ALN - HPC

In. Co.
Facultad de Ingeniería
Universidad de la República

Temario:

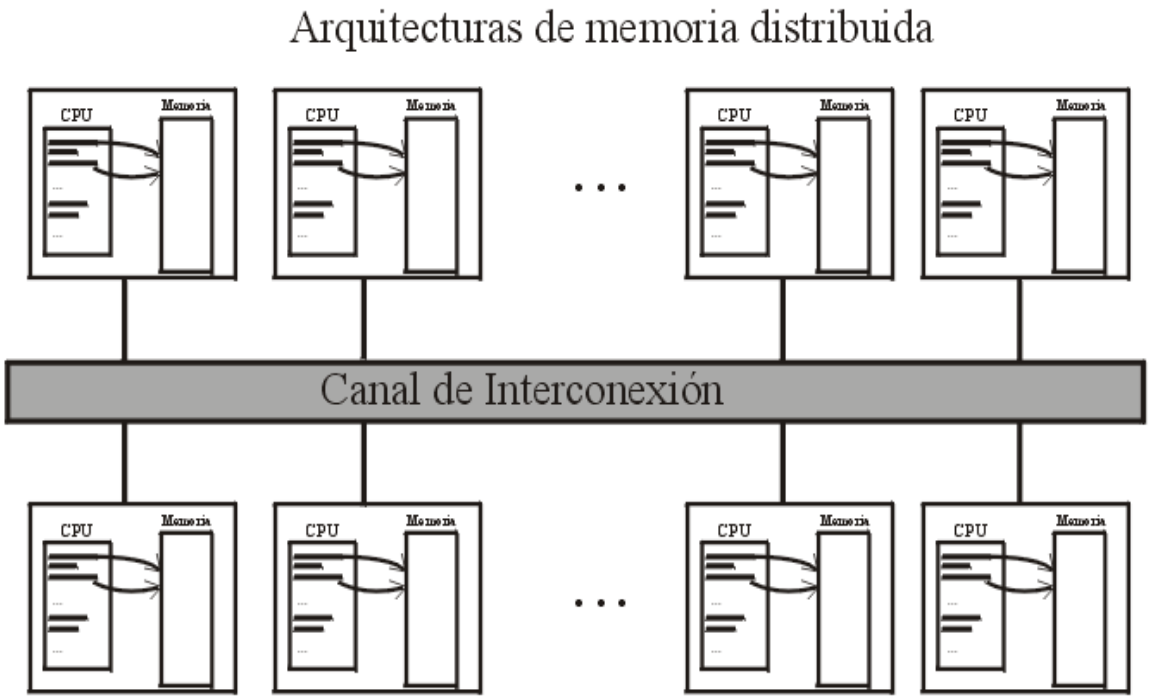
- **Arquitecturas paralelas**
- **Modelo de programación paralela**
- **Técnicas de programación paralela**
- **Medidas de performance**
- **Scheduling y balance de cargas**
- **Herramientas**

Arquitecturas paralelas

- Modelo de la computación estándar : Arquitectura de Von Neumann.
  - CPU única
    - Ejecuta un programa (único).
    - Accede a memoria.
  - Memoria única.
    - Operaciones read/write.
    - Dispositivos.
- Modelo robusto, independiza al programador de la arquitectura subyacente.
- Permitió el desarrollo de las Técnicas de Programación (estándar).

Arquitecturas paralelas

- Extendiendo el modelo a la computación paralela
  - Para lograr abstraer el hardware subyacente.
- Varias alternativas
  - Multicomputador
    - Varios nodos (CPUs de Von Neumann).
    - Un mecanismo de interconexión entre los nodos.



Arquitecturas paralelas

- Extendiendo el modelo a la computación paralela
- Otras alternativas
  - Computador masivamente paralelo.
    - Muchísimos nodos (sencillas CPUs estilo Von Neumann).
    - Topología específica para interconexión entre los nodos.
  - Multiprocesador de memoria compartida.
    - Nodos de Von Neumann.
    - Memoria única.
  - Cluster.
    - Multiprocesador que utiliza una red LAN como mecanismo de interconexión entre sus nodos.

Arquitecturas paralelas

CATEGORIZACION DE FLYNN

		instrucciones	
		SI	**MI**
datos	**SD**	SISD	(MISD)
	MD	SIMD	MIMD

S=single, M=multi, I=Instrucción, D=Datos

- SISD - Modelo convencional de Von Neumann.
- SIMD - Paralelismo de datos, Computación Vectorial.
- MISD - Arrays sistólicos.
- MIMD – Modelo general, varias implementaciones.

Arquitecturas paralelas

- SISD = Máquina de Von Neumann
  - Un procesador capaz de realizar acciones secuencialmente, controladas por un programa el cual se encuentra almacenado en una memoria conectada al procesador.
  - Este hardware esta diseñado para dar soporte al procesamiento secuencial clásico, basado en el intercambio de datos entre memoria y registros del procesador, y la realización de operaciones aritméticas en ellos.

Arquitecturas paralelas
¿ Por qué el modelo SISD no fue suficiente ?

- Los problemas crecieron, o surgió la necesidad de resolver nuevos problemas de grandes dimensiones (manejando enormes volúmenes de datos, mejorando precisión de las grillas, etc.).

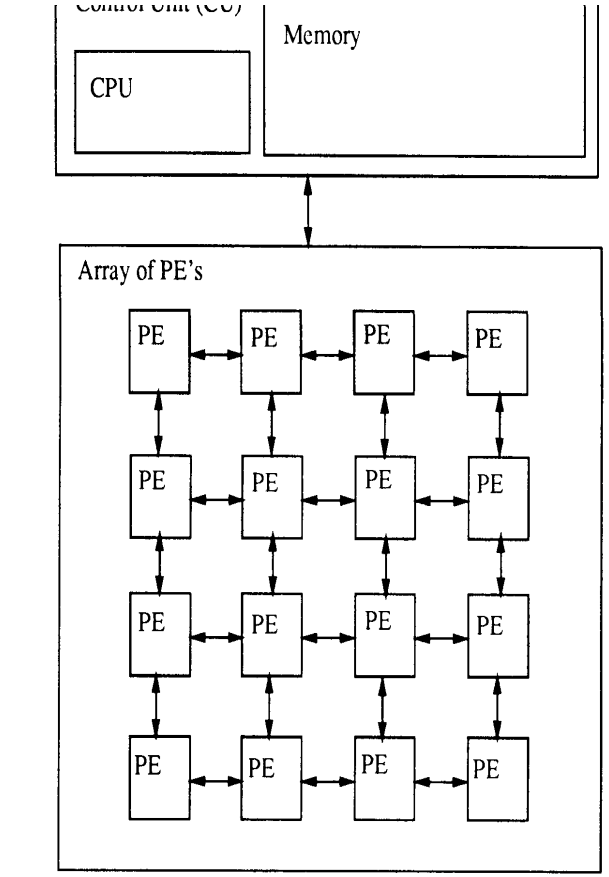
- Si bien las maquinas SISD mejoraron su performance
  - Compiladores optimizadores de código.
  - Procesadores acelerando ciclos de relojes, etc.

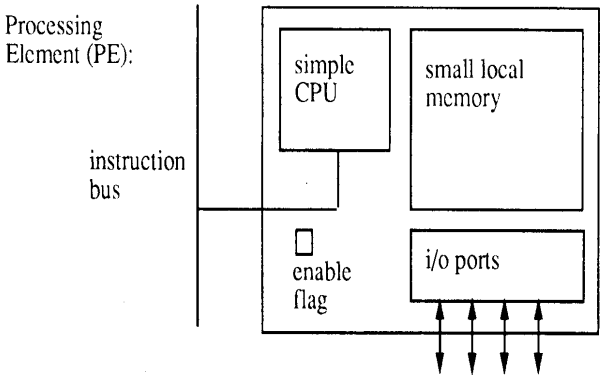
- Aún no fue suficiente, y se prevé que el ritmo de mejoramiento se desacelere (debido a limitaciones físicas)

En este contexto se desarrollaron las maquinas paralelas

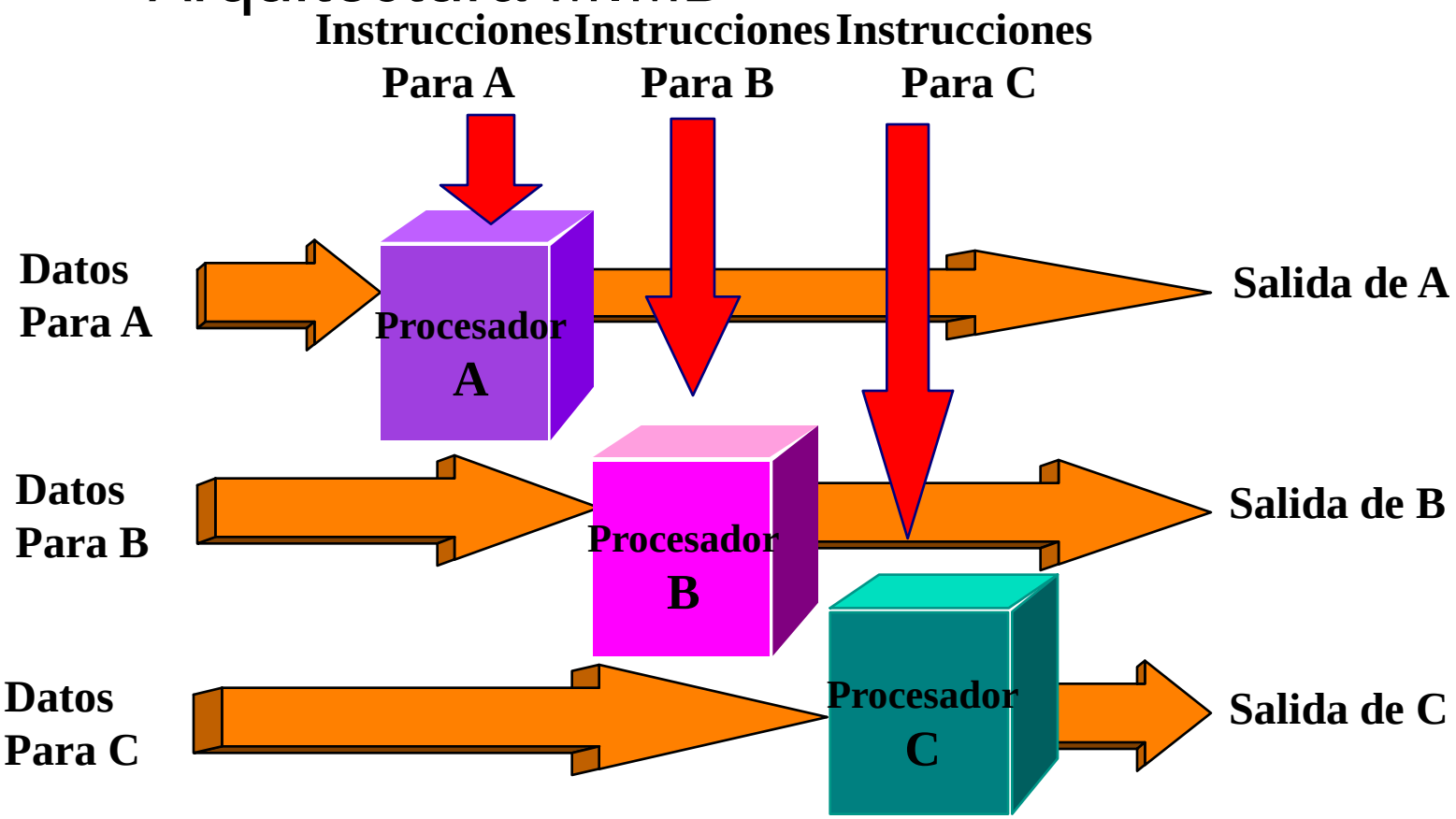
Arquitecturas paralelas

- SIMD
  - Único programa controla los procesadores.
  - Útil en aplicaciones uniformes.
  - Aplicabilidad limitada por las comunicaciones fijas entre procesadores : procesamiento de imágenes, diferencias finitas, etc.





Arquitectura MIMD



A diferencia de SISD y MISD las computadoras MIMD trabajan asincrónicamente.

- MIMD de memoria compartida (fuertemente acopladas)
- MIMD de memoria distribuída (poco acopladas)

Arquitectura MIMD con memoria compartida

COMO SE COMPARTEN LOS DATOS ?

- UMA = Uniform Memory Access
  - Acceso uniforme (todos los procesadores acceden a la memoria en el mismo tiempo).
  - Multiprocesadores simétricos (SMP).
  - Pocos procesadores (32, 64, 128, por problemas de ancho de banda del canal de acceso).
- NUMA = Non-Uniform Memory Access.
  - Colección de memorias separadas que forman un espacio de memoria direccionable.
  - Algunos accesos a memoria son más rápidos que otros, como consecuencia de la disposición física de las memorias (distribuidas físicamente).
  - Multiprocesadores masivamente paralelos (MPP).

Arquitectura MIMD con memoria distribuida

- No existe el concepto de memoria global.
- La comunicación y sincronización se realiza a través del pasaje de mensajes explícitos (mayor costo que en memoria compartida).
- Arquitectura escalable para aplicaciones apropiadas para esta topología (decenas de miles de procesadores).

Conectividad entre procesadores

- FACTORES QUE DETERMINAN LA EFICIENCIA
  - Ancho de banda
    - Número de bits capaces de transmitirse por unidad de tiempo.
  - Latencia de la red
    - Tiempo que toma a un mensaje transmitirse a través de la red.
  - Latencia de las comunicaciones
    - Incluye tiempos de trabajo del software y retardo de la interfaz.
  - Latencia del mensaje
    - Tiempo que toma enviar un mensaje de longitud cero.
  - Valencia de un nodo
    - Número de canales convergentes a un nodo

Conectividad entre procesadores

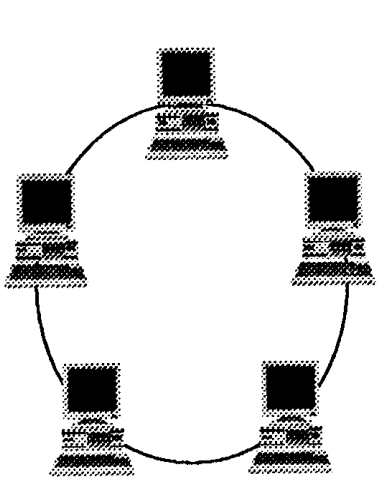
- FACTORES QUE DETERMINAN LA EFICIENCIA
  - Diámetro de la red
    - Número mínimo de saltos entre los nodos más alejados.
    - Permite estimar el peor caso de retardo de un mensaje.
  - Ancho de bisección
    - Número mínimo de enlaces que, en caso de no existir, separarían la red en dos componentes conexas.
  - Largo máximo de un tramo de comunicación.
  - Costo
    - Cantidad de enlaces de comunicación.

Conectividad entre procesadores

- FACTORES QUE DETERMINAN LA EFICIENCIA
  - CONFIGURACIÓN ÓPTIMA
    - Ancho de banda grande.
    - Latencias (de red, comunicación y mensaje) bajas.
    - Diámetro de la red reducido.
    - Ancho de bisección grande.
    - Valencia constante e independiente del tamaño de la red.
    - Largo máximo de tramo reducido, constante e independiente del tamaño de la red.
    - Costo (monetario) mínimo.

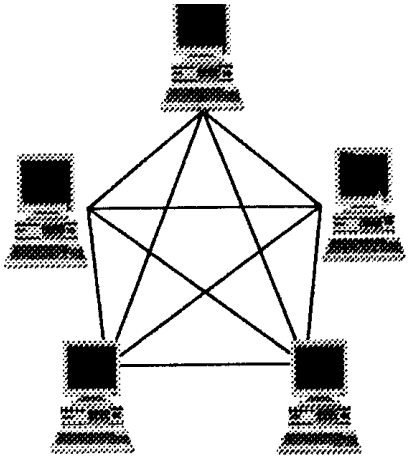
Conectividad entre procesadores

Modelos de conectividad entre procesadores



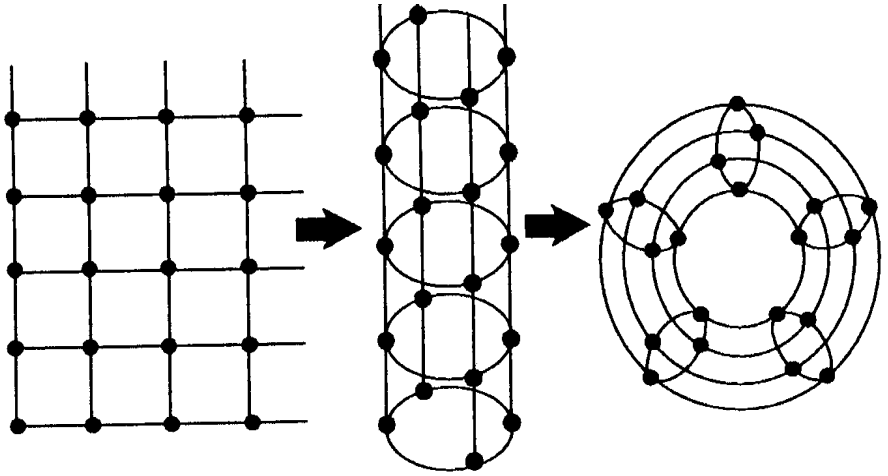
RING

Anillo



STAR

Estrella

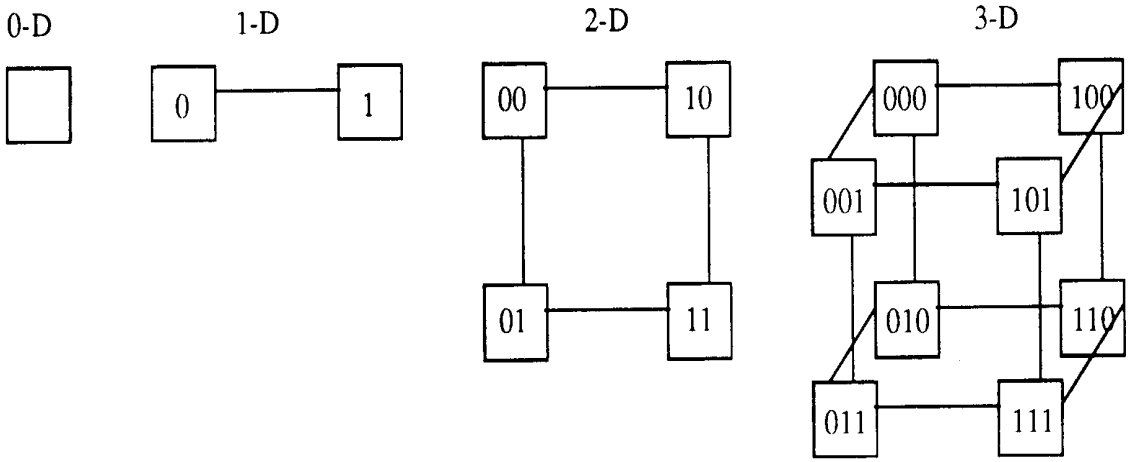


2D CYLINDER TORUS

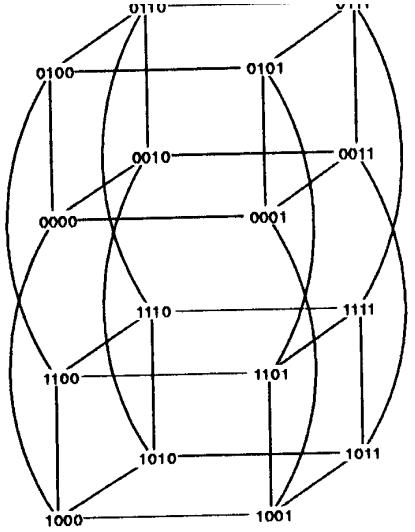
Topologías geométricas (Mallas)

Conectividad entre procesadores

Modelos de conectividad entre procesadores



Topologías “dimensionales” (distancia constante)



Hipercubo

MODELO DE PROGRAMACIÓN PARALELA

Modelo de Programación Paralela

- Programación en máquina de Von Neumann
  - Secuencia de operaciones (aritméticas, read/write de memoria, avance de PC).
  - Abstracciones de datos e instrucciones.
  - Técnicas de programación modular.

- Programación en máquina paralela
  - Incluye complicaciones adicionales.
    - Multiejecución simultánea.
    - Comunicaciones y sincronización.
  - La modularidad pasa a ser fundamental, para manejar la (potencialmente) enorme cantidad de procesos en ejecución simultánea.

Modelo Conceptual de Paralelismo

- El paradigma de diseño y programación difiere del utilizado para diseñar y programar aplicaciones secuenciales.
  - Una buena estrategia de división del problema puede determinar la eficiencia de un algoritmo paralelo.
  - Es importante considerar la disponibilidad de hardware.

- Respecto a los mecanismos de comunicación entre procesos, existen dos paradigmas de computacion paralela
  - MEMORIA COMPARTIDA
  - PASAJE DE MENSAJES

  (Existen también MODELOS HÍBRIDOS, que combinan ambas técnicas)

Niveles de Aplicación del Procesamiento Paralelo

- El paralelismo puede aplicarse :
  - A nivel intrainstrucción (hardware, pipelines).
  - A nivel interinstrucción (SO, optimización de código, compilador).
  - A nivel de procedimientos o tareas (algorítmico – tareas).
  - A nivel de programas o trabajos (algorítmico – funcional).

- Los niveles 3 y 4 de aplicación de las técnicas de paralelismo (enfocado al diseño y programación de algoritmos paralelos).

Modelo Conceptual de Paralelismo

- GRAFOS DIRIGIDOS ACICLICOS (ADGs)

  nodos = tareas o procesos (partes de código que ejecutan secuencialmente)

  aristas = dependencias (algunas tareas preceden a otras)

El problema a resolver se divide en tareas a ejecutar cooperativamente en múltiples procesadores.

Se debe :

Definir tareas que pueden ejecutar concurrentemente.

Lanzar la ejecución y detener la ejecución de tareas.

Implementar los mecanismos de comunicación y sincronización.

Modelo Conceptual de Paralelismo

- No siempre es una buena decisión partir de un algoritmo secuencial (“paralelizar” una aplicación). En ocasiones será necesario diseñar un nuevo algoritmo muy diferente.

- Resumiendo las etapas de diseño de aplicaciones paralelas :
  - Identificar el trabajo que puede hacerse en paralelo.
  - Partir el trabajo y los datos entre los procesadores.
  - Resolver los accesos a los datos, comunicación entre procesos y sincronizaciones.

- DESCOMPOSICIÓN – ASIGNACIÓN – ORQUESTACIÓN – MAPEO

Modelo Conceptual de Paralelismo

- DESCOMPOSICIÓN
  - Identifica concurrencia y decide el nivel y la forma en la cual se explotará.
    - Cantidad de tareas.
    - Tareas estáticas o dinámicas.
    - Criterios de división y utilización de recursos.

- ASIGNACIÓN
  - Asigna datos a procesos
  - Criterios de asignación
    - Estáticos o dinámicos.
    - Balance de cargas.
    - Reducción de costos de comunicación y sincronización.

Modelo Conceptual de Paralelismo

- ORQUESTACIÓN
  - Decisión sobre parámetros de arquitectura, modelo de programación, lenguaje o biblioteca a utilizar.
    - Estructuras de datos, localidad de referencias.
    - Optimizar comunicación y sincronización.

- MAPEO (SCHEDULING)
  - Asignación de procesos a procesadores.
  - Criterios de asignación
    - Performance.
    - Utilización.
    - Reducción de costos de comunicación y sincronización.

Modelo Conceptual de Paralelismo

- Los mecanismos para definir, controlar y sincronizar tareas deben formar parte del lenguaje a utilizar o ser intercalados por el compilador.
- Estos mecanismos deben permitir especificar
  - El control de flujo de ejecución de tareas.
  - El particionamiento de los datos.
- Algunos ejemplos
  - Definición de tareas, lanzado y detención : fork & join (C), parbegin-parend (Pascal concurrente), task (Ada), spawn (PVM).
  - Coordinación y sincronización : Semáforos, monitores, barreras (memoria compartida), mensajes asincrónicos (C, PVM, MPI) y mensajes sincrónicos (rendezvouz de Ada) (memoria distribuida).
  - El particionamiento de los datos es una tarea usualmente asignada al diseñador del algoritmo paralelo.
- El modelo se complica por características particulares de la computación paralela – distribuida.

Problemas con la computación paralela - distribuida

- CONFIABILIDAD
  - Varios componentes del sistema pueden fallar
    - nodos, interfases, tarjetas, caches, bridges, routers, repeaters, gateways, medios físicos.
  - Problemas de utilizar equipamiento no dedicado
    - problemas de uso y tráfico (propio y ajeno), etc.
    - no repetibilidad de condiciones de ejecución.

- NO DETERMINISMO EN LA EJECUCIÓN

  los programas tienen muchos estados posibles: adiós a "diagramas de flujo" para especificar secuencias de control

Problemas con la computación paralela - distribuida

- SEGURIDAD
  - Acceso a equipos remotos.
  - Datos distribuidos.
- DIFICULTAD DE ESTIMAR LA PERFORMANCE FINAL DE UNA APLICACIÓN
  - Como consecuencia del no determinismo en la ejecución.
  - Criterios estadísticos.
- DIFICULTAD DE TESTEAR / DEBUGGEAR PROGRAMAS
  - difícil reproducir escenarios (múltiples estados, asincronismo).
  - datos y procesos no centralizados (un debugger en cada nodo).

Problemas con la computación paralela - distribuida

- INCOMPATIBILIDADES ENTRE PRODUCTOS Y PLATAFORMAS (para sistemas heterogéneos)
  - Sistemas Operativos: AIX, Solaris, Linux, W2K, NT, OSF, VMS, etc.
  - Protocolos de comunicaciones: TCP/IP, DEC-NET, IPX, NetBEUI, etc....
  - Herramientas de software: PVM, MPI, C, HPF, etc....

TÉCNICAS DE PROGRAMACIÓN PARALELA

Técnicas de programación paralela

Introducción

Técnica de descomposición de dominio

Técnica de descomposición funcional

Técnicas Híbridas

Pipeline

Introducción

- Analizaremos las técnicas de DESCOMPOSICIÓN o PARTICIONAMIENTO, que permiten dividir un problema en subproblemas a resolver en paralelo.

- El objetivo primario de la descomposición será dividir en forma equitativa tanto los cálculos asociados con el problema como los datos sobre los cuales opera el algoritmo.

Introducción

- ¿ Cómo lograr el objetivo de la descomposición ?
  - Definir al menos un orden de magnitud más de tareas que de procesadores disponibles (utilización de recursos)
  - Evitar cálculos y almacenamientos redundantes.
  - Generar tareas de tamaño comparable.
  - Generar tareas escalables con el tamaño del problema.
  - Considerar varias alternativas de descomposición, en caso de ser posible.
- Según se enfoque principalmente en la descomposición de datos o de tareas, resultará una técnica diferente de programación paralela.

Descomposición de Dominio

- Se concentra en el particionamiento de los datos del problema (data parallel).
  - Se trata de dividir los datos en piezas pequeñas, de (aproximadamente) el mismo tamaño.
- Luego se dividen los cálculos a realizar
  - Asociando a cada operación con los datos sobre los cuales opera.
- Los datos a dividir pueden ser
  - La entrada del programa.
  - La salida calculada por el programa.
  - Datos intermedios calculados por el programa.

Descomposición de Dominio

- Si bien no existe una regla general para determinar como realizar la división de datos, existen algunas sugerencias obvias dadas por:
  - La estructura o “geometría” del problema.
  - La idea de concentrarse primero en las estructuras de datos más grandes o las accedidas con más frecuencia.

Descomposición de Dominio

- La técnica de descomposición de dominio se asocia comúnmente con la estrategia de “Divide & Conquer” y los modelos SIMD y SPMD de programas paralelos.

- Ejemplo 2 : Resolución de ecuaciones
  - Discretización de la solución.
  - División de dominios de cálculo.
  - Mismo programa en cada dominio.
  - Comunicación necesaria para cálculos en los bordes.

Descomposición de Dominio

Aplicable en modelos de programa SIMD y SPMD.

SIMD: Single Instruction Multi Data

SPMD: Single Program Multi Data

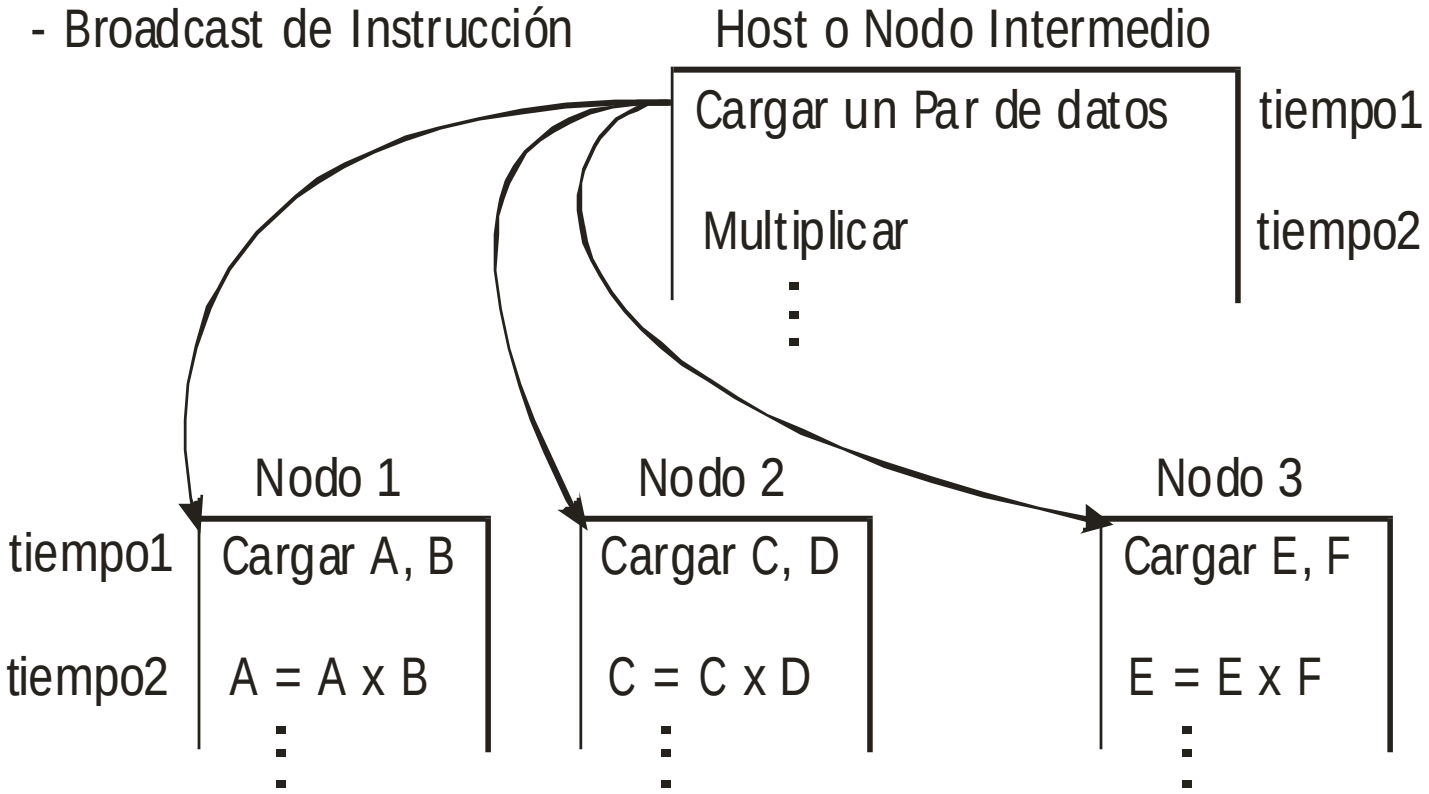
SIMD sobre arquitecturas de memoria compartida.

SPMD sobre arquitecturas de memoria distribuida.

Descomposición de Dominio

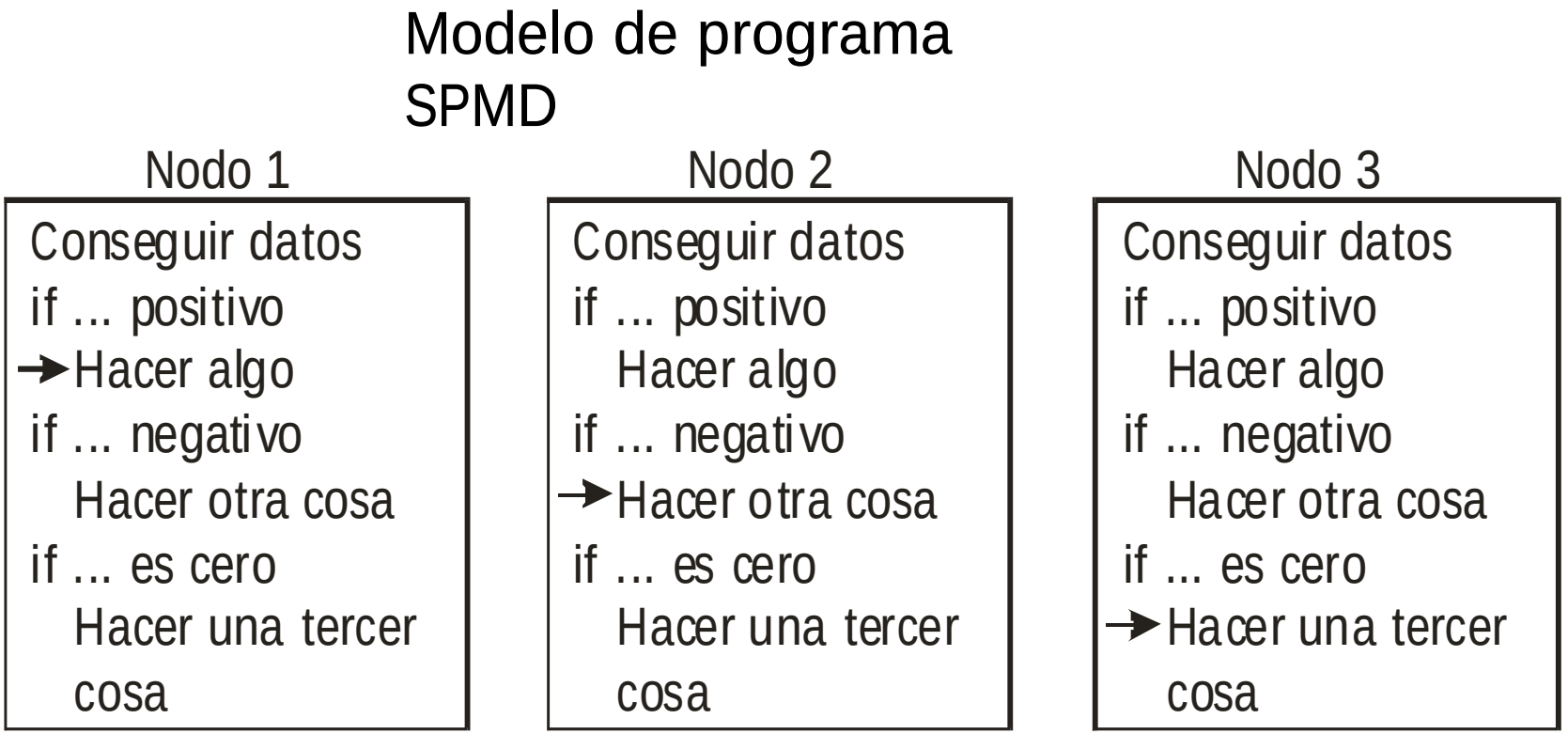
Modelo de programa SIMD

- Cargar Programa
- Broadcast de Instrucción



- Los Nodos reciben y ejecutan

Descomposición de Dominio

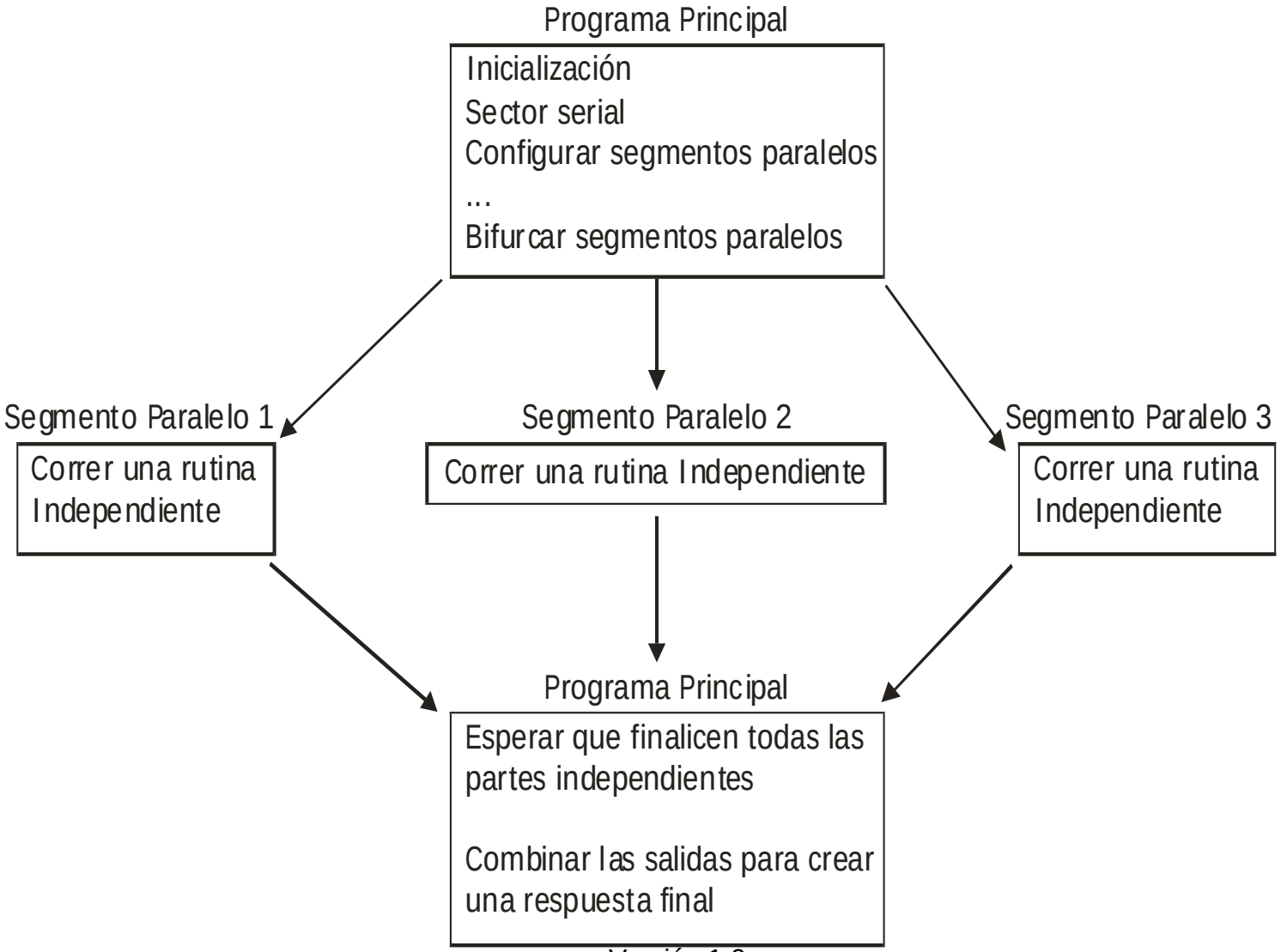


- Todos los nodos ejecutan el mismo programa, pero no las mismas instrucciones

Descomposición funcional

- Se concentra en el particionamiento de las operaciones del problema (control parallel).
  - Se trata de dividir el procesamiento en tareas disjuntas.
- Luego se examinan los datos que serán utilizados por las tareas definidas.
  - Si los datos son disjuntos, resulta un PARTICIONAMIENTO COMPLETO.
  - Si los datos NO son disjuntos, resulta un PARTICIONAMIENTO INCOMPLETO. Se requiere replicar los datos o comunicarlos entre los procesos asociados a las diferentes tareas.

Descomposición funcional



Modelos Híbridos

- Dos o más tipos de programación paralela dentro del mismo programa.
- Comúnmente utilizados en programas paralelos distribuidos en Internet (donde casi siempre existe la posibilidad de conseguir recursos *ociosos* adicionales).

Diseño y “paralelización”

- No siempre se dispone de recursos (en especial de tiempo) para diseñar una aplicación paralela – distribuida de acuerdo a los criterios y técnicas de programación especificadas.
- En múltiples ocasiones, se trata de obtener resultados de eficiencia computacional aceptable adaptando programas secuenciales a los modelos de programación paralela:

  “Paralelizar” una aplicación existente.
- Problemas

  Utilización de un código existente que no fue diseñado para ejecutar sobre múltiples recursos computacionales.

  Modelos enmarañados, “contaminación” del código heredado.

Paralelizando aplicaciones existentes

- Deben analizarse varios aspectos
  - ¿ Existe una partición funcional evidente ?
    - El código modular es el más fácil de paralelizar.
  - ¿ Existe forma de particionar los datos ?
    - Si existe, cual es la relación procesamiento/datos ?
  - ¿ Existen muchas variables globales ?
    - Cuidado !!! El manejo de recursos compartidos es un problema a resolver.
    - Considerar el uso de un servidor de variables globales

Modelos de comunicación entre procesos

- Cómo comunicar y/o sincronizar procesos paralelos.
- Modelos de comunicación:

  Modelo maestro-esclavo (master-slave).

  Modelo cliente-servidor.

  Modelo peer to peer.

Modelos de comunicación entre procesos

Master-slave

- Uno de los paradigmas de comunicación más sencillo
- Modelo en el cual se generan un conjunto de subproblemas y procesos que los resuelven
- Un proceso distinguido (*master*, maestro) y uno o varios procesos idénticos (*slaves*, esclavos).
- El proceso master controla a los procesos slave.
- Los procesos slave procesan.

Modelo master-slave

- El proceso master lanza a los esclavos y les envía datos.
- Luego de establecida la relación master/slave, la jerarquía impone la dirección del control del programa (del master sobre los slaves).
- La única comunicación desde los esclavos es para enviar los resultados de la tarea asignada.
- Habitualmente no hay dependencias fuertes entre las tareas realizadas por los esclavos (poca o nula comunicación entre esclavos).
- Modelos sincrónicos y asincrónicos.

Modelo master-slave

- Características:
  - El paradigma es sencillo, pero requiere programar el mecanismo de lanzado de tareas, la distribución de datos, el control del master sobre los slaves y la sincronización en caso de ser necesaria.
  - La selección de recursos es fundamental para la performance de las aplicaciones master/slave (balance de cargas).

Modelo cliente/servidor

- Modelo de comunicación que identifica dos clases de procesos. Procesos de una clase (los *clientes*) solicitan servicios a procesos de otra clase (los *servidores*), que atienden los pedidos.
- Puede ser utilizado para comunicar procesos que ejecutan en un único equipo, pero es una idea potencialmente más poderosa para comunicar procesos distribuidos en una red.
- El modelo C/S provee un mecanismo para comunicar aplicaciones distribuidas (que funcionen simultáneamente como clientes y servidores para diferentes servicios), convenientemente, de acuerdo a las características de la red.

MEDIDAS DE PERFORMANCE

MEDIDAS de PERFORMANCE

Objetivos :

- Estimación de desempeño de algoritmos paralelos.
- Comparación con algoritmos seriales.

Factores intuitivos para evaluar la performance:

- Tiempo de ejecución.
- Utilización de los recursos disponibles.

MEDIDAS de PERFORMANCE

Desempeño es un concepto complejo y polifacético.

Tiempo de ejecución: Medida tradicionalmente utilizada para evaluar la eficiencia computacional de un programa.

Almacenamiento de datos en dispositivos y transmisión de datos entre procesos influyen en el tiempo de ejecución de una aplicación paralela.

Utilización de recursos disponibles y capacidad de utilizar mayor poder de cómputo para resolver problemas más complejos o de mayor dimensión son las características más deseables para aplicaciones paralelas.

Tiempo total de ejecución como medida del desempeño:

- Simple de medir.
- Útil para evaluar el esfuerzo requerido al resolver un problema.

TIEMPO de EJECUCIÓN

Tiempo en estado ocioso es consecuencia del no determinismo, minimizarlo debe ser un objetivo del diseño.

Motivos: ausencia de recursos de cómputo disponibles o ausencia de datos sobre los cuales operar.

Soluciones: técnicas de balance de cargas para distribuir requerimientos de cómputo o rediseño del programa para distribuir datos adecuadamente.

MEJORA de PERFORMANCE

SPEEDUP

Es una medida de la mejora de rendimiento (performance) de una aplicación al aumentar la cantidad de procesadores (comparado con el rendimiento al utilizar un solo procesador).

SPEEDUP ABSOLUTO $S_N = T_0 / T_N$

Siendo :

- T_0 el tiempo del mejor algoritmo SERIAL (en cuanto al tiempo de ejecución, o sea el algoritmo más rápido) que resuelve el problema.
- T_N el tiempo del algoritmo paralelo ejecutado sobre N procesadores.

SPEED-UP

Siendo T_K el tiempo total de ejecución de una aplicación utilizando K procesadores.

Se define el **SPEEDUP ALGORÍTMICO** como

$$S_N = T_1 / T_N$$

T_1: tiempo *serial*, T_N: tiempo *paralelo*

El speedup algorítmico es el más frecuentemente utilizado en la práctica para evaluar la mejora de desempeño (en cuanto a tiempos de ejecución) de programas paralelos.

El speedup absoluto es difícil de calcular porque no es sencillo conocer el mejor algoritmo serial que resuelve un problema determinado.

SPEED-UP

Para la medición de los tiempos, debe considerarse la configuración de la máquina paralela utilizada.

Comparación justa :

- Para calcular speedup absoluto utilizar el mejor algoritmo serial disponible (o el mejor conocido, en caso de no existir una cota inferior para su complejidad).
- Análisis del hardware disponible.
- Coeficientes (“pesos”) asignados a los equipos en máquinas heterogéneas.
- Tomar en cuenta el asincronismo : Medidas estadísticas.
- Existen algoritmos diseñados para la comprobación (benchmarks).

SPEED-UP

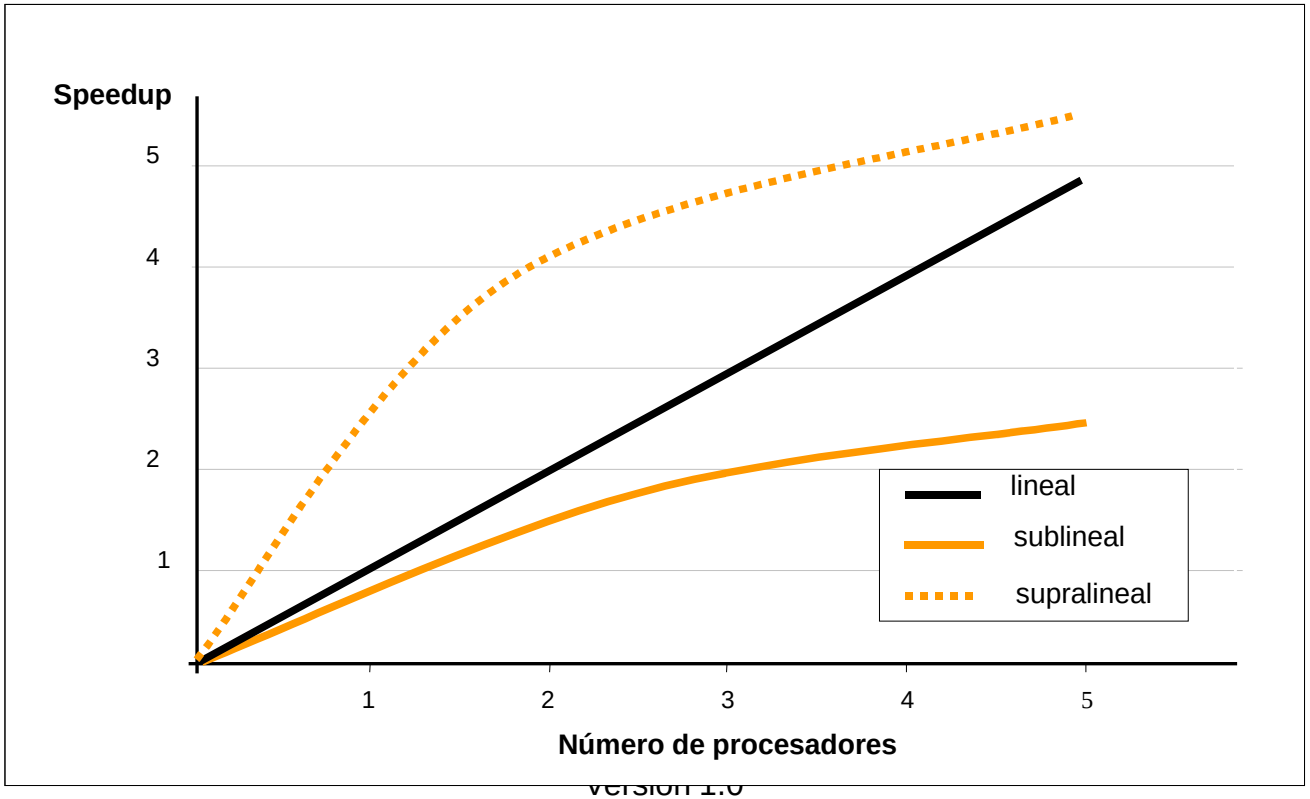
La situación ideal es lograr el speedup lineal.

Al utilizar *p* procesadores obtener una mejora de factor *p*.

La realidad indica que es habitual obtener speedup *sublineal*.

Utilizar *p* procesadores no garantiza una mejora de factor *p*.

En ciertos casos es posible obtener valores de speedup *superlineal*.



SPEED-UP

Motivos que impiden el crecimiento lineal del SPEED-UP

- Demoras introducidas por las comunicaciones.
- Overhead en intercambio de datos.
- Overhead en trabajo de sincronización.
- Existencia de tareas no paralelizables.
- Cuellos de botella en el acceso al hardware necesario.

Los factores mencionados incluso pueden producir que el uso de más procesadores sea contraproducente para la performance de la aplicación.

EFICIENCIA

La eficiencia se define mediante :

$$E_N = \frac{T_1}{N * T_N}$$

Es decir $E_N = S_N / N$.

Corresponde a un valor normalizado del SPEED-UP (entre 0 y 1), respecto a la cantidad de procesadores utilizados.

Valores de eficiencia cercanos a uno identificarán situaciones casi ideales de mejora de performance.

ESCALABILIDAD

Capacidad de agregar procesadores para obtener mejor rendimiento en la ejecución de aplicaciones paralelas.

Constituye una de las principales características deseables de los algoritmos paralelos y distribuidos.

MEDIDAS de PERFORMANCE
OTRO ENFOQUE

EVALUANDO la UTILIZACIÓN de RECURSOS DISPONIBLES

- UTILIZACIÓN

  Mide el porcentaje de tiempo que un procesador es utilizado durante la ejecución de una aplicación paralela.

  USO = tiempo ocupado / (tiempo ocioso + tiempo ocupado)

  Lo ideal es mantener valores equitativos de utilización entre todos los procesadores de una máquina paralela.

MEDIDAS de PERFORMANCE

LEY DE AMDAHL (1967):

"La parte serial de un programa determina una cota inferior para el tiempo de ejecución, aún cuando se utilicen al máximo técnicas de paralelismo."

MEDIDAS DE PERFORMANCE

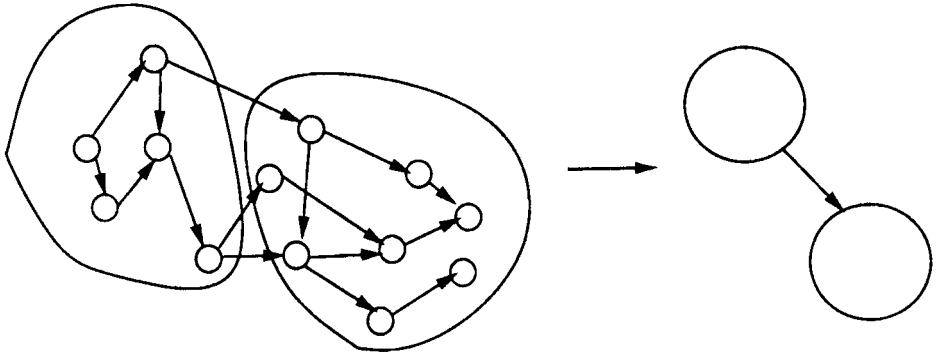
CONCLUSIÓN de la LEY de AMDAHL:

- La razón para utilizar un número mayor de procesadores debe ser resolver problemas más grandes o más complicados, y no para resolver más rápido un problema de tamaño fijo.

FACTORES QUE AFECTAN LA PERFORMANCE

GRANULARIDAD

Cantidad de trabajo
que realiza cada nodo



TAREA	GRANULARIDAD
operación sobre bits	super-fino (extremely fine grain)
una instrucción	grano fino (fine grain)
un proceso	grano grueso (large grain)

Aumentar la granularidad:

- Disminuye overhead de control y comunicaciones.
- Disminuye el grado de paralelismo.

MEDIDAS DE PERFORMANCE

El OBJETIVO del diseño de aplicaciones paralelas es lograr un compromiso entre:

- Overhead de sincronización y comunicación.
- Grado de paralelismo obtenido.

- Técnicas de SCHEDULING y de BALANCE de CARGAS son útiles para mejorar el desempeño de aplicaciones paralelas.

SCHEDULING y

BALANCE de CARGAS

SCHEDULING o MAPEO

Asignación de recursos a los múltiples procesos que ejecutarán en paralelo.

El mapeo determina dónde y cuándo se ejecutará una tarea.

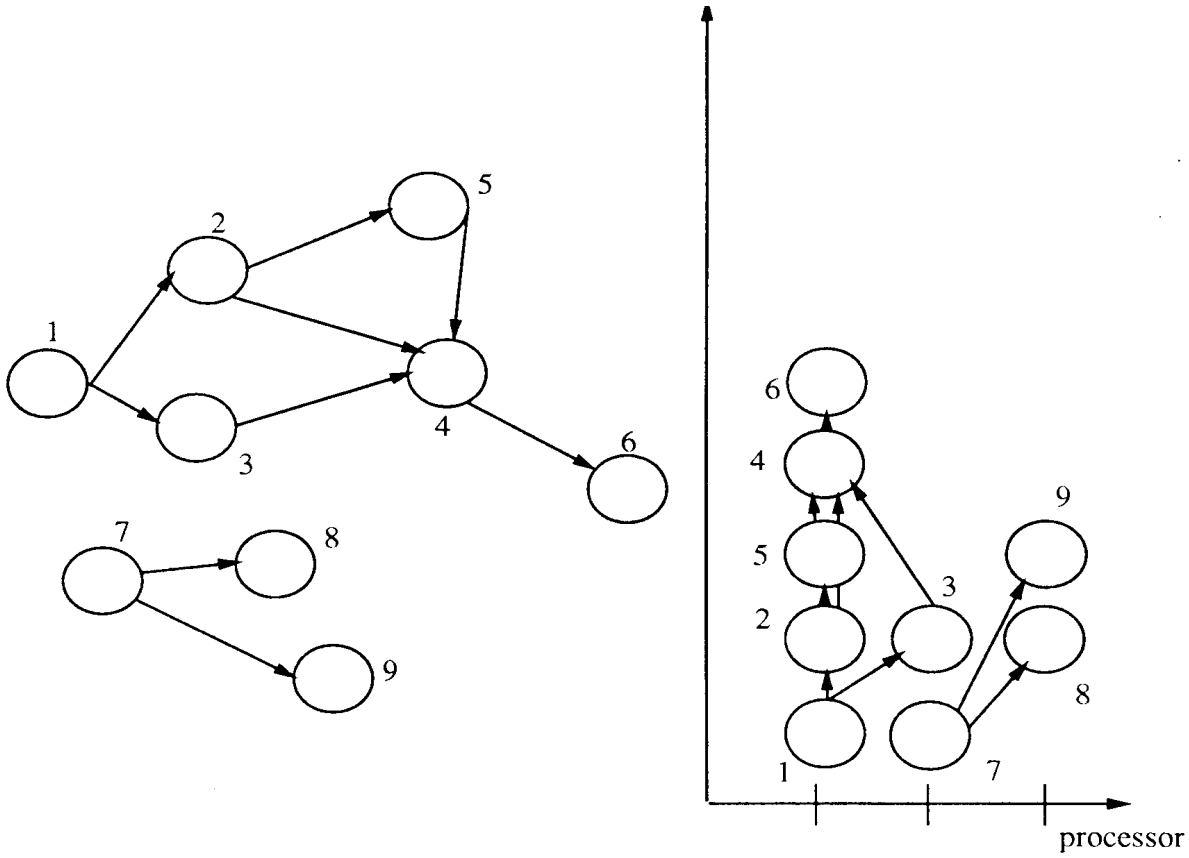
Las dependencias entre tareas definirán pautas para la asignación.

Usualmente se utilizan técnicas de IO para planificar la asignación de recursos de modo de optimizar un determinado criterio.

- Tiempo total de ejecución.
- Utilización de los recursos.
- Balance de cargas entre recursos.

SCHEDULING

Enlaza el algoritmo (grafo) con el hardware (procesador, tiempo)



SCHEDULING

Debe tomarse en cuenta la topología de la red conjuntamente con las dependencias de datos y las comunicaciones, de modo de no afectar los costos de comunicaciones.

- Aprovechar la “localidad” de datos.
- Reducir las comunicaciones.

SCHEDULING

Estrategias

- Procesos que pueden ejecutar concurrentemente se colocan en procesadores diferentes.
- Procesos que se comunican con alta frecuencia se colocan en el mismo procesador, o en procesadores “vecinos”

Existen mecanismos teóricos de asignación de recursos para los diferentes modelos de descomposición en tareas y arquitecturas paralelas estudiadas.

- Ejemplo : los grafos de algoritmos (árboles, anillos, mallas) sobre mallas 2D, hipercubos, etc.

BALANCE de CARGAS

Factor relevante sobre el desempeño de aplicaciones distribuidas ejecutando en una red.

El objetivo consiste en evitar que la performance global del sistema se degrade a causa de la demora en tareas individuales.

Muy importante en entornos no dedicados.

TÉCNICAS de BALANCE de CARGAS

- También conocidas como “técnicas de despacho”.
- Clasificación :
  - Técnicas estáticas (planificación).
  - Técnicas dinámicas (al momento del despacho).
  - Técnicas adaptativas.

- Principales criterios utilizados :
  - Mantener los procesadores ocupados la mayor parte del tiempo.
  - Minimizar las comunicaciones entre procesos.

TÉCNICAS de DESPACHO ESTÁTICAS

- Decisiones de despacho tempranas.
- Se utilizan técnicas de planificación.
- La asignación inicial se mantiene, independientemente de lo que suceda.
- Efectiva en ambientes de redes poco cargadas.
- Falla en ambientes compartidos de carga variable.

NO TIENEN EN CUENTA FLUCTUACIONES DE CARGA DE LA RED

TÉCNICAS de DESPACHO DINÁMICAS

- Involucran estrategias para determinar un procesador a una tarea *durante* la ejecución de la aplicación.
- Usuales en Modelo amo-esclavo.
- La asignación se realiza en el momento de creación de una nueva tarea.
- Usualmente consideran la situación en el instante del despacho exclusivamente.
- Efectiva en ambientes compartidos de carga variable.

TRATAN DE APROVECHAR FLUCTUACIONES DE CARGA DE LA RED

TÉCNICAS de DESPACHO ADAPTATIVAS

- Realizan el despacho de acuerdo al estado actual de la red.
- Pueden incorporar herramientas de predicción del futuro.
- Utilizan técnicas de migración de procesos.

APROVECHAN COMPLETAMENTE LAS FLUCTUACIONES DE CARGA DE LA RED

PARÁMETROS RELEVANTES en la DETERMINACIÓN de la CARGA

- Consumo de CPU

  Porcentaje de uso u operaciones/seg.
- Uso de disco

  Bloques transferidos del controlador al dispositivo.
- Tráfico de red

  Paquetes transmitidos y recibidos.

Herramientas

- Multi-Threading
- MPI
- HPF

Threads

- En un sistema tradicionales UNIX los procesos tienen un espacio de direccionamiento y un único thread (hilo) de control.
- Sobre fines de los ’80 surgen sistemas multiprocesadores debido a una mayor necesidad de procesamiento.
- Los diseñadores de los sistemas operativos invirtieron esfuerzos focalizándose en el desarrollo de sistemas escalables para los multiprocesadores.
- Surgen nuevos núcleos que permiten varios threads de ejecución y se proveen primitivas para permitir threads a nivel de proceso de aplicación.
  - Beneficios:
    - Mejor aprovechamiento de los recursos de procesador en ambientes multiprocesadores.
    - Rapidez en la comunicación entre los diferentes threads.
    - Dividir las aplicaciones en módulos funcionales.
    - Mejor tiempo de ejecución que procesos independientes.
  - Desventajas:
    - Programación más difícil.
    - Dificultad de encontrar los errores.

Threads

- Todos los threads en un proceso comparten el espacio de direccionamiento.
- No existe ningún tipo de protección entre los threads.
- Información privada de cada thread:
  - Contador de programa.
  - Pila (Stack).
  - Conjunto de registros.
  - Estado.
- Comparten:
  - Espacio de Direccionamiento de memoria.
  - Archivos abiertos.
  - Señales.
- Los threads se pueden proveer tanto a nivel de usuario como a nivel de núcleo del sistema operativo

Threads a nivel de Núcleo del sistema

- Son soportados directamente por el sistema operativo.
- El sistema operativo provee soporte para la creación, planificación y administración en el ambiente del núcleo.
- El sistema operativo reconoce a cada thread como una unidad a ser planificada.
- Beneficios:
  - El bloqueo de un thread de un proceso no bloquea a todos los demás threads del mismo proceso.
  - Mayor nivel de paralelismo en sistemas multiprocesador, ya que los threads de un proceso son planificados independientemente por el sistema operativo.
- Desventajas:
  - El cambio de contexto entre los diferentes threads es más costoso (en términos de ciclos de CPU utilizada) que en la implementación a nivel de usuario.
  - El sistema operativo debe mantener más estructuras en memoria.

Threads a nivel de Usuario

- Son implementados en una biblioteca a nivel de usuario.
- La biblioteca provee soporte para la creación, planificación y administración sin soporte de parte del sistema operativo.
- El sistema operativo desconoce la existencia de los threads.
- Beneficios:
  - Rápido cambio de contexto entre los diferentes threads. El núcleo del sistema no interviene.
  - La planificación entre los threads es independiente del tipo de planificación utilizado por el sistema operativo.
- Desventajas:
  - En un ambiente multiprocesador solo ejecuta un thread en cada momento aunque puedan existir recursos de procesador que estén ociosos.
  - Si el thread que está ejecutando se bloquea (p.ej: a través de un llamado a una rutina del sistema o un fallo de página), entonces bloquea a todos los demás threads.

MPI Message Passing Interface

MPI

- Biblioteca estándar para programación paralela bajo el paradigma de comunicación de procesos mediante pasaje de mensajes.
- Biblioteca, no lenguaje. Proporciona funciones.

- Creado por la industria, desarrolladores de software y aplicaciones y científicos (IBM, Intel, PVM, nCUBE).

Objetivo: desarrollar un estándar portable y eficiente, para programación paralela.

MPI

Características

- Plataforma objetivo: Memoria distribuida.
- Paralelismo explícito (definido y controlado en su totalidad por el programador).
- Único mecanismo de comunicación : MP.
- Modelo de programas : SPMD.
- Número de tareas fijado en tiempo pre-ejecución, no incluye primitiva spawn.
- 125 funciones (más que PVM).

Objetivos específicos

- Portabilidad

  Definir entorno de programación único.

- Eficiencia

  Sacar ventajas del hardware especializado.

- Funcionalidad

  Estructuras avanzadas, manejo de grupos, comunicación optimizada.

¿ Qué incluye el estándar ?

- Comunicaciones Punto a Punto.
- Operaciones Colectivas.
- Agrupamiento de procesos.
- Contextos de comunicación.
- Topologías de procesos.
- Soporte para Fortran y C.
- Manejo del entorno de programación.
- Interfaz personalizada.

¿ Qué NO incluye el estándar ?

- Comunicaciones de memoria compartida.
- Soporte de SO para recepciones por interrupción
- Ejecución remota de procesos.
- Facilidades de debug.
- Soporte para threads.
- Soporte para control de tareas.
- Pobre manejo de I/O.

¿ Qué permite el estándar MPI ?

- Disponer de un amplio conjunto de rutinas de comunicación punto a punto.
- Disponer de un amplio conjunto de rutinas de comunicación entre grupos de procesos.
- Definir contextos de comunicación entre grupos de procesos.
- Especificar diferentes topologías de comunicación.
- Crear tipos de datos derivados para enviar mensajes que contengan datos no contiguos en memoria.

¿ Qué permite el estándar MPI ?

- Hacer uso de comunicación asincrónica.
- Administrar eficientemente el pasaje de mensajes.
- Desarrollar aplicaciones más eficientes en MPP y clusters.
- Portabilidad total.
- Existe una especificación formal de MPI.
- Existen varias implementaciones de calidad de MPI disponibles (LAM, MPICH, CHIMP).

Las seis rutinas básicas MPI

Inicializar

ierr = MPI_Init(&argc,&argv);

Número de Procesos

ierr=MPI_Comm_size(MPI_COMM_WORLD,&npes);

Identificador del Proceso

ierr=MPI_Comm_rank(MPI_COMM_WORLD,&iam);

Envio de mensajes

ierr=MPI_Send(buffer,count,datatype,destino,tag,comm)

Recibir Mensajes

ierr=MPI_Recv(buffer,count,datatype,source,tag,comm,estado)

Finalizar

ierr = MPI_Finalize()

Primitivas

Pasaje de mensajes estándar .

Send (address, length, destino, tag)

address = dirección de memoria inicial del buffer que contiene datos.

lenght = largo en byte del mensaje.

destino = identificación del proceso destino.

tag = tipo de mensaje, permite discriminar recepción.

Primitivas

MPI_Recv(buffer, count, datatype, source, tag, comm, status)

buffer: Dirección de inicio del buffer de envío o de recepción.
count: Número de elementos a enviar o a recibir.
datatype: Tipo de dato de cada elemento.
source: Identificador del remitente .
tag: Etiqueta del mensaje.
comm: Representa el dominio de comunicación.
status: Permite obtener fuente, tag y contador del msg recibido.

Funciones generales

MPI_Init

Inicio de un proceso que utilizará funciones MPI.

int MPI_Init (int *argc, char **argv)

MPI_Finalize

Fin de un proceso que utilizó funciones MPI.

int MPI_Finalize ()

MPI_Get_processor_name

Retorna el nombre del procesador (o equipo) donde ejecuta el proceso.

int MPI_Get_processor_name(char *name, int *resultlen)

Comunicaciones Colectivas

- Las comunicaciones colectivas permiten la transferencia de datos entre todos los procesos que pertenecen a un grupo específico.

- No se usan etiquetas para los mensajes, estas se sustituyen por identificadores de los grupos (*comunicadores*).

  Las operaciones colectivas comprenderán a todos los procesos en el alcance del comunicador.

- Por defecto, todos los procesos se incluyen en el comunicador genérico MPI_COMM_WORLD.

Operaciones colectivas de MPI

Se clasifican en tres clases:

- Sincronización. (Operaciones de barrera).

  Procesos que esperan a que otros miembros del grupo alcancen el punto de sincronización.

- Movimiento (transferencia) de datos.

  Operaciones para difundir, recolectar y esparcir datos entre procesos.

- Cálculos colectivos.

  Operaciones para reducción global, tales como suma, máximo, mínimo o cualquier función definida por el usuario.

Operaciones colectivas de MPI

- Las operaciones colectivas son bloqueantes.
- Las operaciones colectivas que involucran un subconjunto de procesos deben precederse de un particionamiento de los subconjuntos y relacionar los nuevos grupos con nuevos comunicadores.

Operaciones colectivas de MPI

- **MPI_Barrier (comm)**

  Crea una *barrera* de sincronización en un grupo.

  Al llegar a la invocación a MPI_Barrier, cada tarea se bloquea hasta que todas las tareas del grupo alcancen la misma invocación de MPI_Barrier.

- **MPI_Bcast (buffer, count, datatype, root, comm)**

  Envía un msg desde el proceso con rango root a todos los demás procesos del grupo.

- **MPI_Scatter (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)**

  Distribuye distintos mensajes desde una tarea a cada tarea en el grupo.

Operaciones colectivas de MPI

- **MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)**

  Reúne distintos mensajes desde cada tarea del grupo a un destino único

- **MPI_Allgather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)**

  Concatena los datos a todas las tareas de un grupo. Cada tarea realiza un broadcast uno a uno con las restantes del grupo.

- **MPI_Reduce (sendbuf, recvbuf, count, datatype, op, root, comm)**

  Aplica una operación de reducción a todas las tareas en el grupo y almacena el resultado en una tarea.

Sincronización

Modelo de “barrera”.

- Todos los procesos invocan a la rutina.
- La ejecución es bloqueada hasta que todos los procesos ejecuten la rutina.

  - ***En C ierr = MPI_Barrier(communicator);***

Difusión

- Un proceso envía un mensaje (***root***).
- Todos los otros reciben el mensaje.
- La ejecución se bloquea hasta que todos los procesos ejecuten la rutina.
- Automáticamente actúa como punto de sincronización.
  - **En C**

```
ierr = MPI_Bcast(&buffer,count,datatype,
                 root,communicator);
```

Recolección

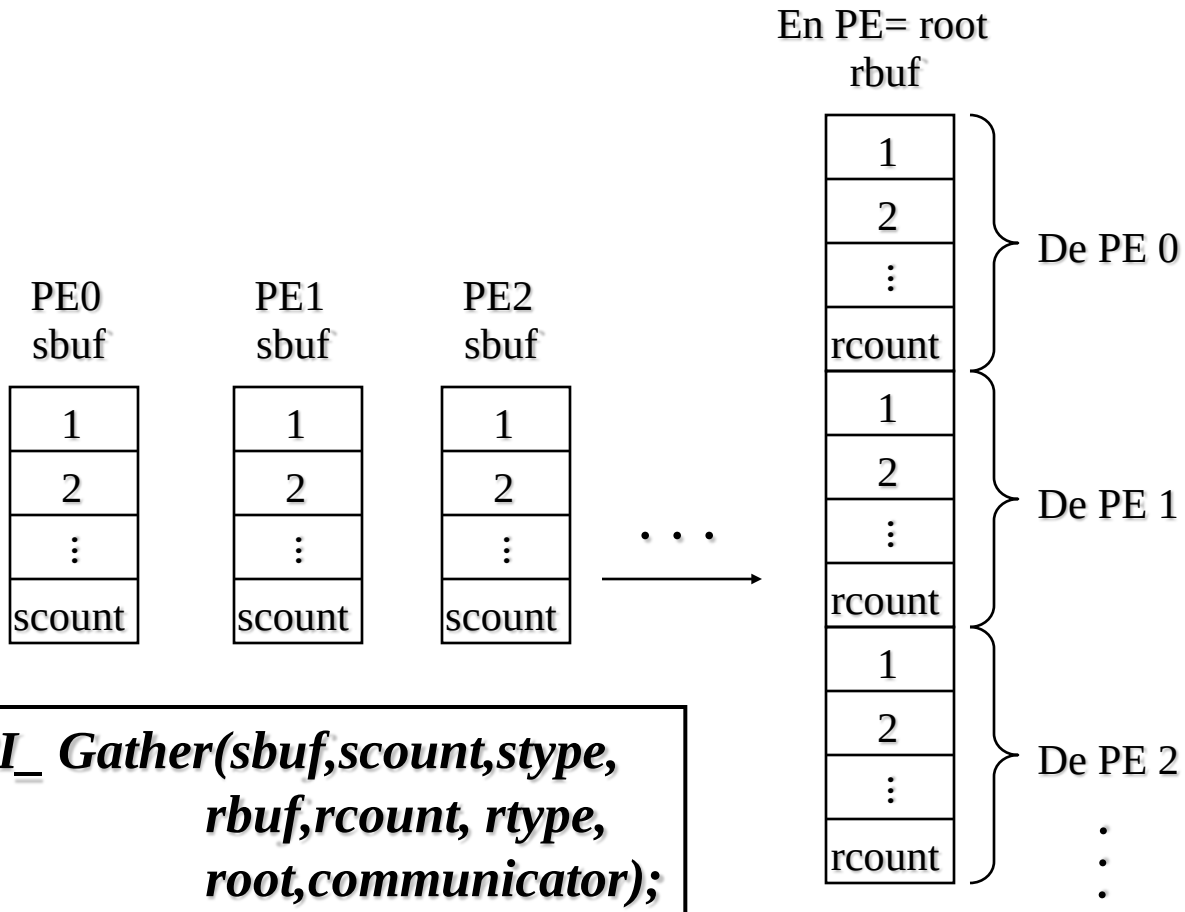
Intercambio global de datos.

- Cada proceso envía diferentes datos al proceso principal (***root***) y este los agrupa en un arreglo.

- **En C**

  ierr = MPI_ Gather(sbuf,scount,stype,rbuf,rcount,
  rtype,root,communicator);

Recolectar datos



MPI_ Gather(sbuf,scount,stype, rbuf,rcount, rtype, root,communicator);

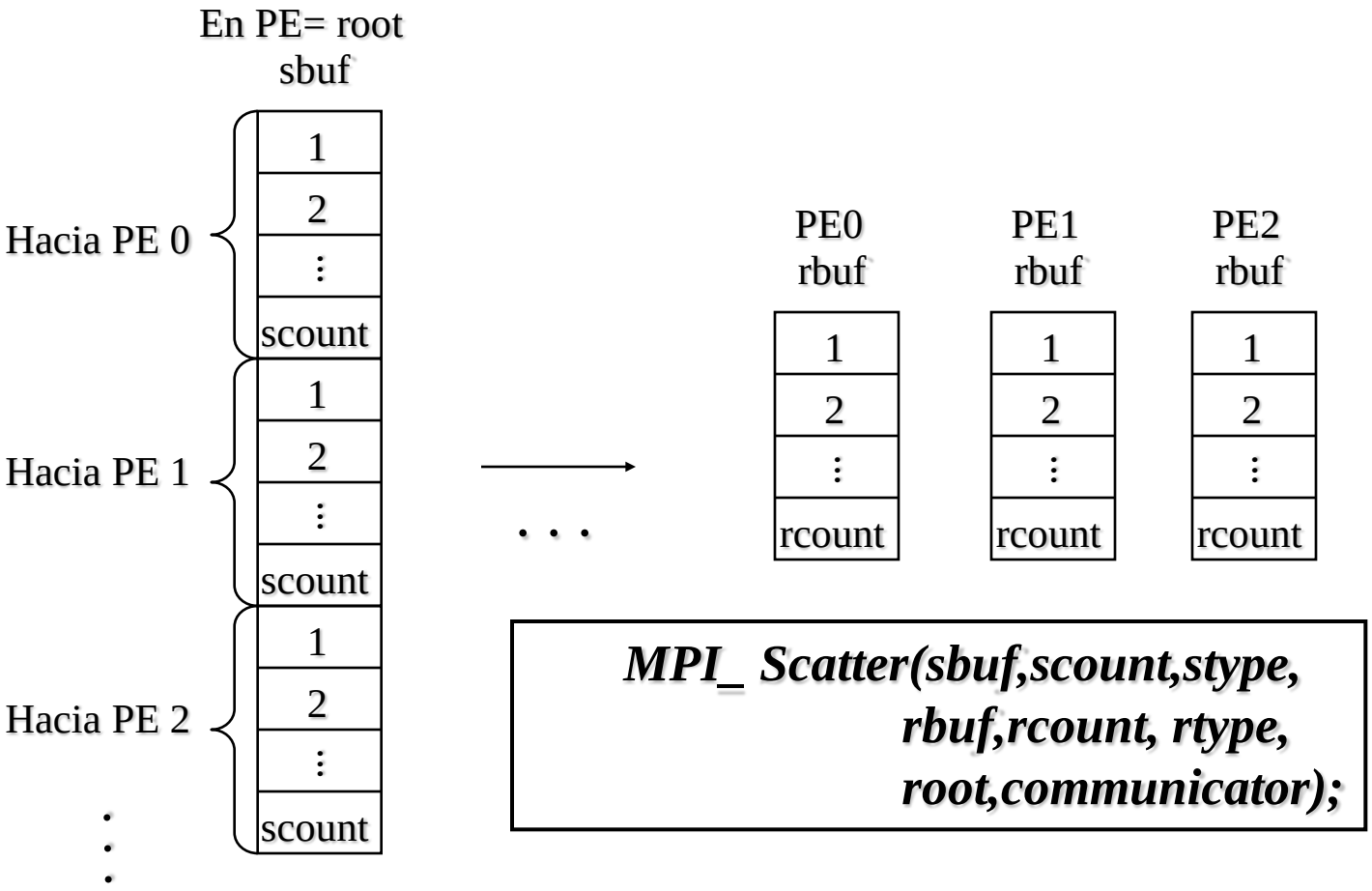
Esparcir

- Intercambio global de datos.
  - El proceso principal (***root***) rompe un arreglo de datos y envíe las partes a cada procesador.

  - **En C**

    ierr = MPI_ Scatter(sbuf,scount,stype,rbuf,rcount,
    rtype,root,communicator);

Esparcir



MPI_ Scatter(sbuf,scount,stype, rbuf,rcount, rtype, root,communicator);

Cálculos Colectivos

Conceptos

- El usuario puede combinar cálculos parciales de todos los procesos.
- Los resultados están disponibles en un proceso particular o en todos los procesos.
- Ejecución sincronizada.

Cálculos Colectivos

- Combinación de resultados parciales.
  - El proceso principal (***root***) recibe los cálculos parciales y los combina usando la operación indicada.
  - ***En C***

    ierr = MPI_ Reduce(sbuf,rbuf,count,datatype,
    operation, root, communicator);

- Todos almacenan el resultado con : ***MPI_ Allreduce***
  - La rutina no requiere el parámetro ***root***

MPI-2

Nuevas funcionalidades:

- Creación dinámica de procesos.
- Mejora de operaciones de Entrada/salida.
- Primitivas para comunicación utilizando memoria compartida.
- Nuevas operaciones de comunicación colectiva.
- Extensión a C++ y Fortran90.
- Es necesario un proceso demonio por usuario.

HPF High Performance Fortran

Fortran - Historia

- FORmula TRANslation, creado por IBM en los años 1950.
- Popular en los años 1960.
- En 1972 se estandariza el Fortran 66.
- Entre 1977 y 1980 se define Fortran 77.
- Fortran 90.
- Fortran 95.
- HPF.

Fortran 77

- Sintaxis difícil.
- No utiliza notación vectorial.
- No dispone de allocamiento dinámico.
- No se puede definir tipos de datos estructurados.
- No soporta recursión.
- Utilización de bloques de memoria COMMON.

Fortran 90

- Sintaxis libre.
- Operaciones vectoriales.
- Allocamiento dinámico.
- Permite definir tipos de datos estructurados.
- Recursión.
- Módulos.
- Conceptos de programación orientada objetos:
  - Abstracción de datos
  - Ocultamiento de datos
  - Encapsulamiento
  - Herencia
  - Polimorfismo

Fortran 90

- Funciones vectoriales
  - SUM(x)
  - SUM(MX,DIM=1), SUM(MX,DIM=2)
  - SUM(X,MASK=W>4)
  - ALL(SOURCE)
  - ANY
  - COUNT
  - MAXVAL, MINVAL

Fortran 90

- Funciones de vectores
- MERGE (v1,v2,MASK)
- SPREAD(v1,DIM,NCOPIES)
- PACK(v1,MASK)
- UNPACK(v1,MASK,FIELD)

HPF

- Promovido por el High Performance Fortran Forum (HPFF)
  - (Cray, DEC, IBM, INTEL, otros)
  - http://www.hpfpc.org/index-E.html
- 1993 v1.0
- 1997 v2.0
- 1999 HPF/JA 1.0

HPF

- Objetivos del HPF:
  - Soportar paralelismo a nivel de datos.
  - Obtener buenos desempeños en arquitecturas MIMD y SIMD.
  - Soportar optimizar los códigos en diferentes hardware.

HPF - Conceptos

- Cada procesador ejecuta el mismo programa (SPMD).
- Cada procesador opera sobre un sub-conjunto de los datos.
- HPF permite especificar que datos procesa cada procesador.

HPF - Directivas

- !HPF$ <directiva>
- Si se trabaja con un compilador HPF se interpretan las directivas.
- Sino, p. ej. compilador F90, se toman como comentario.

HPF – Processor

Establece una grilla conceptual de procesadores

- !HPF$ PROCESSOR, DIMENSION(4) :: P1
- !HPF$ PROCESSOR, DIMENSION(2,2) :: P2
- !HPF$ PROCESSOR, DIMENSION(2,1,2) :: P3

HPF – Distribute

Especifica como distribuir los datos entre las unidades de procesamiento

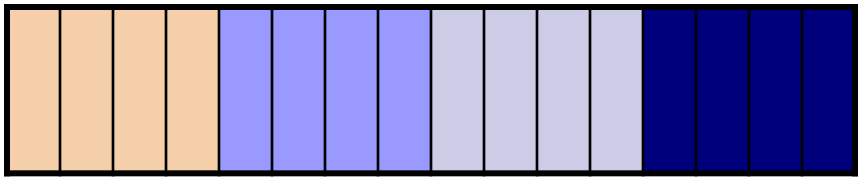
- !HPF$ DISTRIBUTE (BLOCK) ONTO P1 ::A
- !HPF$ DISTRIBUTE (CYCLIC) ONTO P1 ::B
- !HPF$ DISTRIBUTE (CYCLIC,BLOCK) ONTO P2 ::B

Variante BLOCK(k) y CYCLIC(k)

HPF – Distribute

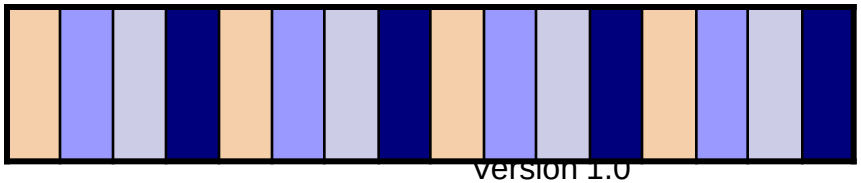
BLOCK

```
REAL, DIMENSION(16) :: A
!HPF$ PROCESSOR, DIMENSION(4) :: P1
!HPF$ DISTRIBUTE (BLOCK) ONTO P1 ::A
```



CYCLYC

```
REAL, DIMENSION(16) :: A
!HPF$ PROCESSOR, DIMENSION(4) :: P1
!HPF$ DISTRIBUTE (CYCLYC) ONTO P1 ::A
```



HPF – Quien calcula ?

El dueño del lado izquierdo de una asignación.

```
DO i=1,n
  A(i) = B(i) – C(i)
END DO
```

HPF – Variables escalares

Generalmente las variables escalares se replican a todos los procesadores.

HPF – ALING

Distribuir un vector o matriz y alinear los demás.

- !HPF$ ALIGN A(:) WITH B(:)
- !HPF$ ALIGN A(i,:) WITH B(:,i)
- !HPF$ ALIGN A(i,:) WITH B(i)

HPF – Forall

Conceptualmente ejecuta cada entrada en paralelo.

```
FORALL (i=2:n)
  A(i) = A(i-1)
END FORALL
```

HPF – Intrínsecas

Funciones intrínsecas, dan el valor real (hardware) no lo especificado por la instrucción PROCESSOR

- NUMBER_OF_PROCESSORS()
- PROCESSORS_SHAPE()

HPF – Reglas

- Más procesos, más comunicaciones
- Buscar buen balance
- Preservar la localidad de datos
- Usar sintaxis vectorial

HPF – “Compiladores”

- ADAPTOR
- PGHPF
- Southampton Translation
- Japoneses, Otros

HPF – ADAPTOR

- SPMD Fortran.
- Soporta hpf 2.0.
- 2 componentes básicas:
  - Fadapt: traduce de HPF a fortran90.
  - Dalib: se encarga del paralelismo.

HPF – ADAPTOR

- Traduce a pasaje de mensajes para memoria distribuida.
- Multithread para memoria compartida.
- Combinado.

HPF – ADAPTOR mem. distribuida

- Se dividen los datos según la especificación.
- Se computan utilizando la regla del dueño.

HPF – ADAPTOR multithreading

- Conjunto de threads.
- Se alloca en una porción de la memoria compartida.
- La información de la distribución de datos se utiliza para distribuir la computación por los distintos threads.
- Hay un thread “principal” que es el dueño de la información no distribuida y el que define el flujo de trabajo.
- Invoca a las rutinas de OpenMP.