# Arquitecturas paralelas

Hay distintos enfoques para clasificar arquitecturas paralelas:

Enfoque 1: Organización por espacio de direcciones

Tiene dos clasificaciones: Por memoria compartida o memoria distribuida.

Estas clasificaciones están relacionadas a la forma de comunicación que hay entre los procesos, en memoria compartida los procesos se comunican accediendo a datos compartidos en memoria mientras que en memoria distribuida se utiliza pasaje de mensajes como forma de comunicación.

En los **multiprocesadores con memoria compartida** hay múltiples procesadores que acceden a una memoria compartida por todos y se interactúa modificando datos en esa memoria.

Tienen dos subclasificaciones:

* Puede ser un esquema UMA que posee un bus o un crossbar switch. Esto puede generar problemas de sincronización y consistencia.
* Se pueden usar esquemas NUMA con un mayor número de procesadores.
* En un esquema NUMA un CPU tiene un bloque de memoria compartida y se conecta con otros CPU mediante una red de comunicación muy rápida. El acceso es más rápido para acceder al bloque propio de un CPU.

Diagrama

Descripción generada automáticamente

En los **multiprocesadores con memoria distribuida** hay procesadores conectados por una red, donde cada procesador tiene su memoria local, no puede acceder a la memoria local de otro procesador. No hay problemas de consistencia y tampoco hay por pasaje de mensajes.

Se pueden subclasificar por su grado de acoplamiento. El grado de acoplamiento es la cercanía entre los procesadores y las memorias. Fuertemente acoplado significa que los procesadores y la red están físicamente cerca. El otro extremo débilmente acoplado es cuando hay más lejanía, por ejemplo una red de computadoras conectadas mediante internet en lugares físicamente distanciados. Un ejemplo del medio son casoso como un cluster, donde hay varias pcs conectadas a una red local que está dedicada específicamente a ese cluster.

Enfoque 2: Granularidad

Indica la relación entre la potencia de las máquinas que tengo y la potencia o velocidad de la red de comunicación.

Si tengo máquinas muy potentes y una red de comunicación lenta entonces tenemos una arquitectura de **grano grueso.** Puedo tener pocas máquinas potentes y una comunicación lenta, siendo conveniente hacer pocos procesos o hilos que se deban comunicar poco entre ellos. Se sacrifica la concurrencia al haber menos máquinas.

En cambio una arquitectura de **grano fino** son muchas máquinas poco potentes con una red de comunicación muy rápida. En este caso conviene tener una aplicación con mucha concurrencia para aprovechar las máquinas, de forma que cada tarea sea de poco cómputo y que se comuniquen mucho entre máquinas.

Hay que adaptar el programa que tenga de acuerdo a la granularidad, ya que sino habría desperdicio, por ejemplo no puedo poner un programa con mucha comunicación en una arquitectura de grano grueso.

Enfoque 3: Mecanismo de control

Este enfoque clasifica las arquitecturas en 4 clases de acuerdo a la cantidad de instrucciones diferentes que se pueden ejecutar con respecto a la cantidad de datos sobre los cuales se puede ejecutar.

* **SISD (single instruction, single data):** La arquitectura va a ejecutar una única instrucción por ciclo de reloj y lo va a hacer sobre un dato particular. Un ejemplo claro son los monoprocesadores que hacían esto. Su ejecución es determinística y secuencial
* **MISD (multiple instruction, single data):** En un ciclo de reloj se ejecutan múltiples instrucciones sobre un mismo dato, son máquinas hipotéticas. Su modo de operar es sincrónico.
* **SIMD:(single instruction, multiple data):** Una sola instrucción que se aplica a múltiples datos de forma sincrónica y determinística. Hay varios procesadores idénticos con sus memorias donde a estos procesadores se le envía una única instrucción para que ejecuten esa misma instrucción sobre algún dato en su memoria particular.   
  Un ejemplo puede ser un if con una condición para la cual unos procesadores deben ejecutar y otros no, con esto podemos deshabilitar y habilitar de forma selectiva los procesadores para que ejecuten o no.  
  Esta clasificación es adecuada para aplicaciones con un alto grado de regularidad. Se está tomando en cuenta por las GPUs que usan este mecanismo de control.
* **MIMD (multiple instruction multiple data):** Es la opción usada hoy en día para hacer paralelismo, es de propósito general y es la arquitectura sobre la cual se ejecutan las aplicaciones vistas en la materia. Cada unidad de procesamiento ejecuta su propia instrucción sobre sus propios datos , de forma asincrónica, a su propio ritmo. Pueden tener memoria compartida o memoria distribuida. Ejemplo son los clusters o los clusters de multicore.

Tienen dos subclasificaciones

* + **MPMD (multiple program multiple data):** Cada procesador ejecuta su propio programa
  + **SPMD (single program multiple data):** Hay un único programa fuente y cada procesador ejecutara su copia independiente.

Enfoque 4: Red de interconexión

La memoria compartida y distribuida pueden construirse mediante la conexión de varios CPU con varios bloques de memoria. Esas redes a usar pueden ser de distinto tipo:

* **Redes de interconexión estáticas:** Son links punto a punto para conectar los módulos, usualmente se utiliza en máquinas de pasaje de mensajes. En ellas cada nodo es una computadora completa.

Ejemplos:

Diagrama, Esquemático

Descripción generada automáticamente

* + Completamente conectada es la más rápida de todas ya que hay comunicación directa en un solo paso (hay link directo). Un nuevo procesador agregado implica agregar un montón de links.
  + La conexión en estrella es un nodo central y todos los demás se conectan a él, agregar un nuevo nodo es agregar un solo link, lo malo es que la comunicación entre procesos que no sean nodos central, entonces la comunicación es en 2 pasos.
  + El arreglo lineal o anillo, es una secuencia de nodos, cada uno conoce a su vecino izquierdo y derecho y los extremos conocen un solo vecino, la comunicación es lenta, si quiero conectar los procesos del extremo debo comunicarme en 3 pasos. En el caso del anillo se reduce la comunicación de extremos ya que estos están unidos, el peor caso es de un extremo al medio.
  + En el árbol estático, las aristas son las conexiones, se reduce la cantidad de pasos.
  + Los hipercubos, permiten asegurar que la comunicación se va a hacer como máximo en D pasos.
* **Redes de interconexión dinámica:** Son construidas usando switches y enlaces de comunicación, este caso se piensa para máquinas de memoria compartida.

Ejemplos:

Diagrama, Dibujo de ingeniería

Descripción generada automáticamente

* + - En crossbar se activa un switch por cada bloque de memoria, es la red más rápida de todas pero lo malo que tiene es que es costosa de implementar y agregar procesador/memoria es agregar nuevos cables.
    - En buses hay un único canal a través del cual las unidades de procesamiento se comunican y acceden a memoria, es la menos costosa si se quiere agregar un CPU o más memoria pero es la más lenta para las comunicaciones o memoria.
    - En multistage se pasa por una serie de etapas que cada una tiene un conjunto de switch que van desviando las CPUs a los bloques de memoria deseado, es rápido y es fácil de ampliar. Es más un caso intermedio.

# Diseño de algoritmos paralelos

La mejor solución a un algoritmo paralelo puede ser distinta por la mejor solución de un algoritmo secuencial, no siempre se puede paralelizar algoritmos secuenciales ( a veces no vale el costo).

Hay un enfoque metódico para maximizar el rango de opciones a tener en cuenta, la idea es generar mecanismos para evaluar alternativas, de forma que se puede reducir el backtracking por malas elecciones.

Los aspectos independientes de la máquina se consideran de forma temprana mientras que los aspectos específicos se demoran. Esto se termina simplificándose en 2 pasos importantes: Primero la **descomposición del problema en tareas concurrentes** ( no toma en cuenta la arquitectura) y el **mapeo de las tareas en las unidades de procesamiento.**

Para esto se toman en cuenta:

* Cual es la granularidad de las tareas, que conviene respecto a la arquitectura
* De qué forma se mapean las tareas y datos a las distintas unidades de procesamiento
* Como se manejara la comunicación y sincronización entre procesos. Si es una arquitectura distribuida no hay memoria compartida, si hay memoria compartida se debe elegir como hacer la comunicación.
* Asegurar la corrección del programa (asegurar resultados correctos). Evitar deadlocks y desbalances (un procesador trabaja más que otro).
* Tener un cierto grado de tolerancia a fallo, emplear metodologías para manejarlos.
* Manejar la heterogeneidad, por ejemplo tener distintos procesadores es una misma arquitectura o el cómo funciona cada una
* Lograr que el sistema sea escalable, de forma que se pueden agregar mejoras de forma fácil.
* Consumo energético.

Para diseñar un algoritmo paralelo se deben realizar algunos de los siguientes pasos:

* Identificar tareas concurrentes
* Mapear tareas a procesos en distintos procesadoreh
* Distribuir los datos de entrada, datos intermedios y datos de salida
* Manejar el acceso a los datos compartidos ( como es la comunicación y sincronización)
* Sincronizar los procesos, cuál tipo de sincronización se usará entre procesos.

**Siempre están presentes la descomposición en tareas y el mapeo de tareas a procesos.**

Etapa de descomposición de tareas

Se descompone el problema en componentes funcionales de forma concurrente. Al principio se plantean tareas lo más pequeñas posibles, teniendo una descomposición de grano fino para obtener la máxima concurrencia posible.

En etapas posteriores se analiza cada tarea , su comunicación , sincronización y la arquitectura a usar y se juntan distintas tareas para obtener una composición de grano grueso. Estas tareas de grano grueso se van a poder convertir en procesos o hilos para después mapearlos.

Hay 2 tipos de descomposición posibles:

* **Descomponer el dominio (paralelismo de datos):** Se descomponen los datos, cada tarea de igual código trabaja sobre conjuntos diferentes de esos datos.
* **Descomponer funcionalmente (paralelismo funciona):** Se descompone el cómputo en tareas distintas y luego se tratan los datos. Los datos pueden ser necesitados por una tarea o entre varios, teniendo que revisar cuando comunicar la información entre las tareas. Es muy usada para darle estructura a un programa, para dar más claridad.
* Se puede tener una descomposición híbrida, primero una descomposición funcional y luego una descomposición de datos sobre las tareas resultantes.

Aglomeración

El algoritmo resultante de la descomposición de tareas pequeñas es un algoritmo abstracto ya que no se especializa para ejecutar de forma eficiente en una máquina específica.

Se tratan de juntar varias de estas tareas para generar una tarea que puede transformarse en un proceso o hilo. La idea es obtener una descomposición de grano grueso.

Se analiza también si es necesario replicar datos o cómputo, por ejemplo para evitar comunicación con retardos o que las tareas ya tengan la misma info y no se comuniquen.

Hay 3 objetivos a seguir en la aglomeración y replicación:

* **Incremento de la granularidad de tareas:** Juntar tareas que tengan mucha comunicación entre ellas para reducir la cantidad de comunicaciones. Hay que priorizar tareas que necesitan cierta secuencialidad al comunicarse, no conviene juntar tareas que se pueden ejecutar de forma concurrente.
* **Preservación de la flexibilidad:** Al juntar tareas se puede limitar la escalabilidad del algoritmo, la idea es lograr una composición de grano grueso sin sacrificar tanta portabilidad y escalabilidad del algoritmo.
* **Reducción de costos de IS:** Se intenta evitar cambios extensivos por ejemplo reutilizando rutinas existentes

Una vez finalizada la descomposición se tienen un conjunto de tareas que tienen ciertas características que pueden ser útiles para la etapa de mapeo: La generación de las tareas, su tamaño, el conocimiento del tamaño de las tareas y el volumen de datos asociado a cada tarea.

Etapa de Mapeo de tareas a procesadores

Las tareas de la etapa anterior se distribuyen en las distintas unidades de procesamiento. Esta etapa no existe en uniprocesadores o máquinas de memoria compartida con scheduling de tareas automático.

El objetivo es minimizar el tiempo de ejecución de la aplicación. Para esto hay dos estrategias que a veces conflictúan: La primera es ubicar tareas que pueden ejecutar concurrentemente en diferentes procesadores para mejorar la concurrencia o poner tareas que se comunican con frecuencia en procesadores iguales para incrementar la localidad.

El problema es NP-completo: No existe un algoritmo de tiempo polinomial tratable computacionalmente para evaluar tradeoffs entre estrategias en el caso general.

Normalmente después de la primer etapa habrá más tareas que procesadores físicos de la arquitectura, por lo que se necesita un mecanismo para mapear esas tareas a los procesadores físicos.

El lenguaje de especificación de algoritmos paralelos debe poder indicar claramente las tareas que pueden ejecutarse concurrentemente y su precedencia para el caso que no haya suficientes procesadores para atenderlas.

La dependencia entre tareas condicionarán el balance de carga entre procesadores y su interacción debe tender a minimizar la comunicación de datos entre procesadores físicos.

**Lograr un buen mapping es crítico para el rendimiento de algoritmos paralelos:**

Hay que:

1. Tratar de mapear tareas independientes a distintos procesadores
2. Asignar prioritariamente los procesadores disponibles a tareas que están en un camino crítico
3. Asignar las tareas con alto nivel de interacción al mismo procesador, de modo de disminuir el tiempo de comunicación físico.

Estos criterios pueden oponerse entre sí, por ejemplo el 3 puede llevarnos a no paralelizar. Debe encontrarse un equilibrio que optimice el rendimiento paralelo. EL MAPPING DETERMINA LA EFICIENCIA DEL ALGORITMO.

# Métricas del paralelismo

Es importante medir la velocidad, eficiencia y funcionamiento de una aplicación. En el mundo secuencial, la performance se mide teniendo en cuenta requerimientos de tiempo y memoria del programa.

En un algoritmo paralelo es interesante saber cual es la ganancia en performance. Hay otras medidas que deben tenerse en cuenta siempre que favorezcan a sistemas con mejor tiempo de ejecución.

A falta de un modelo unificador de cómputo paralelo, el tiempo de ejecución depende del tamaño de la entrada y de la arquitectura y número de procesadores

La diversidad de un sistema paralelo torna complejo el análisis de performance, depende de que interesa medir, de qué sistema es mejor que otro y qué sucede si se agregan procesadores.

En una medición de performance es usual elegir un problema y testear el tiempo variando el número de procesadores. Aquí subyacen las nociones de speedup, eficiencia y la ley de Amdahl. Otro tema interesante es la escalabilidad que da una medida de usar eficientemente un número creciente de procesadores.

Speedup

El **tiempo paralelo** es el tiempo que tarda desde que empieza a ejecutarse el primero de los procesos o hilos hasta que termina el último. El **tiempo secuencial** es el tiempo en que tardaría en ejecutarse la mejor solución secuencial a ese problema.

**Speedup** es el cociente entre el tiempo de ejecución del algoritmo serial conocido más rápido y el tiempo de ejecución paralelo del algoritmo elegido. El speedup óptimo depende de la arquitectura:



Texto

Descripción generada automáticamente con confianza media

El rango de valores es en general entre 0 y S óptimo. No hay nunca un rango de valores donde el speedup sea malo. Si el speedup me da un valor menor a 1, significa que la solución paralela es más lenta que la secuencial. Si me da 1, entonces son al mismo tiempo, la paralelización no tiene sentido, si dá mayor a 1 significa que ya tengo mejoras.

El speedup puede ser lineal o perfecto, sublínea y superlineal.

**Speedup lineal:** Es cuando el speedup es igual al speedup óptimo.

**Speedup sublineal:** El speedup tiene valores menores al speedup óptimo, es el resultado más normal.

**Speedup superlineal:** Es cuando se logra un speedup que es mayor al speedup óptimo.

Eficiencia

Es el coeficiente entre el speedup y el speedup Óptimo. Mide la fracción de tiempo en que los procesadores son útiles para el cómputo. El valor está entre 0 y 1, dependiendo de la efectividad de uso de los procesadores. Cuando es 1 corresponde al speedup perfecto.



Escalabilidad

Es muy difícil extrapolar la performance de un sistema paralelo, a partir de configuraciones con pocos procesadores y datos reducidos.

No sirven con estudios con 2,4,8 procesadores que proyectan el Sp alcanzable con 128 y 256 procesadores o el tiempo de procesamiento cuando tengamos 100 o 1000 veces más datos. Básicamente porque los resultados con pequeños conjuntos de datos están afectados por la localidad en el manejo de la memoria y los resultados con pocos procesadores porque las comunicaciones no computan los costos relacionados con la distancia entre procesadores y la disminución del ancho de banda efectivo.

Factores que limitan el speedup obtenible y la eficiencia

* Alto porcentaje de código secuencial, si tengo mucho código secuencial, ese código se ejecutará por un solo proceso por vez, dejando a los procesos ociosos. (Ley de Amdahl). Hay que tratar de reducir lo que se deba hacer secuencialmente.
* Alto porcentaje de operaciones de E/S respecto de la computación. Lo ideal es aprovechar la localidad espacial y temporal de los datos, tratar de no imprimir tanto, de ser posible no esperar a ingresos de datos por teclado, tratar de reducir accesos a memoria, aprovechar uso de cache.
* Algoritmo no adecuado: en este caso necesitamos rehacer el código.
* Excesiva contención de memoria: muchos deben acceder a la misma posición de memoria, el acceso se termina secuencializando. Hay que minimizar la cantidad de variables compartidas.
* El tamaño del problema puede hacer que la arquitectura no pueda crecer.
* Desbalance de carga, pueden haber procesos ociosos, generando retrasos que afectan la eficiencia. Hay que tratar de distribuir el trabajo de otra forma.
* Overhead por el solo hecho de que la aplicación sea paralela

# Paradigmas de programación paralela

Un paradigma de programación son clases de algoritmos que resuelven distintos problemas pero tienen la misma estructura de control.

Para cada paradigma puede escribirse un esqueleto algorítmico que define la estructura de control común.

Dentro de la programación paralela pueden encontrarse paradigmas que permiten encuadrar los problemas en alguno de ellos.

En cada paradigma, los patrones de comunicación son muy similares en todos los casos.

Paradigma cliente servidor

Es el esquema predominante en las aplicaciones de procesamiento distribuido. Los servidores son procesos que esperan pedidos de servicios de múltiples clientes. Naturalmente unos y otros pueden ejecutarse en procesadores diferentes.

La interacción tiene como característica que es bidireccional. Lo que cambia en el paradigma es lo que el cliente hace y la forma en que van a interactuar entre ellos.

Existen varios mecanismos de invocación para usar pero el que más suele servir son aquellos con comunicación bidireccional como RPC, Rendezvous o incluso Monitores. Con pasaje de mensajes se puede implementar pero se necesita un canal para la comunicación y otro para la respuesta. Aparte que habría uno de pedido por cada cliente y uno de respuesta por cada uno.

El soporte distribuido puede ser simple (LAN) o extendido a la WEB. 

Paradigma master/worker o master/Slave (segs)

Está basado en organizaciones del mundo real. El master envía iterativamente datos a los workers y recibe resultados de estos.

El master envía iterativamente datos a los workers y recibe resultados de estos. Hay un posible cuello de botella por ejemplo por tareas muy chicas o slaves muy rápidos. Hay que usar una arquitectura del grano adecuado.

Si el master solo enviará información no sería necesario en memoria compartida porque la info a enviar podría ser compartida ( por estar en memoria ). Normalmente se usa en memoria distribuida.

Hay dos casos de acuerdo con las dependencias de las iteraciones:

* Iteraciones dependientes: El master necesita los resultados de todos los workers para generar un nuevo conjunto de datos.
* Entradas de datos independientes: Los datos llegan al maestro, quien no necesita resultados anteriores para generar un nuevo conjunto de datos.

También hay dos opciones para la distribución de datos:

* Distribuir todos los disponibles de acuerdo a alguna política.
* Bajo petición o demanda (dinámico).

Existen variantes pero básicamente un procesador es responsable de la coordinación y los otros resolver los problemas asignados. Es una variación de SPMD donde hay dos programas en lugar de solo uno.

Pueden darse distintos casos:

* Procesadores heterogéneos y con distintas velocidades: Esto puede dar problemas con el balance de carga.
* Trabajo que debe realizarse en fases, esto requiere sincronización
* Generalización a modelo multi nivel o jerárquico.

Pipeline y Algoritmos sistólicos

El problema se particiona en una secuencia de pasos. El stream de datos pasa entre los procesos y cada uno realiza una tarea sobre él.

Un ejemplo es el filtrado, etiquetado y análisis de escena en imágenes.

Es un mapeo natural a un arreglo lineal de procesadores.

Dividir y conquistar

En general implica **paralelismo recursivo** donde el problema general puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos. Sale de un programa secuencial que es recursivo.

En lugar de hacer llamados recursivos en el programa principal lo que va a hacer es generar nuevos procesos que trabajan con datos más chicos que van a estar ejecutándose.

Una vez que ya se puede resolver el subproblema generado comienza la etapa de conquistar donde se resuelven estos problemas de forma independiente, usualmente de manera recursiva. Las soluciones son combinadas en la solución global.

La subdivisión puede corresponderse con la descomposición entre procesadores. Cada subproblema puede mapearse a un procesador. Cada proceso recibe una fracción de datos: Si puede los procesa sino crea un número de hijos y les distribuye los datos.

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Mucho énfasis en paralelismo recursivo, I know.

SPMD : Paralelismo iterativo

Van a haber varios procesos que todos van a ejecutar el mismo proceso pero con distintos sectores de datos ( algo así como recorrer un vector con hilos). La diferente evaluación de un predicado en sentencias condicionales permite que cada nodo toma distintos caminos del programa.

Tiene 2 fases: La elección de la distribución de datos y la generación del programa paralelo.

La primera determina el lugar que ocuparan los datos en los nodos. La carga es proporcional al número de datos asignado a cada nodo. Tiene dificultades en computaciones irregulares y máquinas heterogéneas.

La segunda convierte al programa secuencial en SPMD. En la mayoría de los lenguajes depende de la distribución de datos.

Suele implicar un **paralelismo iterativo,** donde un programa consta de un conjunto de procesos los cuales tienen 1 o más loops. Cada proceso es un programa iterativo. Generalmente el dominio de datos se divide entre los procesos siguiendo diferentes patrones.

Imagen que contiene Calendario

Descripción generada automáticamente

Ejemplo con multiplicación de matrices: Son 3 for anidados, no voy a entrar en detalle.

Paralelizado cada elemento resultado de la matriz se podría estar ejecutando independientemente. Lo que hay que tener en cuenta con cuántos procesos contamos, si usáramos N2 procesos ( con una matriz de tamaño n) y hay procesadores más rápidos que otros , esto haría que se desperdicie tiempo, porque unos terminaron mientras que los otros no.

Texto

Descripción generada automáticamente con confianza baja

En la primera cada proceso calcula una fila de la matriz resultado y en la segunda una columna de la matriz resultante.

Diagrama, Texto

Descripción generada automáticamente

En la primera es un co que genera N filas, en la segunda a su vez crea n hilos por la cantidad de columnas.

El problema es que no tenemos n procesadores, por lo que tenemos que hacer que cada proceso ejecute un grupo de filas (solución de grano un poco más grueso).

El grupo óptimo de filas a ejecutar es un problema interesante para balancear el costo de procesamiento con costo de comunicaciones.

Texto

Descripción generada automáticamente