Programación Avanzada (TC2025)

Tema 6. Programación multinúcleo

Instituto Tecnológico y de Estudios Superiores de Monterrey. Campus Santa Fe
Departamento de Tecnologías de Información y Electrónica
Dr. Vicente Cubells (vcubells@itesm.mx)

Temario

- Modelos de programación paralela
  - Memoria compartida
  - Threads
  - Memoria distribuida / Paso de mensajes
  - …
- Diseño de algoritmos paralelos
  - Paralelización manual vs paralelización automática
  - Particionamiento, Comunicaciones, Sincronización, Dependencia de datos, Balance de carga, Granularidad, Entrada/Salida
  - Límites y costo de la programación paralela

Modelos de programación paralela

Modelos de programación paralela…

- Memoria compartida (sin threads)
- Threads
- Memoria distribuida/ Paso de mensajes
- Datos paralelos
- Híbridos
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

Modelos de programación paralela…

Memoria compartida (sin threads)

- Las tareas comparten un espacio de direcciones común, al cual acceden asíncronamente
- Mecanismos como bloqueos / semáforos pueden ser usados para controlar el acceso a la memoria compartida
- Desde el punto de vista del programador la noción de propietario del dato es ambigua, por tanto no hay necesidad de especificar explícitamente la comunicación de datos entre tareas. La programación se simplifica
- Una desventaja importante en términos de performance es la dificultad inherente para comprender y manejar las direcciones de memoria
- Mantener los datos locales al procesador que trabaja en ella conserva los accesos a memoria, actualizaciones de memoria caché y tráfico en el bus que se produce cuando varios procesadores utilizan los mismos datos
- Controlar los datos localmente es difícil de comprender por el usuario promedio

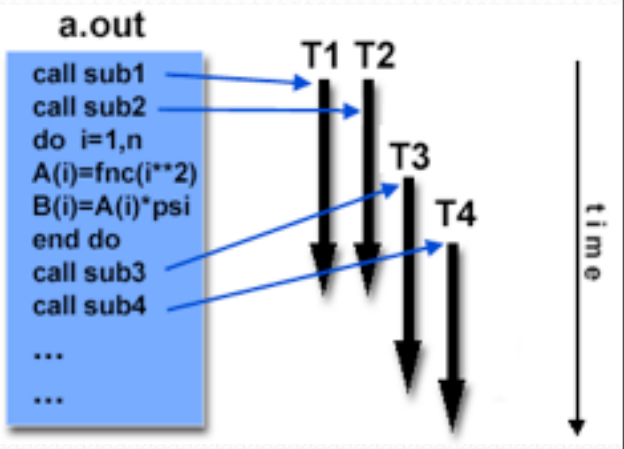
- **Implementaciones:**

  - Compiladores nativos traducen las variables de usuario en direcciones de memoria real, las cuales son globales. En máquinas SMP, esta tarea es sencilla de realizar.
  - En máquinas de memoria compartida distribuida, tales como SGI Origin, la memoria se distribuye físicamente a través de una red de computadoras, pero se hace global mediante el uso de hardware y software especializado

Modelos de programación paralela…

Modelo de Threads

- Es un tipo de programación de memoria compartida
- Un proceso puede tener múltiples caminos de ejecución concurrentes
- Una analogía simple: Un programa que incluye un número de funciones:
- El programa principal **a.out** se planifica para ejecutarse por el SO nativo. a.out obtiene todos los recursos de usuario y del sistema que necesita para ejecutarse.
- a.out realiza algún trabajo en serie y después crea un número de tareas (threads) que pueden planificarse para ejecutarse concurrentemente por el SO.
- Cada hilo tiene datos locales pero comparte los recursos globales de a.out. Elimina la sobrecarga de apropiarse de recursos a nivel de thread. Cada thread comparte el espacio de memoria de a.out.
- Un thread es como una subrutina.
- Los threads se comunican entre sí mediante la memoria global. Esto requiere sincronización.
- Los threads van y vienen, pero a.out permanece presente hasta que la app termina su ejecución.



Modelos de programación paralela…

Modelo de Threads

- **Implementaciones:**
- Desde una perspectiva de programación, las implementaciones comunes de threads son:
  - Una biblioteca de funciones
  - Un conjunto de directivas del compilador
- En ambos casos el programador es el responsable de determinar el paralelismo

- Los threads no son nuevos, históricamente cada fabricante de hardware ha implementado su propia versión. Estas versiones difieren sustancialmente y dificultan el desarrollo de código portable
- Los esfuerzos de estandarización han resultado en dos implementaciones muy diferentes: **POSIX Threads** y **OpenMP**

Modelos de programación paralela…

Modelo de Threads

POSIX Threads

- Biblioteca de funciones
- Especificado por el estándar IEEE POSIX 1003.1c (1995)
- Solo disponible para C
- Conocido como Pthreads
- La mayoría de los fabricantes de hardware ofrecen estos en adición a su implementación propietaria
- Paralelismo muy explícito; requiere una atención especial del programador

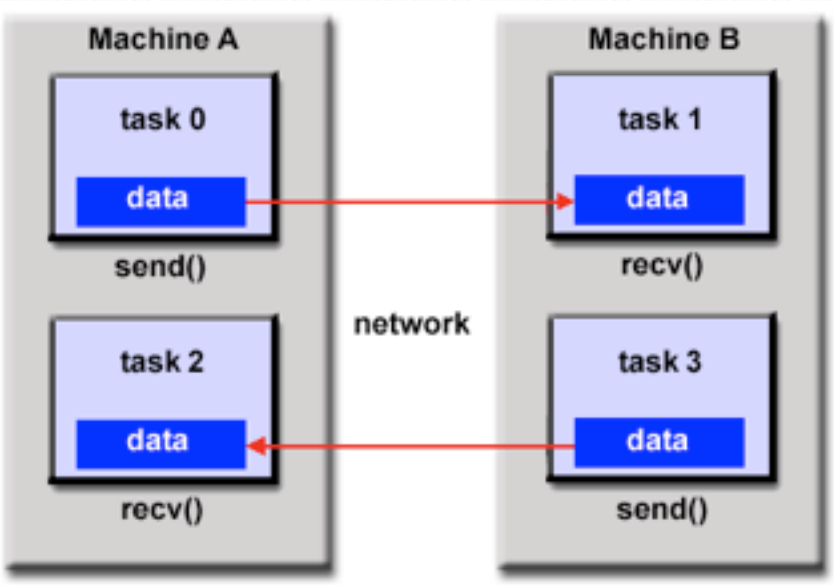
OpenMP

- Basado en directivas del compilador; puede usar código en serie
- Definido por varias compañías.
- OpenMP Fortran API fue liberada en octubre de 1997
- C/C++ API fue liberada a finales de 1998
- Portable / multiplataforma (Unix y Windows NT)
- Puede ser fácil y simple de usar

Modelos de programación paralela…

Modelo de Memoria distribuida / Paso de mensajes

- Un conjunto de tareas usa su propia memoria local durante el cómputo
- Múltiples tareas pueden residir en la misma máquina física o en un número arbitrario de máquinas
- Las tareas intercambian datos a través de comunicaciones mediante el envío y recepción de mensajes
- La transferencia de datos requiere operaciones cooperativas entre los procesos. Por ejemplo: un envío debe tener siempre una operación de recepción



Modelos de programación paralela…

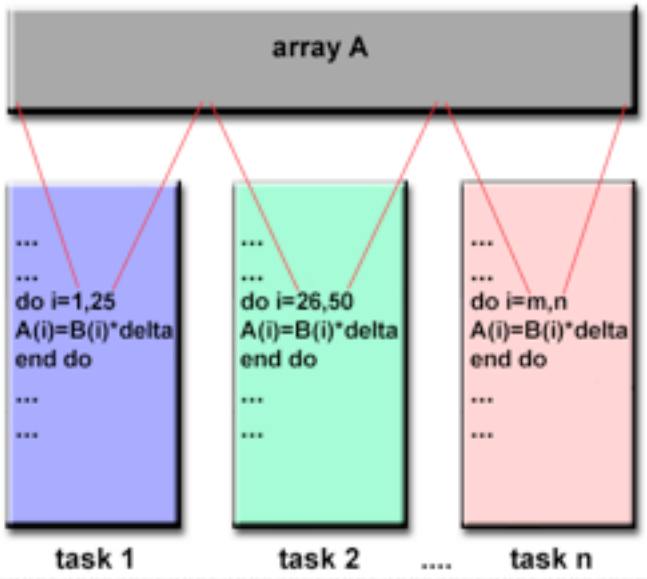
Modelo de Memoria distribuida / Paso de mensajes

- **Implementaciones:**
- Desde una perspectiva de programación, usualmente consiste en una biblioteca de funciones. Las llamadas a dichas funciones son incluidas en el código fuente
- El programador es el responsable de todo paralelismo
- Varias implementaciones que difieren entre sí y dificultan el desarrollo de aplicaciones portables
- En 1992, el MPI Forum se dió a la tarea de establecer un estándar
- Parte 1 de **Message Passing Interface (MPI)** fue liberada en 1994. Parte 2 (MPI-2) fue liberada en 1996.
- MPI es el estándar "de facto"
- Existen implementaciones para todas las plataformas populares de cómputo paralelo
- No todas las implementaciones incluyen todo lo de ambas partes (MPI-1 y MPI-2)

Modelos de programación paralela…

Modelo de Datos paralelos

- La mayor parte del trabajo paralelo se centra en las operaciones sobre los datos. Los datos se organizan en estructuras (arreglos, cubos)
- Un conjunto de tareas trabaja colectivamente sobre la misma estructura de datos, sin embargo, cada tarea tiene una porción diferente de los datos
- Todas las tareas realizan la misma operación sobre su partición, por ejemplo, ”adicionar 4 a cada elemento del arreglo”



- En arquitecturas de memoria compartida, todas las tareas pueden tener acceso a los datos mediante la memoria global
- En arquitecturas de memoria distribuida, la estructura de datos se divide y reside como “fragmentos” en la memoria local de cada tarea.

Modelos de programación paralela…

Modelo de Datos paralelos

- **Implementaciones:** Programar con el modelo de datos paralelos usualmente incluye escribir programas con construcciones paralelas (bibliotecas de funciones o directivas de compilador)

Fortran 90 and 95 (F90, F95):
Extensiones ISO/ANSI a Fortran 77.

- Contiene todo lo de Fortran 77
- Nuevo formato del código fuente; adiciones al conjunto de caracteres
- Adiciones a la estructura del programa y los comandos
- Métodos y argumentos
- Apuntadores y memoria dinámica
- Procesamiento de arreglos como objetos
- Recursividad
- …

High Performance Fortran (HPF):
Extensiones a Fortran 90 para soportar paralelismo.

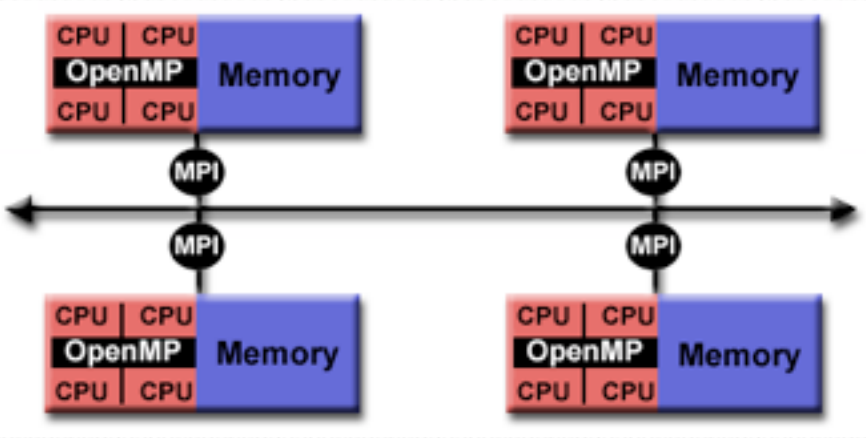
- Contiene todo lo de Fortran 90
- Directivas para decirle al compilador como distribuir los datos
- Optimizaciones
- Construcciones de datos paralelos

- Ya no se implementa

Modelos de programación paralela…

Modelo Híbrido

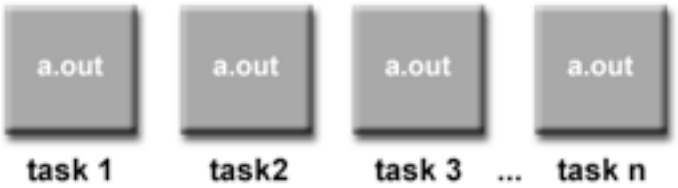
- Combina más de uno de los modelos anteriores
- Un ejemplo común es la combinación de MPI con OpenMP
  - Los threads realizan el cómputo intensivo utilizando datos locales
  - Las comunicaciones entre procesos de diferentes nodos ocurren mediante una red utilizando MPI
- Otro ejemplo es el uso de MPI con GPU (Graphics Processing Unit)
  - GPUs realizan el cómputo intensivo utilizando datos locales
  - Las comunicaciones entre procesos de diferentes nodos ocurren mediante una red utilizando MPI
- Este modelo híbrido se presta bien para el entorno de hardware cada vez más común: clúster de máquinas multi-núcleos



Modelos de programación paralela…

Modelo Single Program Multiple Data (SPMD):

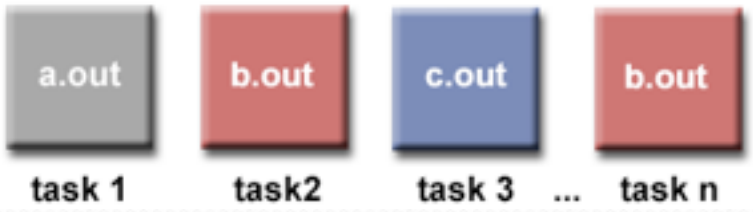
- Modelo de programación de alto nivel que puede construirse con la combinación de los modelos anteriores
- SINGLE PROGRAM: Todas las tareas ejecutan su copia del mismo programa, simultáneamente. El programa puede utilizar threads, paso de mensajes, datos paralelos o un híbrido
- MULTIPLE DATA: Todas las tareas pueden usar datos diferentes
- Programas SPMD suelen tener la lógica necesaria programada en ellos para permitir que las tareas no ejecuten necesariamente todo el programa – quizás sólo una parte



Modelos de programación paralela

Modelo Multiple Program Multiple Data (MPMD):

- Como SPMD, es un modelo de programación de alto nivel que puede construirse con la combinación de los modelos anteriores
- MULTIPLE PROGRAM: Las tareas pueden ejecutar diferentes programas simultáneamente. Los programas puede utilizar threads, paso de mensajes, datos paralelos o un híbrido
- MULTIPLE DATA: Todas las tareas pueden usar datos diferentes
- Las aplicaciones MPMD no son tan comunes como SPMD pero pueden ser mejores para determinados tipos de problemas



Diseño de programas paralelos

Paralelización manual vs automática…

Paralelización manual vs automática…

- Tradicionalmente, el programador ha sido el responsable de implementar el paralelismo
  - Es un proceso iterativo
  - Propenso a errores
  - Complejo
  - Gran consumo de tiempo

Paralelización manual vs automática…

- Tradicionalmente, el programador ha sido el responsable de implementar el paralelismo
  - Es un proceso iterativo
  - Propenso a errores
  - Complejo
  - Gran consumo de tiempo
- Existen herramientas de apoyo
  - Compilador de compiladores
  - Pre-procesadores

Paralelización manual vs automática…

Paralelización manual vs automática…

- Un compilador de compiladores trabaja de dos formas
  - Completamente automático
    - El compilador analiza el código para identificar oportunidades para el paralelismo
    - El análisis incluye la identificación de inhibidores del paralelismo y posiblemente, una ponderación de costos sobre si o no el paralelismo mejoraría realmente el rendimiento
    - Los ciclos (do, for) son el objetivo más frecuente para la paralelización automática
  - Dirigido por el programador
    - Usando directivas o banderas del compilador, explícitamente el programador le indica al compilador como paralelizar el código
    - Puede usarse en conjunto con algún grado de paralelización automática

Paralelización manual vs automática

- Si se empieza con un código secuencial existente y existen restricciones de tiempo o de presupuesto, la paralelización automática puede ser la respuesta
- Sin embargo, hay elemento a tener en cuenta:
  - Pueden producirse resultados erróneos
  - El performance puede degradarse
  - Es menos flexible que la paralelización manual
  - Está limitada a un subconjunto del código (ciclos)
  - Puede no paralelizar el código si el análisis sugiere la existencia de inhibidores o el código es muy completo

Entendiendo el problema y el programa…

- El primer paso: comprender el problema
  - Si se parte de código secuencial, comprender el programa existente
- Antes de perder tiempo, analizar el problema
  - Paralelizable: Calcular la energía potencial de cada uno de los varios miles de conformaciones independientes de una molécula. Cuando haya terminado, buscar la conformación de energía mínima.
  - No paralelizable: F(n) = F(n-1) + F(n-2)

Entendiendo el problema y el programa

Entendiendo el problema y el programa

- Identificar las partes del programa que hacen un uso intensivo de la CPU y centrarse en ellas
  - Herramientas de análisis pueden ser de ayuda

Entendiendo el problema y el programa

- Identificar las partes del programa que hacen un uso intensivo de la CPU y centrarse en ellas
  - Herramientas de análisis pueden ser de ayuda
- Identificar los cuellos de botella
  - Áreas desproporcionalmente lentas o que causan qu el trabajo paralelo se detenga o se posponga
    - Ejemplo: E/S
  - Puede ser posible reestructurar el programa o usar u algoritmo diferente para reducir o eliminar áreas lentas innecesarias

Entendiendo el problema y el programa

- Identificar las partes del programa que hacen un uso intensivo de la CPU y centrarse en ellas
  - Herramientas de análisis pueden ser de ayuda
- Identificar los cuellos de botella
  - Áreas desproporcionalmente lentas o que causan qu el trabajo paralelo se detenga o se posponga
    - Ejemplo: E/S
  - Puede ser posible reestructurar el programa o usar u algoritmo diferente para reducir o eliminar áreas lentas innecesarias
- Identificar inhibidores de paralelismo
  - Una clase común de inhibidor es la “dependencia de datos”

Entendiendo el problema y el programa

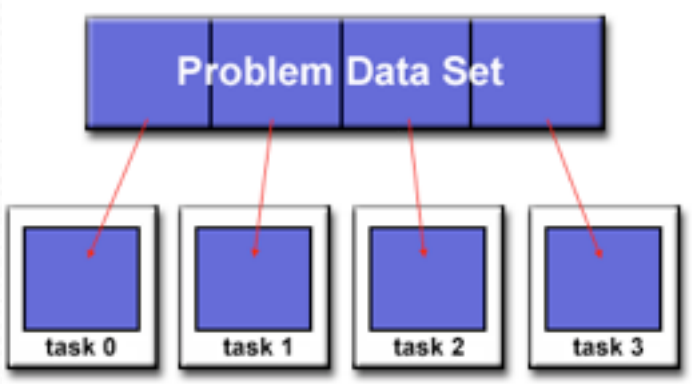
- Identificar las partes del programa que hacen un uso intensivo de la CPU y centrarse en ellas
  - Herramientas de análisis pueden ser de ayuda
- Identificar los cuellos de botella
  - Áreas desproporcionalmente lentas o que causan qu el trabajo paralelo se detenga o se posponga
    - Ejemplo: E/S
  - Puede ser posible reestructurar el programa o usar u algoritmo diferente para reducir o eliminar áreas lentas innecesarias
- Identificar inhibidores de paralelismo
  - Una clase común de inhibidor es la “dependencia de datos”
- Investigar si es posible utilizar otros algoritmos
  - Puede ser la consideración más importante al diseñar una aplicación paralela

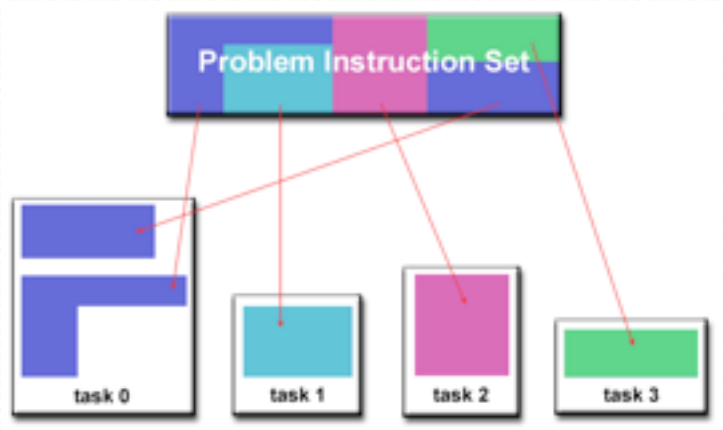
Particionamiento…

Uno de los primeros pasos: descomponer del problema en unidades discretas de trabajo que pueden ser distribuidas en múltiples tareas

Descomposición de dominio

- Los datos se descomponen, cada tarea paralela trabaja en una porción de los datos



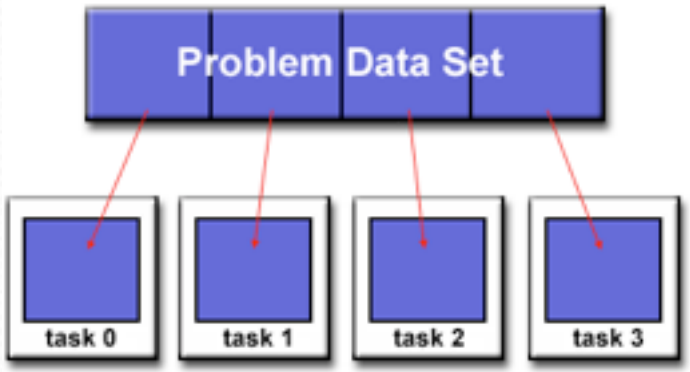


Particionamiento…

Uno de los primeros pasos: descomponer del problema en unidades discretas de trabajo que pueden ser distribuidas en múltiples tareas

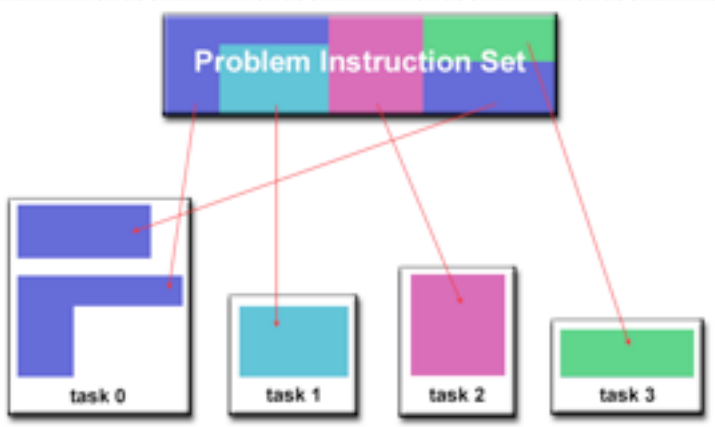
Descomposición de dominio

- Los datos se descomponen, cada tarea paralela trabaja en una porción de los datos



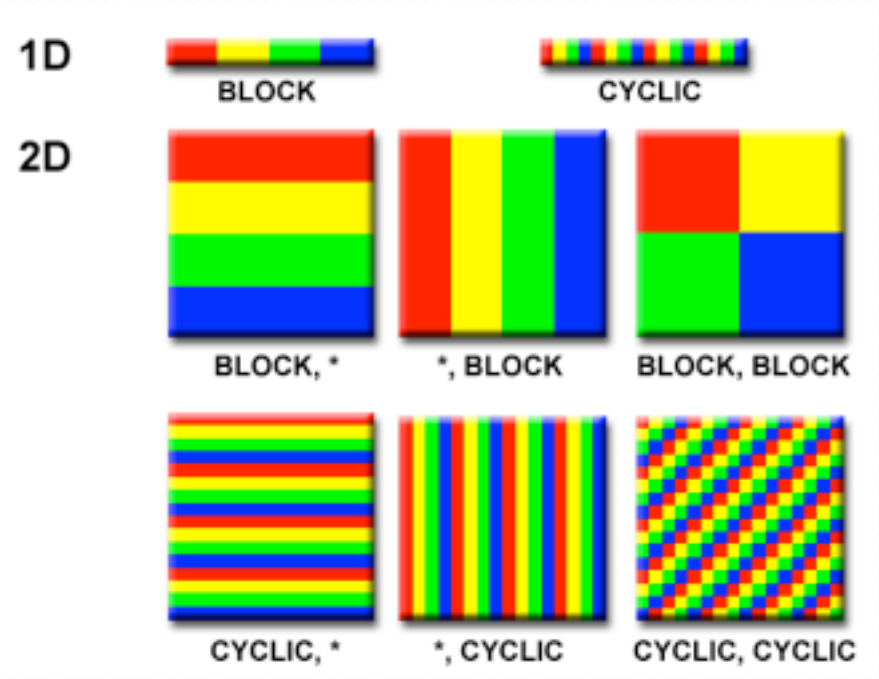
Descomposición funcional

- Se centra en el cómputo ante que en los datos. El cómputo se descompone en tareas



Particionamiento…

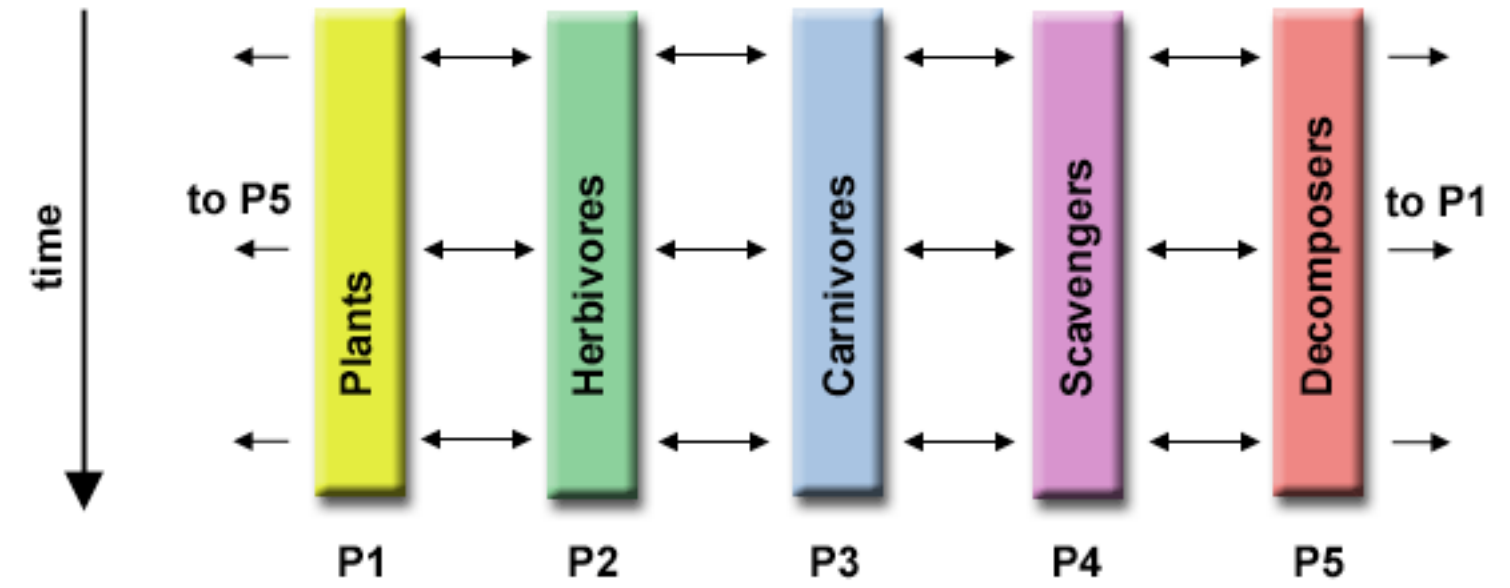
- Descomposición de dominio: Los datos se descomponen, cada tarea paralela trabaja en una porción de los datos



Particionamiento…

- Descomposición funcional: Se centra en el cómputo ante que en los datos. El cómputo se descompone en tareas

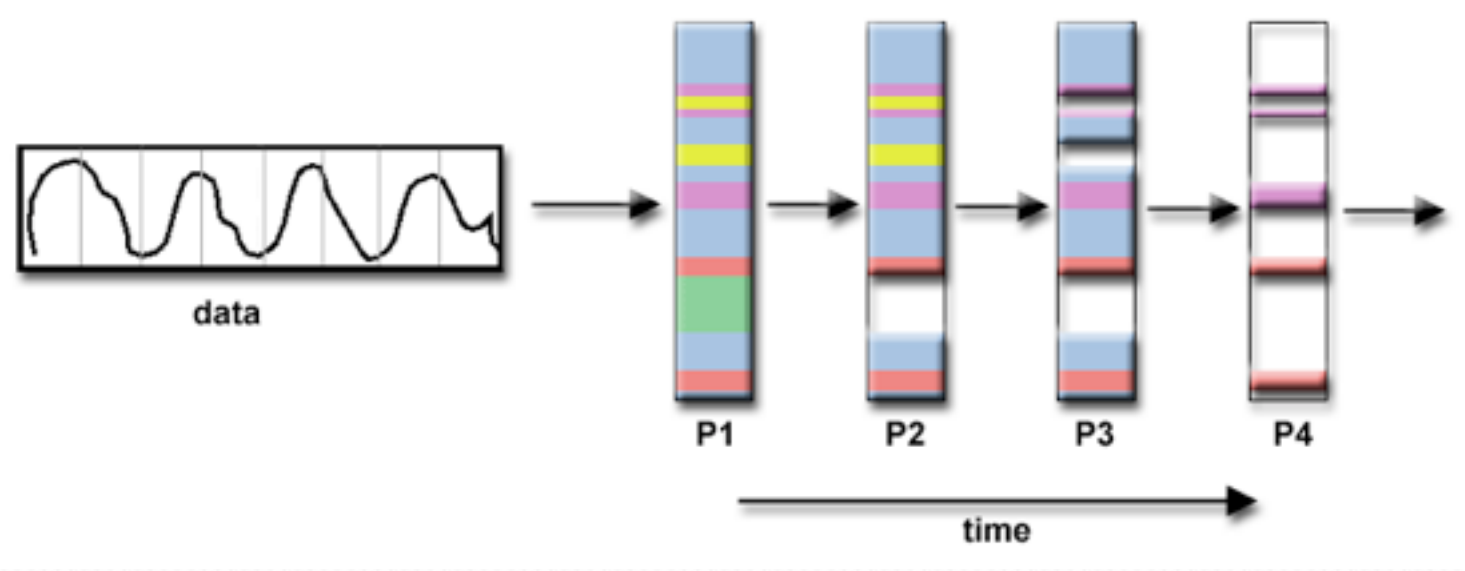
Modelación de un ecosistema



Particionamiento

- Descomposición funcional: Se centra en el cómputo ante que en los datos. El cómputo se descompone en tareas

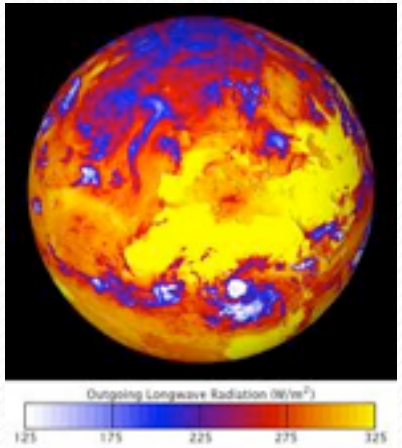
Procesamiento de señales de audio



Comunicaciones…

- ¿Cuándo se necesitan?
  - Depende del problema
    - No: Procesamiento gráfico de una imagen en blanco y negro a nivel de pixel

    - Si: Difusión de calor 3D

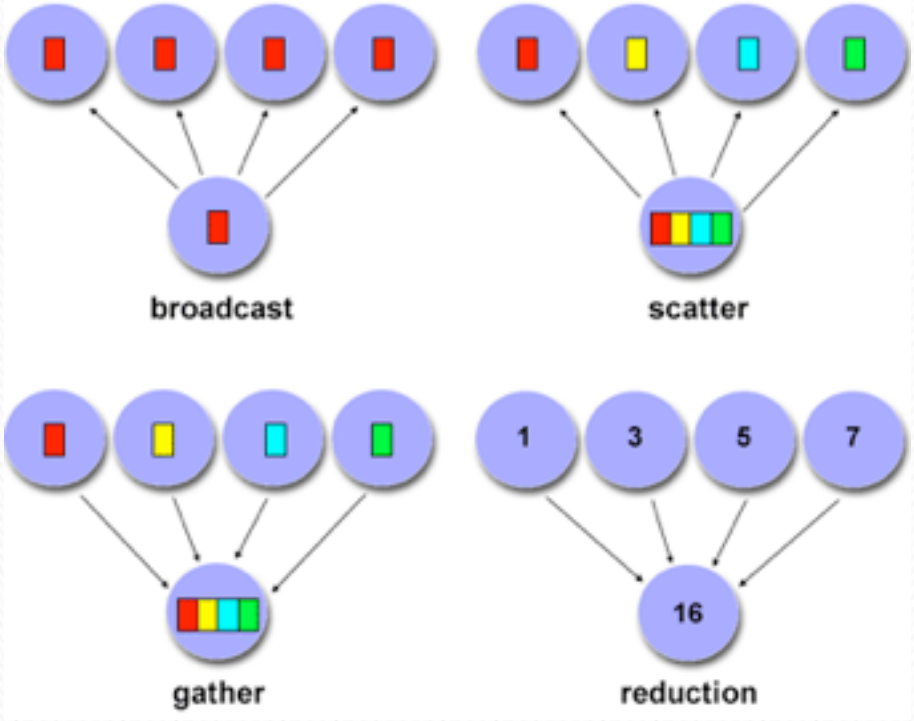


Comunicaciones…

- Factores a considerar
  - Costo de las comunicaciones
    - ciclos de reloj desperdiciados, tiempo de sincronización, saturación de los enlaces
  - Latencia vs. Ancho de banda
    - ¿Muchos mensajes pequeños o uno grande?
  - Visibilidad al programador
  - Comunicaciones síncronas vs. asíncronas
    - Con bloqueos y sin bloqueos
  - Mecanismos de comunicación
    - Broadcast, multicast, scatter, gather, reduction
  - Eficiencia
  - Sobrecarga y complejidad

Comunicaciones

- Mecanismos de comunicación
  - Broadcast, multicast, scatter, gather, reduction



Sincronización

- Tipos de sincronización
  - **Barreras**
    - Todas las tareas están involucradas
    - Cada tarea realiza su trabajo hasta llegar a la barrera, entonces se detiene en espera de las demás
    - Cuando la última alcanza la barrera, todas las tareas son sincronizadas
    - Luego depende
  - **Candados / semáforos**
    - Puede involucrar cualquier número de tareas
    - Usado para serializar el acceso a datos compartidos
    - Si una tarea estableció un candado, las otras deben esperar a que lo libere
    - Pueden ser compartidos o exclusivos
  - Comunicaciones síncronas

Dependencia de datos

- El orden de ejecución de las operaciones afecta el resultado
- Es un inhibidor del paralelismo

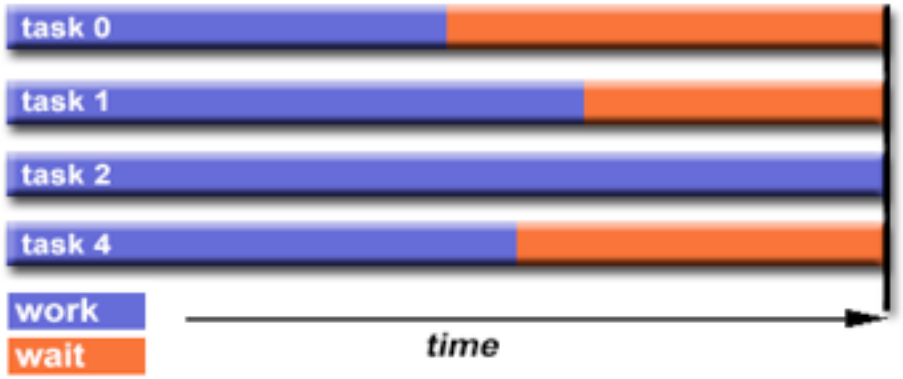
Ejemplo

```
for (int i=0; i < n; ++i)
{
    A[i] = A[i-1] * b;
}
```

¿Cómo manejar la dependencia de datos?

Balance de carga

- Distribuir el trabajo entre todas las tareas para mantenerlas ocupadas todo el tiempo

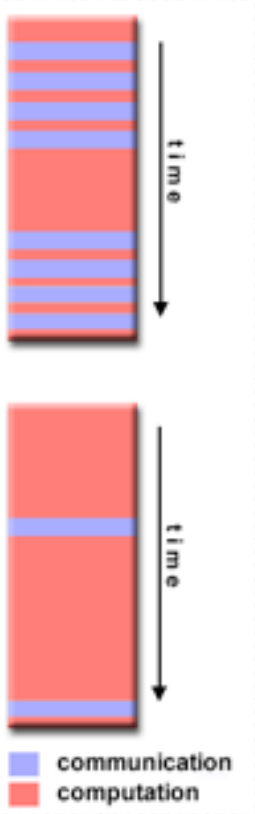


Repartir el trabajo equitativamente
En operaciones matriciales o sobre arreglos, donde cada tarea realiza un trabajo similar, distribuir el conjunto de datos entre todas las tareas
En iteraciones cíclicas, distribuir las iteraciones
En un entorno mixto, utilizar herramientas de análisis

Usar asignación dinámica del trabajo
Ciertos problemas provocan desbalance (Matrices dispersas)
Cuando el trabajo que debe realizar cada tarea es variable o no se puede predecir, utilizar un planificador
Puede ser necesario diseñar un algoritmo que detecte desbalances e incorporarlo dentro del código

Granularidad

- Medida cualitativa
  - Computación / Comunicación
- Grano fino:
  - Poco cómputo entre eventos de comunicación/sincronización
  - Facilita el balance
  - Implica sobrecarga en las comunicaciones
  - Difícil de mejorar el performance
- Grano grueso:
  - Muchos trabajos de cómputo entre eventos de comunicación/sincronización
  - Mas oportunidades de incrementar el performance
  - Difícil de balancear la carga



Entrada / Salida

- En contra
  - Las operaciones E/S son un inhibidor del paralelismo
  - Sistemas de E/S paralelos no están disponibles para todas las plataformas
  - Pueden producirse sobreescrituras de archivos
  - Cuando involucran comunicaciones (NFS) pueden ocasionar cuellos de botella
- A favor
  - Sistemas de arhivos paralelos
    - GPFS: General Parallel File System for AIX (IBM)
    - Lustre: for Linux clusters (Oracle)
    - PVFS/PVFS2: Parallel Virtual File System for Linux clusters (Clemson/Argonne/Ohio State/others)
    - PanFS: Panasas ActiveScale File System for Linux clusters (Panasas, Inc.)
    - HP SFS: HP StorageWorks Scalable File Share.
  - MPI
  - Algunas consideraciones
    - Usar sistemas temporales (/tmp)
    - Acceso a sistemas de archivos paralelos

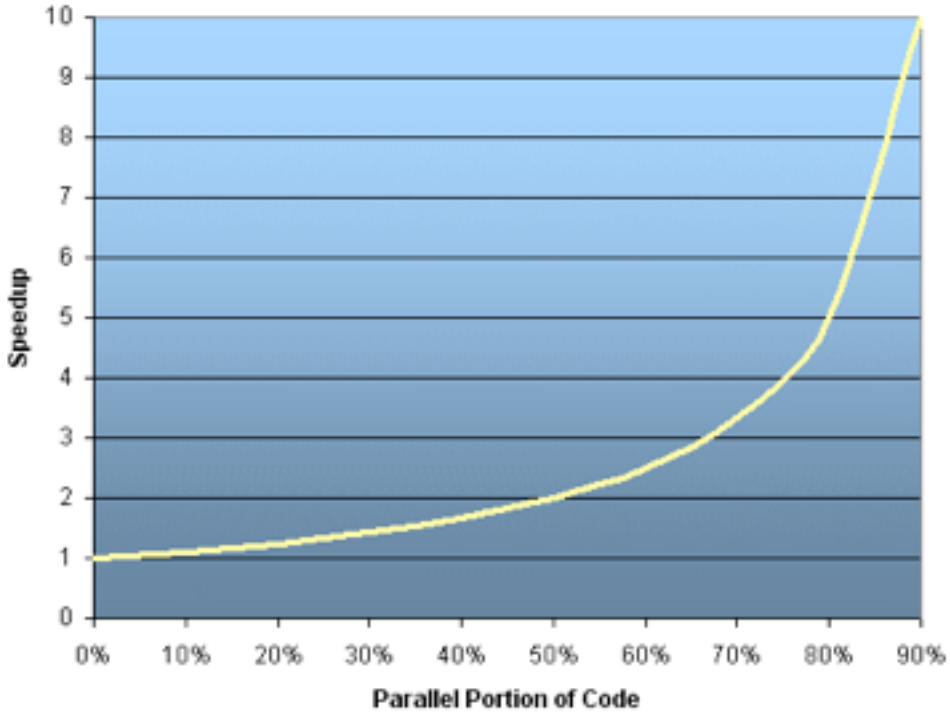
Reducir E/S

Límites y costo…

- Ley de Amdahl: El speedup de un programa está definido por la fracción del código (P) que puede ser paralelizada

$$speedup = \frac{1}{1 - P}$$

$$0 <= P <= 1$$



Límites y costo…

- Si se adiciona a la ecuación los procesadores que realizan la fracción de código paralelo:

$$speedup = \frac{1}{\frac{P}{N} + S}$$

$0 <= P <= 1$

N: Procesadores
S: Fracción secuencial

Límites y costo…

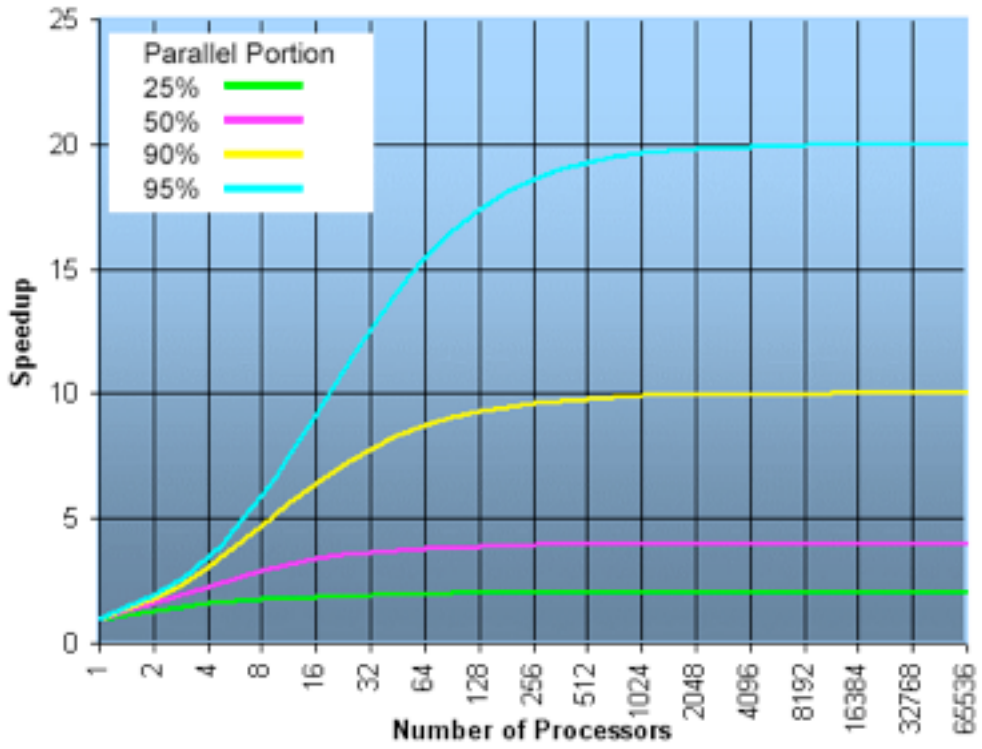
- Si se adiciona a la ecuación los procesadores que realizan la fracción de código paralelo:

$$speedup = \frac{1}{\frac{P}{N} + S}$$

$0 <= P <= 1$

N: Procesadores
S: Fracción secuencial

	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02
100000	1.99	9.99	99.90



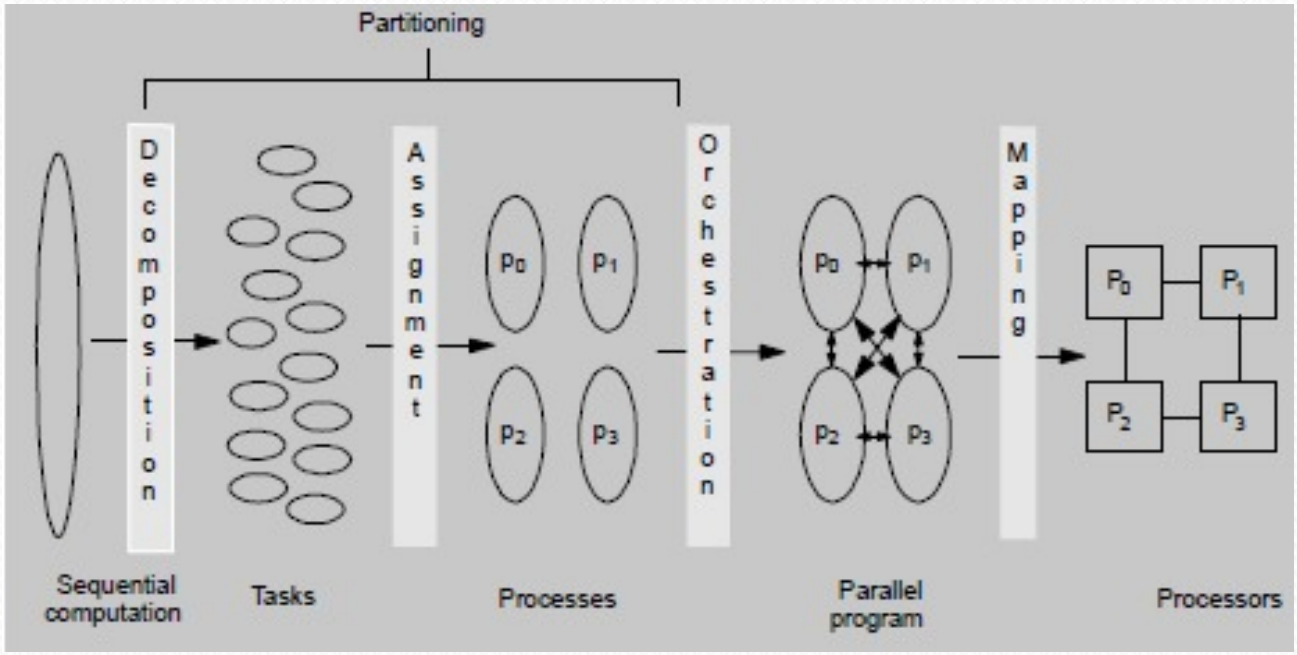
Límites y costo…

- Complejidad: El costo de la complejidad se mide en tiempo del programador durante todo el ciclo de desarrollo del software
  - Seguir buenas prácticas de ingeniería de software es esencial
- Portabilidad: Hoy no es un problema tan grave, sin embargo
  - Los mismos problemas que se presentan en programas secuenciales (construcciones propias de un lenguaje)
  - APIs estándares pero implementaciones diferentes
  - Sistemas operativos
  - Arquitecturas de hardware

Límites y costo

- Requerimientos de recursos: Decrementar los tiempos de ejecución implica mayor consumo de CPU
  - Se requiere más memoria
  - En programas pequeños existen una sobrecarga
- Escalabilidad: El incremento del performance es el resultado de un conjunto de factores interrelacionados
  - ¿Adicionar más computadoras?
  - Los algoritmos pueden no ser escalables. Adicionar recursos no suele ser producente
  - El hardware es importante (bus, CPU, redes, etc.)
  - Las librerías existentes pueden ser una limitante

Pasos en la creación de un programa paralelo



- 4 pasos: Descomposición, Asignación, Orquestación, Mapeo
  - Hecho por el programador o software del sistema(compilador o SO)
  - No hay palalelización automática, responsabilidad del programador

Paso 1. Descomposición

- Dividir el cálculo en tareas para ser asignadas a los procesos
  - Las tareas pueden estar disponibles dinámicamente
  - Número de tareas disponibles puede variar con el tiempo
- Objetivo: Suficientes tareas para mantener los procesos ocupados todo el tiempo, pero no demasiado
  - Número de tareas disponibles a la vez es el límite superior en aceleración que se puede lograr

Paso 2. Asignación

- Especificar mecanismo para dividir trabajo entre procesos
  - Balancear carga de trabajo (cálculos, E/S y comunicación)
- Se puede particionar con:
  - Inspección de código (ciclos paralelos) o entendimiento de la aplicación
  - Técnicas conocidas
  - Asignación Estática vs Dinámica
- Particionar: pasos más importantes al paralelizar
  - Generalmente independientes de arquitectura o modelo de programación
  - El costo y complejidad de usar ciertas primitivas puede afectar decisiones

Paso 3. Orquestación

- Objetivos
  - Reducir costo de comunicación y sincronización
  - Organizar tareas, para que tareas dependientes se hagan rápido
  - Reducir el costo del manejo de paralelismo
- Arquitectos deberían proveer primitivas apropiadas que simplifiquen la orquestación

Paso 4. Mapeo

- Después de la orquestación, se tiene programa paralelo
- Dos aspectos de mapeo:
  - Cuáles procesos correrán en el mismo procesador, si es necesario
  - Cuál proceso corre en cuál procesador particular
    - Mapear a una topología de red
- Un extremo: Procesos asignados a procesadores,
- Otro extremo: control de admón de recursos completo al SO
  - El mundo real es entre los dos, el usuario especifica algunos aspectos, el sistema lo puede ignorar
- proceso <-> procesador
  - Núm. procesadores = Núm. procesos, No cambian durante la ejecución

Resumiendo

- Existen varios modelos de programación paralela que pueden utilizar de manera independiente o combinados
- El diseño de todo algoritmo paralelo consta de 4 etapas: descomposición, asignación, orquestación y mapeo
- Hay que tener en cuenta varios factores en el diseño
  - Hay que reconocer los inhibidores del paralelismo (E/S)
  - Las estructuras de control repetitivas son “buenos candidatos” para paralelizar
    - Siempre que no exista una dependencia crítica