des-actualizados si otra unidad los modifica. El problema de mantener el *caché* de todas las unidades actualizado se denomina *coherencia de caché*.

Una técnica sencilla para sistemas basados en *buses* es *snooping*. Como todos los datos pasan por el *bus* común, se puede interceptar los cambios en memoria. Para esto, basta con que una unidad mande una señal cuando modifique un valor. De esta manera, las demás unidades sabrán que su copia local del valor ya no es válida.

Esta técnica no es escalable para sistemas que usen *switches*, generar demasiado tráfico. Una alternativa es *caché basado en directorios*. La idea es que las unidades mantengan una tabla distribuida donde indiquen que unidad posee que datos en su caché. Así, cuando una unidad modifique un valor, puede consultar esa tabla para solo enviar las notificaciones a las unidades que lo posean.

Sistemas distribuidos

El tipo más común de sistema distribuido son los *clusters* de computadoras. Estas son unidades independientes conectadas a través de una red. Las unidades pueden tener características de hardware muy diferentes, e incluso estar físicamente separadas. Por lo mismo, estos sistemas son más escalables. Aunque ser tan heterogéneos hace que sean menos predecibles.

A pesar de que no compartan memoria, usan sus conexiones para mandar mensajes y coordinar sus acciones. Hay dos tipos principales de conexiones. En las *conexiones directas*, cada *switch* está conectado a una unidad de procesamiento, a manera de proxy. Por lo que las conexiones se pueden pensar entre unidades de procesamiento directamente. Ejemplos de topologías de este tipo son *anillos*, *toros* o *hipercubos*. En las *conexiones indirectas*, los *switches* pueden estar conectados a unidades o a otros *switches*. Un ejemplo de esto es el *crossbar* mencionado anteriormente.

Al ser más robustos, escalables y baratos que los sistemas de memoria compartida, los sistemas distribuidos suelen usarse cuando la cantidad de datos a procesar es masiva. A pesa de su utilidad, en este trabajo se profundiza más en los sistemas de memoria compartida, al ser la infraestructura disponible. Así que no se profundizará más al respecto de los sistemas distribuidos.

2.2. Programas paralelos

La paralelización es una optimización. Así que normalmente se quiere paralelizar un programa ya existente. Esto requiere identificar secciones de código donde la carga de trabajo puede ser dividida entre varias unidades. Pero al dividir el trabajo, cada unidad queda aislada. Si parte de la ejecución depende del trabajo de otra unidad, es necesario que haya comunicación. En sistemas de memoria compartida, la comunicación es implícita al modificar secciones compartida de la memoria. Naturalmente, esta comunicación genera problemas únicos.

Uno de ellos es una *condición de carrera*. Esto sucede cuando dos unidades modifican el valor de una variable compartida, y luego usan ese valor en computaciones posteriores. El valor final de la variable sería el que le asigne la última unidad en verificarla. Lamentablemente, debido a *ILP* y a *TLP*, el orden y la velocidad de ejecución de las instrucciones puede cambiar en cada ejecución. Esto significa que correr el mismo programa varias veces, con los mismo argumentos, daría resultados diferentes. Se denomina una *carrera* porque el resultado de la ejecución depende de que unidad ejecute más rápido su código.

Una solución a este problema son los *mutex*, o candados de exclusión mutua. Son mecanismos (pueden ser a nivel hardware o software) que solo permiten la ejecución de cierto bloque de código a una unidad a la vez. Estas secciones se denominan *secciones críticas*. Claramente, una sección crítica no se ejecuta en paralelo, así que es importante evitarlas, y cuando se usan, que sean lo más breves posibles. Una alternativa a esto son los *semáforos*. Tienen un funcionamiento similar a los *mutex*, pero pueden permitir más de un permiso a la vez, entre otras cosas. Otra solución a más alto nivel, común en bases de datos, son las *transacciones*. Durante una transacción, si alguna de las operaciones no se puede ejecutar, por haber encontrado un *mutex* o algún otro motivo, todas las operaciones se revierten. Esto evita que la memoria compartida quede en un estado no determinista.

Otro problema único de la paralelización es la *sincronización*. Esto consiste en que cada unidad espere en un punto dado hasta que todas las demás unidades lleguen a este punto. Esto requiere comunicación entre todas las unidades. La implementación se suele denominar una *barrera*, y puede lograrse con varios *semáforos*.

2.2.1. Métricas de desempeño

Al ser la paralelización una optimización, es muy importante medir cuál es la mejora real obtenida.

Incremento y Eficiencia

Si se divide el trabajo en p partes, en el mejor caso se tendría una mejora de p veces la velocidad inicial. Es decir $t_p = \frac{t_s}{p}$. Esto no suele pasar, ya que lidiar con los problemas de exclusión, comunicación y sincronización suelen ser costosos. Así que despejando de la fracción anterior, se define el *speedup* como la ecuación (2.1)

$$s = \frac{t_s}{t_p} \tag{2.1}$$

Así, el mejor caso estaría acotado por $s = p$. Entre más incremente p, se esperaría que s se acerque más y más a p. Es decir

$$\lim_{p \to \infty} \frac{s}{p} = \frac{t_s}{p t_p} \to 1$$

Este valor $e = \frac{s}{p}$ se suele denominar *eficiencia*. Hay que notar que estos valores no solo dependen de p. Es normal observar que con mayor cantidad de datos, el *speedup* y la *eficiencia* incrementen. Esto se debe a que la ganancia por paralelización incrementa, mientras que el costo de comunicación, exclusión y sincronización se mantienen relativamente constantes.

Ley de Amdahl

Una observación importante es que solo se podrá obtener una mejora en las secciones del programa que puedan ser paralelizables. Esto se puede expresar con la ecuación (2.2)

$$t_p = r t_s + \frac{1-r}{p} t_s = t_s (r + \frac{1-r}{p}) \tag{2.2}$$

donde r es el porcentaje del código que no es paralelizable. Con esta nueva definición, la cota para la definición de *speedup* se puede reescribir como

$$s = \frac{t_s}{t_p p} = \frac{t_s}{t_s (r + \frac{1-r}{p}} = \frac{1}{r + \frac{1-r}{p}}$$

Esto significa, que sin importar cuantas unidades extra se añadan, nunca se obtendrá una mejora mayor a $\frac{1}{r}$. Esta ley toma su nombre de la primera persona que la describió formalmente, Gene Amdahl.

Escalabilidad

En secciones anteriores se mencionó que el costo para comunicación, exclusión y sincronización suele ser constante relativamente al número de unidades. Esto no es del todo cierto. Cuando un sistema tiene esta propiedad se dice que es *fuertemente escalable*. Formalmente, si un programa paralelo tiene una eficiencia e usando p unidades, esta eficiencia no disminuirá al incrementar p, sin importar el tamaño de la entrada. Una variante de esta propiedad es un sistema *débilmente escalable*. En este caso, al incrementar las unidades por un factor de k, la eficiencia se puede mantener únicamente incrementando el tamaño de la entrada por algún factor a lo más lineal sobre k.

2.2.2. Consejos generales

Paralelizar una solución depende de la solución en particular. No hay una proceso metódico para lograrlo. Aun así, existen diferentes técnicas. Una de ellas es la *metodología de Foster*. Esta indica que se puede dividir el proceso en cuatro etapas.

1. Partir

   Consiste en repartir las tareas. En otras palabras, identificar las secciones paralelizable.

2. Comunicar

   Identificar la información a comunicar entre las unidades, y los puntos donde es necesario que lo hagan.

3. Aglomerar

   Identificar tareas extra donde es necesario unir resultados en un resultado común y el mecanismo a usar para ello.

4. Mapear

   Asignar las tareas de aglomeración del paso anterior.

En cada paso, hay que procurar balancear la carga de trabajo y minimizar la comunicación. Esto para tener ejecuciones lo más homogéneas y con menos desperdicio de recursos posible.

La paralelización es una optimización que introduce un costo fijo extra. Como tal, puede ser que un programa perfectamente paralelizable no genera ganancias notables. Hay que determinar si la ganancia a obtener vale la pena el esfuerzo a realizar. A priori esto puede no ser tan claro. Pero Pancake [Pan96] propone algunas reglas que a grandes rasgos permitan determinar si vale la pena dar una solución paralela a un problema.

- Usar el desempeño de la versión secuencial como base para estimar las ganancias.
- Por la ley de Amdahl, no vale la pena paralelizar un programa con una fracción paralelizable menor a 0.95.
- Dependiendo del problema en particular, hay que estimar como aumentar el tamaño de la entrada afecta la eficiencia.
- La mejora real siempre será peor a la mejora teórica.
- Si la naturaleza del problema no es muy compatible con la infraestructura disponible, probablemente no valga la pena.
- Un problema que requiera poca comunicación tendrá un desempeño decente en cualquier sistema. Un sistema con media o alta comunicación probablemente solo sea decente en un sistema *SIMD*.

Además de estos consejos generales, provee algunos consejos más específicos, dependiendo del tipo de problema. Se dividen en cuatro tipos

- Paralelismo perfecto

  Únicamente requieren dividir la carga de trabajo. Ningún intercambio de información entre unidades es necesario. Estos problemas son fáciles de paralelizar y suelen obtener ganancias considerables.

  Tendrían resultados decentes en sistemas *MIMD*, y si se pueden adaptar las dimensiones de los datos, también en sistemas *SIMD*.

- Paralelismo de *pipelining*

  Hay que aplicar cierto procesamiento a todos los datos, y diferentes etapas del procesamiento requieren, potencialmente todos, los datos de etapas anteriores. Al igual que el *ILP*, la manera más sencilla de paralelizar esto es asignar una etapa por unidad y dividir los datos en grupos. Cada unidad procesa un a la vez grupo, y cuando termine manda la información a la siguiente unidad.

  Esto puede alcanzar un rendimiento decente en sistemas *MIMD*, siempre y cuando sea posible distribuir la carga de manera efectiva, para que cada unidad tarda aproximadamente el mismo tiempo en procesar un grupo. De otra manera, el retraso de una unidad podría alentar demasiado todo el programa.

  También sería posible adaptarlo a un sistema distribuido, siempre y cuando la comunicación sea suficientemente rápida.

- Paralelismo síncrono

  Hay que aplicar cierto procesamiento a los datos, pero cada paso debe aplicarse simultáneamente a todos los datos, por lo que no se pueden dividir en grupos. En este caso, cada unidad aplicar todos los pasos a una sección de los datos. Para esto, las unidades se tienen que sincronizar y comunicar datos relevantes.

  Un problema podría ser que diferentes regiones de los datos requieran una intensidad diferente dependiendo de sus valores. Si no se puede balancear esta carga, el costo fijo de la paralelizar sería demasiado elevado.

  Si se pueden adaptar las operaciones vectores, tendría un desempeño decente en un sistema *SIMD*. También en un sistema distribuido, siempre y cuando no se requiera mucha comunicación. Se desempeñaría pobremente en cualquier otro tipo de infraestructura.

- Paralelismo vagamente síncrono

  Similar al paralelismo síncrono, pero los datos que necesitan ser procesados en el paso siguiente dependen totalmente del paso anterior. Y no es posible predecir estas dependencias a priori. Por lo que en general estos problemas requieren mucha comunicación y no es posible balancear la carga de las unidades.

  Probablemente no valga la pena paralelizarlo, al menos que la comunicación entre diferentes unidades sea mínima.

  Probablemente solo desempeñe decentemente en un sistema *MIMD*. Si hay poca comunicación y una rápida conexión, probablemente también en un sistema distribuido.

2.3. Resumen

Hay diferentes tipos de arquitectura para sistemas paralelos. Tienen características particulares que las hacen más aptas para ciertos tipos de problemas. Determinar cuál es la mejor arquitectura para un problema dado no es sencillo. Requiere conocimiento profundo del problema en particular. Además de tomar muchas medidas del desempeño actual, y usarlas para aproximar las posibles al paralelizar.

Independientemente de la mejora en desempeño, hay que tomar en cuenta que paralelizar un programa introduce nuevos tipos de problemas. Esto podría resultar en más tiempo de desarrollo y frustraciones al programar.

A pesar de haber tantas complicaciones para decidir si vale la pena paralelizar un problema dada la infraestructura disponible, hay una gran cantidad de consejos empíricos que se pueden tomar en cuenta para facilitar la decisión.

Capítulo 3

Paralelización de *B-PSO*

3.1. Optimización por Enjambre de Partículas

La optimización por enjambre de partículas (*PSO* por sus siglas en inglés) es una técnica de optimización para funciones contínuas fue desarrollada por Kennedy y Eberhart en 1995[Yan14]. Es parte de una familia más extensa de algoritmos de optimización llamados *inteligencia de enjambre*. Estos intentan imitar los cardúmenes de peces o las parvadas de aves en su manera de movilizarse para evitar depredadores.

En PSO, se tiene una función objetivo, y un conjunto de partículas. Cada partícula tiene una posición, que representa una posible solución para la función objetivo, y una velocidad. En cada paso, cada partícula tiene una tendencia a moverse en dirección de la mejor partícula global y de su mejor posición individual histórica, además de tener cierto movimiento aleatorio.

Puntualmente, si en un momento t, una partícula tiene posición x^t y velocidad v^t, su velocidad al momento $t+1$ está dada por la ecuación (3.1)

$$v^{t+1} = v^t + \epsilon_0 c_1 (x^t - x^\star) + \epsilon_1 c_2 (x^t - \mathbf{x}^\star) \tag{3.1}$$

donde $x^\star$ es la mejor posición alcanzada por$x$, $\mathbf{x}^\star$ es la mejor posición global actual, c_i son constantes para determinar cuando influencia la mejor solución personal y global y $\epsilon_i \in [0,1]^n$ son vectores aleatorios.

Con la velocidad actualizada, se puede actualizar la posición, dada por la ecuación (3.2)

$$x^{t+1} = x^t + v^{t+1} \tag{3.2}$$

El proceso de optimización consta de rondas. En cada una, las partículas se mueven. Las rondas terminan cuando se cumple algún criterio de optimización. Este puede ser un número fijo de rondas, un umbral conocido de valores deseados, o que las soluciones obtenidas ya no mejores después de cierta cantidad de rondas. Este proceso está descrito en el algoritmo 1

3.2. PSO Binario

En PSO la velocidad y posición deben tomar valores reales, por lo la técnica solo se puede usar para funciones continuas. Pero en la práctica muchos problemas de optimización están definidos sobre espacios discretos. Para lidiar con esto, los autores de la heurística posteriormente propusieron una variante del algoritmo que funciona sobre espacios binarios.

Para esto, usan la función sigmoide (ecuación (3.3)) para discretizar valores.

Algoritmo 1 Optimización por Enjambre del Partículas (PSO)

Entrada f función objetivo, n número de partículas, c_1, c_2
Salida Mejor posición encontrada $\mathbf{x}^\star$

```
1: funcion pso(f, n, c_1, c_2)
2:     Iniciar n soluciones X para f de manera uniforme
3:     Iniciar mejores soluciones por partícula X* ← X
4:     Iniciar velocidades V en cero
5:     Encontrar mejoro solución inicial x*
6:     mientras No se cumpla el criterio hacer
7:         para todo i ∈ [1..n] hacer
8:             Actualizar V[i] de acuerdo a ecuación (3.1)
9:             Mover X[i] de acuerdo a ecuación (3.2)
10:            Actualizar mejor posición X*[i]
11:        fin para
12:        Actualizar la mejor posición global x*
13:    fin mientras
14:    devolver x*
15: fin funcion
```

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.3}$$

Luego, para reemplazar el movimiento para la posición x_i de una partícula, se usa la ecuación (3.4).

$$x_i^{t+1} = \begin{cases} 1, r \leq \sigma(v^{t+1}) \\ 0 \end{cases} \tag{3.4}$$

Donde $r \in [0, 1]$ en un número aleatorio. El resto del procedimiento es el mismo que en el algoritmo 1, solo reemplazando la ecuación (3.2) por ecuación (3.4).

3.3. Paralelización

Capítulo 4

Cadenas de Suministros

La *logística* de un sistema consiste en satisfacer ciertas demandas, de materias primas, de tiempo, bajo ciertas restricciones, normalmente presupuesto, mientras se intenta optimizar una medida de desempeño, normalmente el costo de materiales y transporte.

En el ámbito industrial, la logística se denomina *cadena de suministros*. Esta tiene dos etapas, la *cadena de producción* y la *cadena de distribución*[GLM04].

4.1. Formalización

Su estructura depende del caso en particular, pero siguiendo el modelo simplificado planteado por[Lic16], se considera que la cadena de producción tiene entrada fija de proveedores y plantas de ensamblaje. A su vez, la cadena de distribución tiene centros de distribución, proveídos por las plantas de ensamblaje, y puntos de venta. Cada instalación tiene un costo de operación fijo. El transporte entre cada instalación tiene un costo unitario. Cada instalación tiene una capacidad de producción máxima, y los puntos de venta tiene una demanda. Formalmente el problema tiene

- Proveedores S, plantas de ensamblaje P, centros de distribución D y puntos de venta C. Las instalaciones se denominan $I = S \cup P \cup D \cup C$.
- Matriz de costo unitario $U \in \mathbb{M}_{|I|\times|I|}$, donde U_{ij} representa el costo de mover una unida de productos de la instalación i a la instalación j
- Cada instalación tiene un costo fijo $F : I \to \mathbb{R}^+$.
- Cada instalación tiene una capacidad (o demanda) fija $W : I \to \mathbb{R}^+$.
- P_{max} y D_{max} para limitar la cantidad de plantas y centros de distribución a abrir.

El objetivo de la logística es encontrar como repartir la carga entre las plantas de producción y en los centros de distribución para respetar las restricciones y minimizando el costo. Formalmente

- Determinar $O : I \to \{0, 1\}$ que indique que instalaciones hay que abrir. Tanto los proveedores como los puntos de venta tiene que estar abiertos siempre.
- Determinar matriz de cargas X, donde X_{ij} es la cantidad de productos que va a circular entre la instalación i y j.

4.2. Programación lineal

Dado una O fija, encontrar X se puede formular como un problema de programación lineal, por lo que se pueden encontrar los valores óptimos en tiempo polinomial. La formulación sería minimizar

$$z(O) = \sum_{i,j \in I^2} O(i)O(j)X_{i,j}U_{i,j} + \sum_{i \in P \cup D} O(i)F(i) \text{ costo fijo + costo unitario} \tag{4.1}$$

sujeto a

$$\sum_{j \in I} X_{i,j} \leq W(i), \forall i \in I \text{ respetar capacidades}$$
$$\sum_{i \in I} X_{i,c} \geq W(c), \forall c \in C \text{ satisfacer demandas}$$
$$\sum_{i \in I} X_{i,j} = \sum_{k \in I} X_{j,k}, \forall j \in P \cup D \text{ flujo constante}$$
$$\sum_{p \in P} O(p) \leq P_{max}, \sum_{d \in D} O(d) \leq D_{max} \text{ respetar máximo de instalaciones}$$

Luego, para resolver este problema usando el método Simplex (ver) hay que pasar esta formulación a un formato equacional.

Capítulo 5

Resultados

Capítulo 6

Conclusiones y Trabajo Futuro

Apéndice A

Arquitectura de CUDA y CUDA-C

A.1. Motivaciones

A.2. CPUs

A.3. GPUs

Kernels Indices Huesped y devices

A.4. TPUs

A.5. Resumen

Apéndice B

Especificación de Julia

B.1. Sintaxis

B.2. Estructuras básicas

B.3. Sistema de tipos

Apéndice C

Programación Lineal

Un problema de optimización lineal requiere maximizar (o minimizar) una función lineal z bajo m restricciones lineales.

$$a_j \leq k_j \tag{C.1}$$
$$a_j \geq k_j \tag{C.2}$$
$$a_j = k_j \tag{C.3}$$

donde tanto z como a_i son combinaciones lineales de un conjunto $x_i{}_{i=1}^{k}$ de variables y k_j son números reales. Esto puede ser restrictivo en varios casos, pero aún así existen una gran cantidad de problemas que pueden ser formulados de esta forma. Usando la linealidad de la función objetivo y de las restricciones, se han desarrollado varios métodos para encontrar soluciones óptimas en tiempo razonable.

C.1. Forma ecuacional

Un problema está en forma ecuacional si todas sus restricciones (excepto las de no negativiad) son ecuaciones (i.e. $a_j = k_j$) con $k_j \geq 0$ y todas las variables son no negativas. Para todo problema, es posible construir un problema en forma ecuacional con la misma solución óptima. Para hacer esto se puede hacer lo siguiente

- Si $k_j < 0$, se multiplica la restricción por -1 y se usa alguno de los casos siguientes.
- Si se tiene una variable x_j que puede tener valores positivos y nagativos, se usan dos nuevas variables para obtener la ecuación $x_j = x_j^+ - x_j^-$, con $x_j^-, x_j^+ \geq 0$.
- Para las desigualdades de la forma $a_j \leq k_j$, se introduce una variable s_j de *sobra* (*slack* en inglés). La ecuación equivalente sería $a_j + s_j = k_j$ y además se agrega la restricción $s_j \geq 0$
- Para las desigualdades $a_j \geq k_j$, se hace algo análogo al caso anterior. La ecuación resultante es $a_j - s_j = k_j$ y $s_j \geq 0$.

Índice alfabético

Referencias

[GLM04] Gianpaolo Ghiani, Gilbert Laporte y Roberto Musmanno. *Introduction to Logistics Systems Planning and Control*. USA: Halsted Press, 2004. isbn: 0470849177.

[Lic16] Diana Xochitl Canales Licona. «Un Algoritmo de Cúmulo de Partículas para la Optimización del Costo de Operación de una Red de Cadena de Suministro de Múltiples Etapas mediante una Estrategia de Distribución Física». Maestro en Ciencias en Ingeniería Industrial. Ciudad del Conocimiento, Mineral de la Reforma, Hidalgo, México: Universidad Autónoma del Estado de Hidalgo, jul. de 2016.

[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.

[Yan14] Xin-She Yang. «Chapter 7 - Particle Swarm Optimization». En: *Nature-Inspired Optimization Algorithms*. Ed. por Xin-She Yang. Oxford: Elsevier, 2014, págs. 99-110. isbn: 978-0-12-416743-8. doi: https://doi.org/10.1016/B978-0-12-416743-8.00007-5. url: https://www.sciencedirect.com/science/article/pii/B9780124167438000075.