DR. GABRIEL LEON PAREDES

gleon@ups.edu.ec

www.linkedin.com/in/gabrielleonp

Cloud Computing, Smart Cities & High-Performance Computing

Cuenca, Ecuador

COMPUTO PARALELO

Modelos de programación paralela

- Los modelos de programación en paralelo son una abstracción de las arquitecturas de hardware.
  - Estos modelos de programación no son específicos para una computadora en particular.
  - Pueden ser implementados (en teoría) en cualquier arquitectura.
  - Los modelos de programación son de alto nivel y representan la manera en la que el software debe ser implementado para realizar operaciones en paralelo.
  - Cada modelo de programación tiene su propia manera de compartir, acceder y dividir los trabajos entre los procesadores.

Modelos de programación paralela

- ¿Cual es el mejor modelo de programación?
  - El modelo de programación depende del problema, no existe uno mejor que otro.
- Los modelos de programación mas utilizados son:
  - Modelo de memoria compartida
  - Modelo de multihilos
  - Modelo de memoria distribuida/paso de mensajes
  - Modelo de datos paralelos

Modelo de memoria compartida

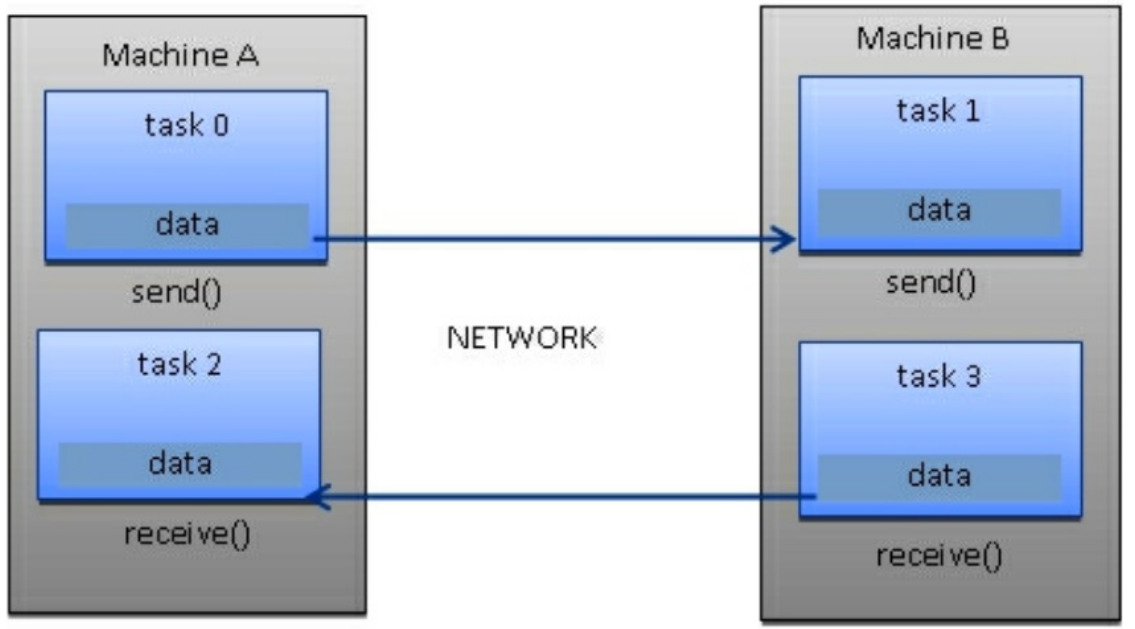
- Las tareas comparten una sola área de memoria, donde el acceso (lectura/escritura) para compartir los recursos es asíncrono.
- Existen mecanismos para controlar el acceso a la memoria compartida:
  - Bloqueos
  - Semáforos
- La ventaja de este modelo es que los programadores no deben especificar como se realiza la comunicación entre las tareas.
- Una importante desventaja en términos de rendimiento, es que resulta mas difícil entender y manejar los datos locales.
  - Esto se refiere a mantener los datos locales en el procesador que funciona para conservar el acceso a la memoria, las actualizaciones de caché y el tráfico del bus que ocurre cuando varios procesadores usan los mismos datos.

Modelo multihilos

- Un proceso puede tener varios flujos de ejecución, por ejemplo, una parte serial se crea y a continuación, una serie de tareas creadas pueden ser ejecutadas en paralelo.
  - Este modelo de programación es utilizado generalmente en arquitecturas de memoria compartida.
  - Es muy importante manejar la sincronización entre los hilos, ya que operan sobre la misma memoria compartida.
  - El programador debe prevenir que los hilos actualicen los mismo espacios de memoria a la vez.
  - La generación actual de CPUs soportan multihilos tanto en software como en hardware.
    - Los subprocesos POSIX (abreviatura de interfaz de sistema operativo portátil) son ejemplos clásicos de la implementación de subprocesos múltiples en software.
    - La tecnología Hyper-Threading de Intel implementa subprocesos múltiples en hardware al cambiar entre dos subprocesos cuando uno está esperando E/S.

Modelo de paso de mensajes

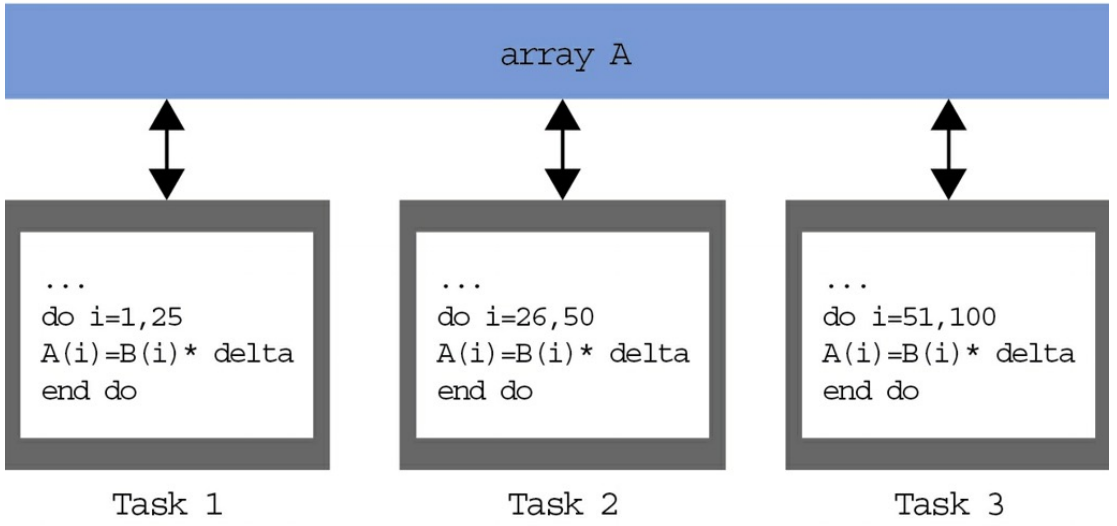
- Este modelo, es aplicado generalmente cuando cada procesador tiene su propia memoria (sistemas de memoria distribuida).
  - Varias tareas pueden estar en una misma o varias computadoras.
  - El programador es responsable del paralelismo de datos y el intercambio de información que ocurre en el paso de mensajes.
  - Requiere del uso de librerías de software especiales.
  - Existen varias implementaciones desde los 80s, por lo que el modelo a mediados de los 90s se estandarizo y se lo conoce como Interfaz de Paso de Mensajes (MPI).
  - El modelo MPI esta diseñado para memorias distribuidas, pero al ser un modelo de programación paralela y multiplataforma se lo puede usar en maquinas con memoria compartida.



Message passing paradigm model

Modelo de datos paralelos

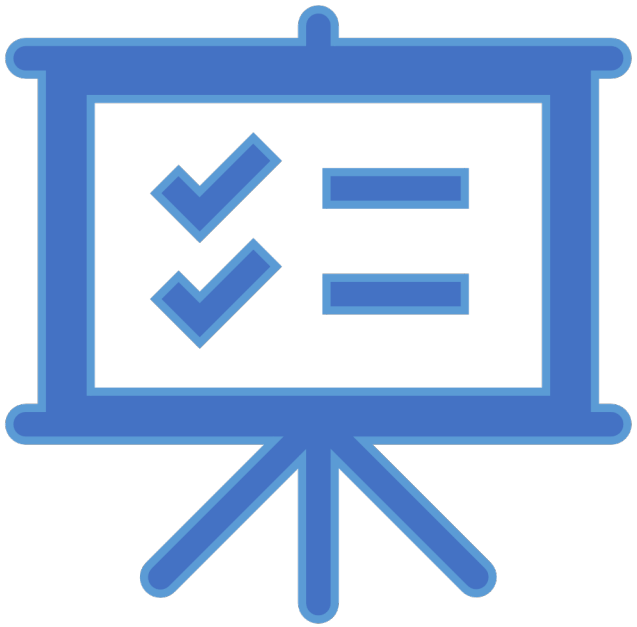
- En este modelo se tiene mas tareas que se ejecutan sobre la mismos estructura de datos, pero cada tarea opera sobre distintas porciones de datos.
- En la arquitectura de memoria compartida, todas las tareas tiene acceso a los datos a través de arquitecturas de memoria compartida y distribuida, en donde la estructura de datos es dividida y almacenada en la memoria local de cada tarea.
- Para implementar este modelo el programador debe desarrollar aplicaciones que especifiquen la distribución y alineación de los datos
- Por ejemplo, la generación actual de GPUs es altamente operacional solo si los datos (Tarea 1, Tarea 2, Tarea 3) esta alineados



The data-parallel paradigm model

Diseñando un programa paralelo

- El diccionario de la IEEE define un algoritmo como “Un conjunto prescrito de reglas o procesos bien definidos para la solución de un problema en un número finito de pasos”
  - Las tareas o procesos de un algoritmos son interdependientes en general.
  - Algunas tareas pueden ejecutarse simultáneamente en paralelo y otras deben ejecutarse secuencialmente una tras otra.
- El diseño de algoritmos que explotan el paralelismo se basa en una serie de operaciones, que deben llevarse a cabo para que el programa realice el trabajo correctamente sin producir resultados parciales o erróneos.

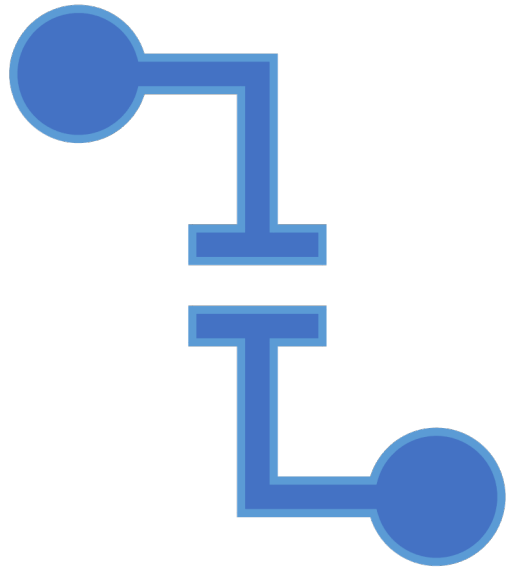


Diseñando un programa paralelo

- Las operaciones macro que deben llevarse a cabo para una correcta paralelización de un algoritmo son las siguientes:
    - Descomposición de tareas
    - Asignación de tareas
    - Aglomeración
    - Mapeo

Descomposición de tareas

- En esta primera fase, el programa de software se divide en tareas o un conjunto de instrucciones que luego se pueden ejecutar en diferentes procesadores para implementar el paralelismo. Para realizar esta subdivisión, se utilizan dos métodos:
    - **Descomposición del dominio**: aquí, los datos de los problemas se descomponen. La aplicación es común a todos los procesadores que trabajan en diferentes porciones de datos. Esta metodología se utiliza cuando tenemos una gran cantidad de datos que deben procesarse.
    - **Descomposición funcional**: en este caso, el problema se divide en tareas, donde cada tarea realizará una operación particular en todos los datos disponibles.



Asignación de tareas

- En este paso, se especifica el mecanismo por el cual las tareas se distribuirán entre los diversos procesos. Esta fase es muy importante porque establece la distribución de la carga de trabajo entre los distintos procesadores.
- El equilibrio de carga es crucial aquí; de hecho, todos los procesadores deben funcionar con continuidad, evitando estar en estado inactivo durante mucho tiempo.
- Para realizar esto, el codificador debe tomar en cuenta la posible heterogeneidad del sistema que intenta asignar más tareas a los procesadores con mejor rendimiento.
- Finalmente, para una mayor eficiencia de la paralelización, es necesario limitar la comunicación tanto como sea posible entre los procesadores, ya que a menudo son la fuente de la desaceleración y el consumo de recursos.

Aglomeración

- La aglomeración es el proceso de combinar tareas más pequeñas con tareas más grandes para mejorar el rendimiento. Si las dos etapas anteriores del proceso de diseño dividieron el problema en una serie de tareas que exceden en gran medida la cantidad de procesadores disponibles, y si la computadora no está diseñada específicamente para manejar una gran cantidad de pequeñas tareas, entonces el diseño puede resultar altamente ineficiente.
    - Algunas arquitecturas, como GPU, maneja esta fina y, de hecho, beneficiosa ejecución de millones, o incluso miles de millones de tareas

- Comúnmente, esto se debe a que las tareas deben comunicarse al procesador o al subproceso para que calculen dicha tarea. La mayoría de las comunicaciones tienen costos que son desproporcionados a la cantidad de datos transferidos, pero también incurren en un costo fijo para cada operación de comunicación (como la latencia, que es inherente a la configuración de una conexión TCP). Si las tareas son demasiado pequeñas, entonces este costo fijo puede hacer que el diseño sea ineficiente.

Mapeo

- En la etapa de mapeo, especificamos dónde se ejecutará cada tarea. El objetivo es minimizar el tiempo total de ejecución. Aquí, a menudo debe hacerse concesiones, ya que las dos estrategias principales a menudo entran en conflicto entre sí:
  - Las tareas que se comunican con frecuencia deben colocarse en el mismo procesador para aumentar la localidad.
  - Las tareas que se pueden ejecutar simultáneamente deben colocarse en diferentes procesadores para mejorar la concurrencia.
- Esto se conoce como el problema de mapeo, y se sabe que es NP-completo. Como tal, no existen soluciones en tiempo polinomial para el problema. Para tareas de igual tamaño y tareas con patrones de comunicación fácilmente identificables, el mapeo es sencillo (también podemos realizar aglomeración aquí para combinar tareas que se mapean al mismo procesador).
- Sin embargo, si las tareas tienen patrones de comunicación que son difíciles de predecir o la cantidad de trabajo varía según la tarea, entonces es difícil diseñar un esquema de mapeo y aglomeración eficiente.
- Para este tipo de problemas, los algoritmos de balanceo de carga se pueden usar para identificar estrategias de aglomeración y mapeo durante tiempo de ejecución. Los problemas más difíciles son aquellos en los que la cantidad de comunicación o el número de tareas cambian durante la ejecución del programa. Para este tipo de problemas, se pueden utilizar algoritmos de balanceo de carga dinámico, que se ejecutan periódicamente durante la ejecución.

Mapeo dinámico

- Existen numerosos algoritmos de balanceo de carga para una variedad de problemas:
  - Algoritmos globales: estos requieren un conocimiento global del cálculo que se realiza, lo que a menudo agrega mucha sobrecarga.
  - Algoritmos locales: estos se basan solo en información que es local para la tarea en cuestión, lo que reduce la sobrecarga en comparación con los algoritmos globales, pero generalmente son peores para encontrar una aglomeración y mapeo óptimos.
- Sin embargo, la sobrecarga reducida puede disminuir el tiempo de ejecución, a pesar de que la asignación es peor por sí misma.
- Si las tareas rara vez se comunican más allá del inicio y el final de la ejecución, entonces un algoritmo de programación de tareas es utilizado, el cuál asigna las tareas a los procesadores libres.
  - En un algoritmo de programación de tareas, se mantiene un pool de tareas, en donde cada una será tomad por un trabajador (worker).

Enfoques del mapeo dinámico

- Hay tres enfoques comunes en este modelo:
  - **Administrador/trabajador:** este es el esquema básico de mapeo dinámico en el que todos los trabajadores se conectan a un administrador centralizado. El administrador envía repetidamente tareas a los trabajadores y recoge los resultados. Esta estrategia es probablemente la mejor para un número relativamente pequeño de procesadores. La estrategia básica se puede mejorar mediante la obtención de tareas por adelantado para que la comunicación y la computación se superpongan entre sí.
  - **Administrador jerárquico/ trabajador**: esta es la variante de un administrador/trabajador que tiene un diseño semi-distribuido. Los trabajadores se dividen en grupos, cada uno con su propio administrador. Estos administradores de grupo se comunican con el administrador central (y posiblemente también entre ellos), mientras que los trabajadores solicitan tareas de los administrador de grupo. Esto distribuye la carga entre varios administradores y, como tal, puede manejar un mayor número de procesadores si todos los trabajadores solicitan tareas del mismo administrador.
  - **Descentralizado**: en este esquema, todo está descentralizado. Cada procesador mantiene su propio grupo de tareas y se comunica con los otros procesadores para solicitar tareas. La forma en que los procesadores eligen otros procesadores para solicitar tareas varía y se determina en función del problema.

Evaluando el rendimiento de un programa paralelo

- El desarrollo de la programación en paralelo creó la necesidad de métricas y herramientas para evaluar el rendimiento de un algoritmo.
- El objetivo del computo en paralelo es el de resolver grandes problemas en un tiempo relativamente corto.
  - Para esto se toma en cuenta: hardware, grado de paralelismo del problema, y el modelo de programación paralela.
  - Para facilitar esto, se introdujo el análisis de conceptos básicos, que compara el algoritmo paralelo obtenido de la secuencia original.
- El rendimiento es alcanzado al analizar y cuantificar el número de hilos y/o el número de procesos usados. Los índices de rendimiento mas utilizados son: **aceleramiento, eficiencia y escalabilidad.**
- Las limitaciones de la computación paralela son introducidos por la ley de Ahmdal's.
- Para evaluar el grado de eficiencia de un algoritmo se introduce la ley Gustafson's.

Aceleramiento

- Mide el beneficio de resolver un problema en paralelo.
- Esta definido por la relación entre el tiempo necesario para resolver un problema en un solo elemento de procesamiento (Ts), con el tiempo necesario para resolver el mismo problema en (*p)* elementos de procesamiento idénticos (Tp).
- Entonces $s = \frac{Ts}{Tp}$

  S = *p* aceleramiento linear (aumenta con el número de procesadores – caso ideal)

  S < *p* aceleramiento real

  S > *p* aceleramiento superlineal

Eficiencia

- En un mundo ideal, un sistema en paralelo con *p* elementos de procesamiento puede darnos una aceleración igual a *p.*
  - *Esto no siempre sucede, ya que se pierde tiempo por comunicación o inactividad.*
- *La eficiencia es un métrica de rendimiento para estimar que tan "bien" son utilizados los procesadores para resolver una tarea, comparado con el esfuerzo desperdiciado en la comunicación y sincronización.*

- *Entonces* $E = \frac{S}{p} = \frac{Ts}{pTp}$
  - *E = 1, es un caso linear*
  - *E < 1, es un caso real*
  - *E<< 1 es un problema que esta paralelizado con poca eficiencia*

Escalabilidad

- Es definido como la habilidad de ser eficiente en una maquina paralela.
- Identifica la velocidad de ejecución en comparación con el numero de procesadores.
- Al incrementar el tamaño del problema y al mismo tiempo el numero de procesadores, no debe existir ninguna perdida en términos de rendimiento debe mantener la misma o mejorar.
- Se dice que un sistema es escalable para un determinado rango de procesadores [1..n], si la eficiencia E(n) del sistema se mantiene constante y en todo momento por encima de un factor 0.5.

Ley de Amdahl

- Es muy utilizada para diseñar procesadores y algoritmos en paralelo.
- La ley nos dice que la máxima aceleración que se puede alcanzar esta limitada por las partes seriales del programa.
  - Entonces $S = \frac{1}{1 - P}$
    - donde (1 – *P)* son los componentes seriales (no paralelizados) del programa y *P* el número de procesadores.

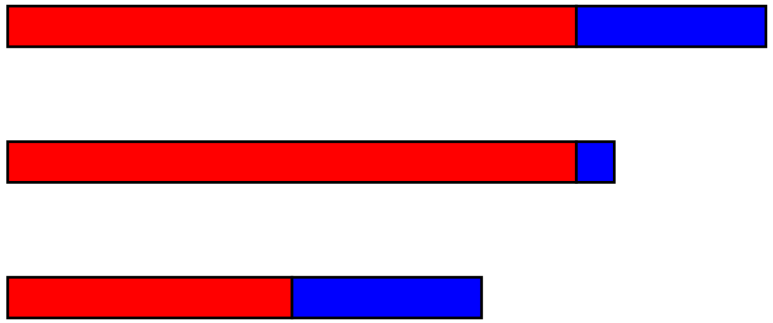
Ejemplo

Dos partes independientes **A** **B**

Proceso original

Haciendo **B** 5x más rápido

Haciendo **A** 2x más rápido



- El incremento de velocidad de un programa utilizando múltiples procesadores en computación paralela está limitada por la fracción secuencial del programa.
- Por ejemplo, si la porción 0.5 del programa es secuencial, el incremento de velocidad máximo teórico con computación paralela será de 2x (1/(1-0.5)) cuando N sea muy grande.

Ley de Gustafson

- *La ley de Gustafson establece lo siguiente:*
  - $S(P) = P + \propto (P - 1)$, donde *P,* es el número de procesadores; *S,* es el factor de aceleración; y $\propto$ es la fracción de no paralelizable de cualquier proceso.
- La ley de Gustafson contrasta con la ley de Amdahl, que, como describimos, supone que la carga de trabajo general de un programa no cambia con respecto al número de procesadores.
- De hecho, la ley de Gustafson sugiere que los programadores primero establezcan el tiempo permitido para resolver un problema en paralelo y luego, en función de eso (ese es el tiempo) para dimensionar el problema. Por lo tanto, cuanto más rápido sea el sistema paralelo, mayores serán los problemas que se pueden resolver durante el mismo período de tiempo.
- El efecto de la ley de Gustafson fue dirigir los objetivos de la investigación informática hacia la selección o reformulación de problemas de tal manera que la solución de un problema mayor aún sería posible en la misma cantidad de tiempo. Además, esta ley redefine el concepto de eficiencia como una necesidad de reducir al menos la parte secuencial de un programa, a pesar del aumento en la carga de trabajo.

Problema 1

- Sea un programa que posee un tiempo de ejecución de 100 unidades de tiempo (por ejemplo segundos). El 80% de su código es perfecta y absolutamente paralelizable. Se pide calcular la aceleración y la Eficiencia de este programa cuando se está ejecutando sobre {1, 2, 4, 8 ,16} procesadores.

Problema 2

- Sea un sistema similar al del problema 1 que posee 20% del código secuencial y el 80% perfectamente paralelizable. Calcular la aceleración aplicando la ley de amdahl's y la ley de Gustafson para n = 16 procesadores.

Referencias

- Zaccone, Giancarlo. Python Parallel Programming Cookbook: Over 70 recipes to solve challenges in multithreading and distributed system with Python 3, 2nd Edition . Packt Publishing.