Parcial

miércoles, 18 de agosto de 2021 18:50

1_ Defina Procesamiento Paralelo y su importancia hoy en día. Defina Sistema Paralelo.
El procesamiento paralelo es el uso de múltiples unidades de procesamiento para resolver un problema computacional.
El procesamiento paralelo es importante por la capacidad que tiene de incrementar la velocidad de cómputo, en especial en el campo de cómputo de alto rendimiento, permitiendo resolver problemas más grandes o más complejos. Es la alternativa que se encontró a las limitaciones en la velocidad de los procesadores, aprovechando el HW en su totalidad, lo que también nos permite ahorrar tiempo y/o dinero.
Un sistema paralelo es la combinación de un algoritmo y una arquitectura paralela en la cual se implementa.

2_ Programación en memoria compartida: características y posibles problemas a enfrentar. Describir los modelos (basado en procesos, threads y directivas).
En la programación en memoria compartida los procesadores se comunican leyendo y escribiendo variables en un espacio de datos común (memoria compartida). Los módulos de memoria pueden ser locales (exclusivos a un procesador) o globales (comunes a todos los procesadores).

Tiene una sub-clasificación por modo de acceso a memoria:
- ○ Acceso uniforme a memoria (UMA)
- ○ Acceso no uniforme a memoria (NUMA)

Se necesita un mecanismo de coherencia de caché
El modelo de programación asociado es la memoria compartida (pasaje de mensajes también es una posibilidad).

Los 3 modelos son:
- Modelo basado en procesos: suponen datos locales (privados) de cada proceso.
- Modelo basado en threads o procesos “livianos”: utiliza hilos que suponen que toda la memoria es global, los hilos a su vez tienen su propio espacio de memoria privada. (Pthreads).
- Modelo basado en directivas: extienden el modelo basado en threads para facilitar su manejo (creación, sincronización, etc) (OpenMP).

Al trabajar con memoria compartida uno de los problemas que podemos es la escalabilidad del sistema, un problema que se ve claramente reflejado en los sistemas de coherencia de cache. Otro problema es que el programador en general no maneja la distribución de los datos ni lo relacionado a la comunicación de los mismos, algo que a veces es necesario para optimizar al máximo un algoritmo.

3_ Pasaje de mensajes: características, ventajas y desventajas (sobre memoria compartida y en gral) y problemas a enfrentar.
Consiste de p procesos (eventualmente procesadores), cada uno de ellos con su espacio de direcciones exclusivo (el espacio de direcciones "global" está particionado en cada proceso). Se usa el intercambio de mensajes para 2 propósitos, el intercambio explícito de datos y la sincronización de los procesos.
Ventajas:
- El programador tiene total control para lograr sistemas más eficientes y escalables.
- Puede implementarse eficientemente en muchas arquitecturas paralelas.
- Más fácil de predecir el rendimiento

Desventaja:
- Mayor complejidad al implementar estos algoritmos para lograr alto rendimiento

En general se implementa con arquitecturas distribuídas.

Los problemas que podes tener es el manejo de los datos y la sincronización, como dependen del programador hay que tener especial cuidado para evitar inconsistencias o situaciones como el deadlock.

Dejo estas caracterísicas random:
Programabilidad: en pasaje de mensajes, se trabaja en bajo nivel. El programador es responsable de la distribución de datos y el mapeo de procesos, así como la comunicación entre tareas

Difícil de programar, difícil de depurar, difícil de mantener. La mayoría de los programas se escriben siguiendo el modelo Single Program Multiple Data (SPMD). ○ No todos ejecutan las mismas instrucciones (sentencias de selección). ○ Los procesos no están sincronizados en la ejecución de cada sentencia.

Eficiencia: el ajuste (tuning) para mejorar el rendimiento puede ser óptimo. Puede manejar el balance de carga, la redistribución de datos y procesos (dinámicamente), replicar datos, entre otras tareas.

Portabilidad: existen librerías para facilitar la ejecución de código sobre diferentes arquitecturas (estándares). Buena portabilidad de código, no necesariamente portabilidad de rendimiento.

4_ Speedup y eficiencia: definir, decir cómo se calculan, límites y qué sucede en arquitecturas heterogéneas. Ejemplo de calculo de la eficiencia en un programa. Porque se puede dar un speedup superlineal.
Con respecto a eficiencia:
Dada una arquitectura heterogénea con 4 procesadores (P0,...,P3) donde la solución secuencial a un problema dado tiene los siguientes tiempos de ejecución en cada uno de ellos P0 -> 200 seg., P1 -> 100 seg, P2 -> 500 seg, P3, 125 seg; si la solución paralela a dicho problema sobre esta arquitectura tarda 40 seg, ¿cual es la eficiencia lograda? explicar paso a paso el cálculo.

El Speedup es una métrica que refleja el beneficio de emplear procesamiento paralelo para resolver un problema dado comparado a realizarlo en forma secuencial.
Se calcula dividiendo el mejor tiempo secuencial sobre el tiempo paralelo:

$$S_p(n) = \frac{T_s(n)}{T_p(n)}$$

El Speedup no tiene un mínimo o máximo definido, puede ser cualquier número mayor a 0. En general, el número máximo que alcanza es p, lo que se llama speedup óptimo, pero lo puede superar, siendo un speedup superlineal (no es lo común). Este speedup superlineal se puede dar por algunos motivos:
Un motivo puede ser que la versión paralela del algoritmo realice menos trabajo que la versión secuencial, por ejemplo la búsqueda DFS. Un segundo motivo de Speedup superlineal es la combinación de características de hardware y distribución de los datos del algoritmo paralelo que ponen en desventaja al algoritmo secuencial. Por ejemplo, al tener más dispositivos, se reduce la cantidad de datos que usa cada uno y se puede aumentar la cantidad de cache hits.

En arquitecturas heterogéneas cambia la forma de evaluarlo, ya que el speedup óptimo pasa de ser p a la potencia de cómputo total.

La eficiencia es una medida de la fracción de tiempo en la cual las unidades de procesamiento son empleadas en forma útil.
Se calcula como la división entre el speedup del algoritmo y el speedup óptimo:

$$E_p(n) = \frac{S_p(n)}{S_{Opt}}$$

El valor varía entre número mayores a 0 y menores o iguales a 1, siendo 1 el caso en donde los procesadores se aprovechan al máximo (puede llegar a devolver un número mayor a 1 si se da un speedup superlineal, pero no es lo usual).

En arquitecturas heterogéneas debemos recordar que el speedup óptimo no es el número de procesadores, sino la potencia de cómputo total.

Dada una arquitectura heterogénea con 4 procesadores (P0,...,P3) donde la solución secuencial a un problema dado tiene los siguientes tiempos de ejecución en cada uno de ellos P0 -> 200 seg., P1 -> 100 seg, P2 -> 500 seg, P3, 125 seg; si la solución paralela a dicho problema sobre esta arquitectura tarda 40 seg, ¿cual es la eficiencia lograda? explicar paso a paso el cálculo.

La eficiencia se calcula como la división entre el speedup y el speedup óptimo.

El speedup se calcula como el mejor tiempo secuencial dividido el tiempo de ejecución paralelo. El mejor tiempo secuencial es 100seg, y el tiempo de ejecución paralelo 40seg. Speedup = 100/40 = 2,5

El speedup óptimo, como es una arquitectura heterogénea, se calcula como la potencia de cómputo total (pct).

La pct es la sumatoria de las potencia de cómputo relativas de cada procesador (pcr).
La pcr de cada procesador se calcula como la división de la potencia del procesador dividido la potencia del mejor procesador.
Vamos a sacar las pcr (la división se hace al reves porque no tenemos las potencias de los procesadores, sino el tiempo que tardan):
P0 --> 100/200 = 0,5
P1 --> 100/100 = 1
P2 --> 100/500 = 0,2
P3 --> 100/125 = 0,8

pct = 0,5 + 1 + 0,2 + 0,8 = 2,5

Eficiencia = 2,5/2,5 = 1

5_ Etapa de Mapeo: definir, qué importancia tiene y que técnicas de mapping existen.
El mapeo es la manera en la que se asignan las tareas a los procesadores, para ello se tienen en cuenta las características de las tareas (modo de generación, tamaño y conocimiento de la tarea, volumen de datos asociado) y de los procesadores.
Es muy importante ya que si no se asignan las tareas de manera correspondiente, se puede presentar un desbalance de carga que degrade el rendimiento del sistema.
Técnicas de mapeo son 2:

- Estática: distribuye las tareas entre los procesos antes de la ejecución, para ello es fundamental conocer las características de las tareas. Para casos complejos se emplean heurísticas. En general los algoritmos son más fáciles de diseñar y programar.
- Dinámica: distribuyen las tareas entre los procesos durante la ejecución. Si las tareas se generan dinámicamente, entonces deben mapearse dinámicamente también. Si no se conoce de antemano el tamaño de las tareas, el mapeo dinámico suele dar mejor resultado. Si el volumen de datos asociado a cada tarea es grande pero el cómputo no es significativo, un mapeo dinámico podría incurrir en un alto overhead por la migración de datos.

6_ Explica la importancia del balanceo de carga en un programa paralelo. Indicar las posibles causas de desbalances . Describir las técnicas (Estática Directa, Estática predictiva y Dinámica por demanda). Indicar cuando conviene usar cada una (teniendo en cuenta la arquitectura y las características de las tareas).
El balance de carga es imporante ya que, si no se realiza de forma correcta, un procesador va a tardar más que otro, y no se estaría aprovechando el paralelismo al máximo.
El balance de carga es importante para optimizar la perfomance. Si el balanceo no se hace bien, algunos procesadores pueden tener más carga que otros, haciendo que los procesadores con menos carga terminen antes que los otros, y queden ociosos, y así prolongar el tiempo total de ejecución.
Posibles causas de desbalances:

- Procesadores heterogéneos

- Tareas con cómputo no uniforme
- Tareas con volumen de datos diferente

El balance estático directo es cuando se reparten todas las tareas de manera equitativa (al ser estático se conocen la cantidad de tareas totales antes de la ejecución).

El balance estático predictivo puede usarse cuando se conoce la cantidad de trabajo que va a realizarse, y además la potencia de cálculo de los procesadores. De esta manera, se puede repartir el trabajo a priori entre los procesadores, de manera proporcional a su potencia.

En el balanceo de carga dinámico por demanda, se reparten las tareas durante la ejecución, en donde cada procesador cuando termina su tarea demanda otra.

El balance estático directo conviene usarlo cuando se conocen la cantidad de tareas que se van a realizar y hay una arquitectura homogénea.

El balance estático predictivo conviene usarlo cuando se conocen la cantidad de tareas que se van a realizar y hay una arquitectura heterogénea, en donde se sabe la potencia relativa de cada procesador.

El balance dinámico conviene usarlo cuando las tareas se generan de manera dinámica (no se conoce el total a priori), cuando las tareas no son uniformes (no se conoce el cómputo que requiere, o se sabe que no es equitativo con el resto de las tareas).

7_ Explique los siguientes paradigmas y ejemplifique: divide and conquer, pipelining, master/worker y sus variaciones, SPMD. Indique y justifique a qué paradigma se adecuan más las siguientes redes estáticas: estrella, todos con todos, anillo, arbol.

Divide and conquer: consiste en 2 etapas:
Dividir: fase en la que se particiona sucesivamente el problema en sub-problemas más pequeños hasta obtener una granularidad deseada.
Conquistar: fase en la que se resuelven los subproblemas en forma independiente.
En ocasiones, se requiere una fase adicional de combinación de resultados parciales para llegar al resultado final.
Las redes basadas en árboles son las más adecuadas para este tipo de paradigma ya que esta topología refleja la descomposición recursiva de los problemas.

Pipelining:
El cómputo se descompone en una secuencia de procesos por la que pasan los datos, se puede ver como una cadena de productores/consumidores en donde cada proceso consume los datos generados por el pipe anterior y los procesa para que los use el pipe siguiente.
Por lo generar tiene forma lineal, pero puede ser de arbol o grafo por ejemplo.
La red de anillo es la más adecuada para este paradigma, porque los datos viajan linealmente de un proceso al siguiente.

Master/worker: hay un proceso denominado master y varios denominados workers.
El proceso Maestro es el responsable de generar trabajo y asignárselo a los Workers.
La manera en que el proceso Master reparte las tareas puede variar:

Si se puede estimar de antemano el tamaño de las tareas, se puede realizar un mapeo estático.

Si no se puede, se realiza un mapeo dinámico, en donde el Master reparte las tareas por demanda de los workers.

La red de estrella es la que más se ajusta al siguiente paradigma, ya que el nodo central sería el proceso master, el cual tiene comunicación con los demás workers.

SPMD:
En este paradigma cada proceso ejecuta el mismo programa sobre una porción de datos diferente. El

cómputo puede tener varias fases, las cuales generalmente son intercaladas con comunicación y sincronización.
La red todos con todos sería la más apropiada, ya que en las diferentes fases los procesos puede comunicarse con los demás para continuar su ejecución.

8) Explique las descomposiciones en tareas: recursiva, basada en los datos, exploratoria, especulativa.
Descomposición de tareas recursiva:
El problema inicial es dividido en un conjunto de subproblemas independientes. Luego, cada uno de estos subproblemas son recursivamente descompuestos en otros subproblemas independientes más pequeños hasta alcanzar una determinada granularidad.
En ocasiones, puede requerirse alguna fase de combinación de resultados parciales.

Descomposición de tareas basada en datos:
Generalmente es usada en problemas que operan sobre grandes estructuras de datos. Requiere de dos pasos, particionar los datos que se procesarán y usar la partición anterior para inducir una descomposición del cómputo en tareas. Tiene dos variantes, la desposición basada en datos de salida, que se utiliza cuando los datos de salida se puede calcular de manera independiente en función de los datos de entrada. La otra alternativa es la función basada en los datos de entrada, que se suele utilizar cuando no se puede realizar una descomposición por datos de salida, y en su lugar se le asigna una partición de los datos de entrada a cada proceso, y el mismo se encarga de todos los cómputos asociados a esa partición.

Descomposición de tareas exploratoria:
Se suele emplear en aquellos problemas cuya solución requiera una búsqueda en un espacio de soluciones. Se particiona el espacio de búsqueda y se le asigna una porción a cada proceso, y se busca la solución de manera concurrente hasta que un proceso la encuentre.

Descomposición de tareas especulativa:
Se emplea cuando un programa podría tomar uno o varios caminos que implican cómputo significativo pero la decisión depende de la salida de algún cómputo anterior.

4) Explicar al menos 6 aspectos a tener en cuenta al realizar un programa paralelo para minimizar el overhead por comunicación al trabajar con pasaje de mensajes sobre una arquitectura distribuida.
Existen diferentes métodos:

- Minimizar volumen de datos intercambiados: a mayor volumen de datos intercambiados, mayor tiempo de comunicación.
- Minimizar frecuencia de las interacciones: cada interacción tiene un costo inicial de preparación. Siempre que sea posible, conviene combinar varias comunicaciones en una sola.
- Minimizar competencia entre recursos y zonas críticas (hotspots): evitar posibles cuellos de botella mediante el uso de técnica descentralizadas. Replicar datos si es necesario.
- Solapar cómputo con comunicaciones: mediante el uso de operaciones no bloqueantes en pasaje de mensajes y técnicas de multi-hilado y prebúsqueda en memoria compartida.
- Replicar datos o cómputo: si permite reducir las interacciones (mensajes o sincronización).
- Usar operaciones de comunicación colectiva.
- Solapar comunicaciones con otras comunicaciones: siempre y cuando el hardware de soporte lo permita, solapar diferentes comunicaciones puede reducir el overhead.

3) Explique en qué consiste la programación con threads. ¿Que diferencia tiene con directivas? Explique ventajas y desventajas de cada uno.

- Modelo basado en threads o procesos “livianos”: utiliza hilos que suponen que toda la memoria es global, los hilos a su vez tienen su propio espacio de memoria privada. (Pthreads).
- Modelo basado en directivas: extienden el modelo basado en threads para facilitar su manejo (creación, sincronización, etc) (OpenMP).

La diferencia del modelo de threads con el modelo de directivas es, basicamente, la facilidad y comodidad en la programación. El modelo basado en directivas es una extención del modelo de threads con varias funciones o comodidades para hacer más fácil el desarrollo, sin que el programador tenga que encargarse directamente de todo el manejo de hilos.
La ventaja del modelo basado en directivas es justamente esa facilidad en el desarrollo, la desventaja es que el código se debe traducir a un código de threads nativo antes de la ejecución, lo que genera un pequeño overhead.
Las ventajas y desventajas del modelo en threads en comparación con el de directivas son las mismas a la inversa, es más dificil su programación pero no necesita conversión para ejecutarse.

3) Diferencias paralelismo explícito e implícito. Explicar el funcionamiento de los procesadores VLIW (Very Long Instruction Word) para lograr paralelismo
En el paralelismo explícito es el programador el que especifica las tareas paralelas que van a ejecutarse, usando primitivas y otros constructores del lenguaje que describen cómo se va a hacer el cómputo paralelo.

En el paralelismo implícito, es el sistema el que introduce el paralelismo, de manera invisible para el usuario y sin afectar la semántica. Esto se logra usando diferentes métodos de hardware como pipelines o ejecución superescalar.

El costo y complejidad del hardware del scheduler fijan una limitación a los procesadores superescalares. Para explotar el paralelismo a nivel de instrucciones, los procesadores VLIW usan otra técnica que le delega al compilador la resolución de dependencias y disponibilidad de recursos en tiempo de compilación. Las instrucciones que pueden ser ejecutadas concurrentemente se agrupan y se envían al procesador como una sola instrucción larga (Very Long Instruction Word), para ser ejecutadas al mismo tiempo en múltiples unidades funcionales

2) Demuestre la importancia de la memoria caché en las máquinas secuenciales con un ejemplo concreto. Que overhead pueden traer en memoria compartida que no ocurre en memoria distribuida.

El overhead que puede traer el uso de memoria compartida es el del mecanismo de coherencia de cache. El mejor que se puede lograr es, con el uso de invalidación para mantener la coherencia, el sistema basado en directorios.
En este sistema, la memoria principal mantiene un registro con información de estado sobre los bloques de caché de cada procesador, esto con el fin de que cada operación de coherencia interactúe solo con aquellos procesos involucrados.
El problema que puede traer es que si hay muchas operaciones de coherencia, la memoria del sistema se vuelva un cuello de botella y los procesadores tengan que competir por su acceso, además del problema de escalabilidad que puede tener si se incrementa la cantidad de procesos en el sistema, haciendo que el directorio ocupe cada vez más memoria.
Este problema no se presenta en memoria distribuída ya que el sistema de directorios se distribuye.

Consideremos un procesador con frecuencia de reloj de 1 GHz (1 ns clock) conectado a una memoria DRAM con una latencia de 100 ns (sin caches).
Supongamos que el procesador tiene dos unidades multiplicación-suma y es capaz de ejecutar 4 instrucciones por ciclo de reloj de 1 ns.

- El pico de rendimiento del procesador es 4 Gflops.
- Dado que la memoria tarda 100 ciclos de reloj en responder a un pedido, esto significa que 1
  pedido a memoria significa 100 ciclos de espera para el procesador.

Si ahora sobre este procesador tenemos que multiplicar dos vectores de K elementos cada uno. Básicamente en cada producto se requiere dos accesos a memoria para buscar los operandos a multiplicar.

- Un producto asociado con dos lecturas, pone el ritmo de procesamiento en 1 operación de producto-suma cada 200 ns (es decir 2 operaciones de punto flotante cada 200 ns, una multiplicación y una suma).

- Pasamos de 4 Gflops pico a 10 Mflops reales

Agregamos una memoria caché de 32 KB con latencia de 1 ns. Trataremos de multiplicar 2 matrices A y
B de 32 x 32 (esto permitiría alojar en caché las matrices A, B y el resultado C).

- El fetch de las dos matrices para llevarlas de la caché al procesador significa leer 2K palabras:
  2000 x 100 ns = 200 µs
- Multiplicar las dos matrices de n × n significa $2n^3$ operaciones. En nuestro problema n=32 significa 64K operaciones, que pueden realizarse en 16K ciclos (o 16 µs) a 4 operaciones por ciclo
  de 1 ns.
- Cada operación son dos lecturas de cache. Aproximadamente: 64000 x 2 ns = 128 µs
- El tiempo total de cómputo pasa a ser entonces: 200 µs + 16 µs + 128 µs +100 µs (store resultado
  en memoria) = 444µs
- Esto corresponde a 64K/444 µs = 144 Mflops

- Consideremos un procesador capaz de ejecutar 1 instrucción de punto flotante por nanosegundo (ns) conectado a una memoria con una latencia de 100 ns (sin cachés)
  - El rendimiento pico teórico de este sistema es de 1 segundo = 1.000.000.000 nanosegundos → 1.000.000.000 flop's por segundo = 1GFlops
- Supongamos que debemos multiplicar 2 matrices A y B de *nxn* elementos cada una
  - Para esto se requieren n^3 operaciones de multiplicación/suma → $2n^3$ ns de cómputo
  - Para cada operación de multiplicación/suma se requiere acceder a memoria para buscar ambos operandos → $200n^3$ ns de acceso a memoria
  - Para guardar los resultados en memoria se requiere un acceso por cada elemento → $100n^2$ ns
- Si n=32 entonces:
  - Se requieren $2x32^3$ operaciones de multiplicación/suma → $2x2^{15}$ ns de cómputo.
  - Para cada operación de multiplicación/suma se requiere acceder a memoria para buscar ambos operandos → $200x2^{15}$ ns de acceso a memoria
  - Para guardar los resultados en memoria se requiere un acceso por cada elemento → $100x2^{10}$ ns
  - En total: $102{,}64x2^{16}$ ns para resolver 2^{16} operaciones → 9,75 Mflops

El rendimiento baja de 1 Gflop (1000 Mflops) a 9,75 Mflops

- Consideremos el mismo sistema del ejemplo anterior pero ahora con una memoria caché de 32Kb con latencia de 4 ns y ancho de banda igual a un elemento de la matriz
- Se debe resolver la misma multiplicación de matrices con n=32. Como ambas matrices (A y B) entran en caché, se accede a memoria únicamente una vez por cada elemento:
  - Para esto se requieren n^3 operaciones de multiplicación/suma → $2n^3$ ns de cómputo
  - Para cada operación de multiplicación/suma se requiere acceder a memoria para buscar ambos operandos → Para cada elemento se accede una vez a memoria (100 ns) y el resto a caché (4x(n-1) ns) → en total se requieren (96+4xn) $2n^2$ ns de acceso a memoria
  - Para guardar los resultados en memoria se requiere un acceso por cada elemento → $100n^2$ ns

- Si n=32 entonces:
  - Se requieren $2x32^3$ operaciones de multiplicación/suma → $2x2^{15}$ ns de cómputo.
  - Para cada operación de multiplicación/suma se requiere acceder a memoria o caché para buscar ambos operandos → $448x2^{10}$ ns de acceso a operandos
  - Para guardar los resultados en memoria se requiere un acceso por cada elemento → $100x2^{10}$ ns
  - En total: $9{,}5625x2^{16}$ ns para resolver 2^{16} operaciones → 104,6 Mflops

El rendimiento sube de 9,75 a 104,6 Mflops