Universidad Católica San Pablo

Parallel Computing

Introduction

mleguen@ucsp.edu.pe

No existe ninguna computadora que no sea capaz de paralelismo

Evolucion de las computadoras

- Desde 1986 hasta 2002 el rendimiento de los procesadores incrementa del 50% cada año
- En 2011 solamente incrementa del 20%

- Solución ?
  - **Múltiples núcleos**
- **Problema :** Agregar más procesadores no mejora mágicamente los programas secuenciales

Porque necesitamos sistemas paralelos ?

- 20 % por año no es suficiente ?
- Los últimos años han conocidos avances muy importantes en ciencias
  - Procesamiento de imágenes/mallas
  - Análisis de datos (Big data)
  - Inteligencia Artificial
  - Simulaciones (previsiones climáticas)

Porque no podemos mejorar estos 20% ?

- Se incrementa la cantidad de transistores
  - Más pequeños
  - Más veloces
  - Mas energia -> mas calientes
- Entre más calientes menos fiables (problema de fiabilidad en supercomputadores)
- Lo que podemos incrementar (por el momento), es la cantidad de procesadores
- Procesadores multi-núcleos

Los programas no son directamente paralelizables ?

- Los programas han sido pensado de forma secuencial
- Hubo intentos para paralelizar automáticamente programas sin mucho éxito
- Pasar un algoritmo de forma secuencial -> paralelo puede resultar ineficientes
    - **Entonces tenemos que pensarlo un poco más**

Ejemplo de la suma global

Codigo serial :

```
sum = 0;

for (i = 0; i < n; i++) {
        x = Compute next value(. . .);
        sum += x;
 }
```

```
my sum = 0;
my first i = . . . ;
my last i = . . . ;
for (my i = my first i; my i < my last i; my i++) {
        my x = Compute next value(. . .);
        my sum += my x;
 }
```

1,4,3, 9,2,8, 5,1,1, 6,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9,

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Ejemplo de la suma global

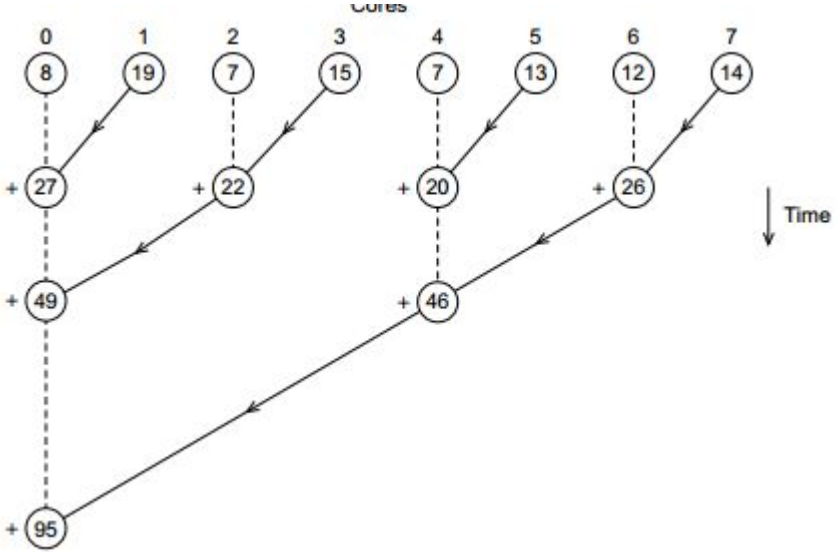
```
if (I'm the master core) {
        sum = my x;
        for each core other than myself {
                receive value from core;
                sum += value;
         }
} else {
        send my x to the master;
}
```

1,4,3, 9,2,8, 5,1,1, 6,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9,

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Ejemplo de la suma global

- Si la cantidad de núcleos es muy grande
  - el núcleo principal hará mucho trabajo

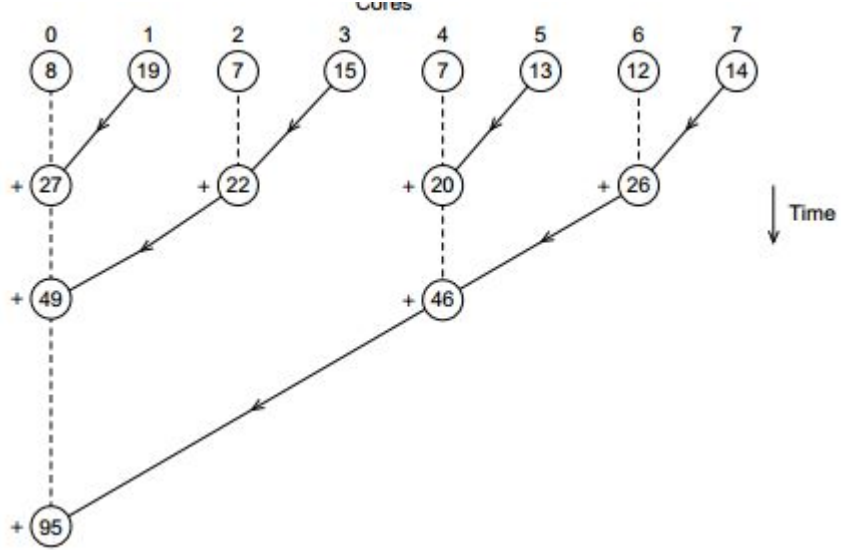


Cómo escribir programas paralelos ?

- Task parallelism : distribuir diferentes tareas a varios núcleos
- Data parallelism : distribuir los datos a varios núcleos
- Ej.: Prof principal tiene 100 alumnos, 5 profesores asistentes, un examen a 5 preguntas
  - Cada profesor asistente corrige 20 alumnos (data parallelism)
    - Cada núcleo hace los mismo
  - Cada profesor asistente corrige 1 pregunta para los 100 alumnos (task paraellism)
    - Cada uno de los núcleos hace algo distinto

Cómo escribir programas paralelos ?

- En el ejemplo de la suma la primera parte puede ser considerada como
  - data parallelism
- La segunda parte sería
  - Task parallelism
  - El núcleo principal recibe y agrega las sumas
  - Los otros núcleos mandan su suma parcial
- Cuando los núcleos pueden hacer un trabajo independiente entonces no hay dificultad
- Mientras que cuando necesitan coordinación las cosas se pueden poner más complejas



Cómo escribir programas paralelos ?

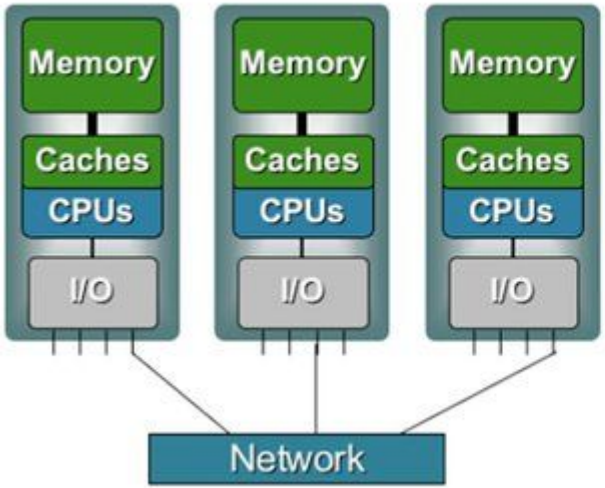
- Comunicación entre núcleos
- Load balancing
  - Queremos aprovechar al máximo cada núcleo
  - sobrecargar un núcleo significa malgastar los demás
- Sincronización
  - Generalmente los programas más eficientes utilizan una sincronización explícita
    - Ej. core 1 task 1 + core 2 task 2 -> synchronize ....
    - Esto puede generar programas complejos
    - Existen métodos de más alto nivel donde se sacrifica rendimiento contra simplicidad (OpenMP)

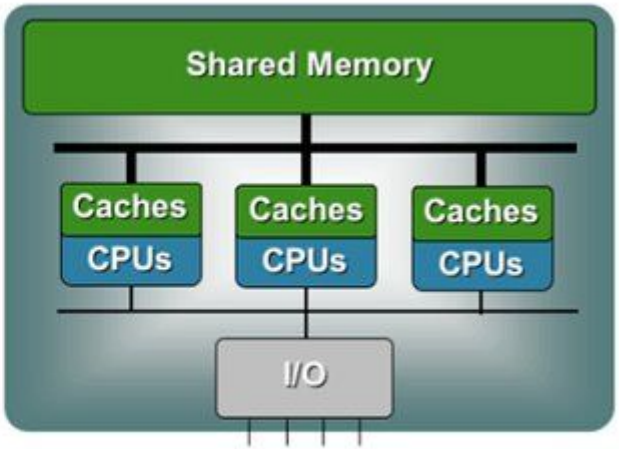
Herramientas

- OpenMP
- Thread C++ 11 o Pthread
- MPI Message Passing Interface

- Porque 3 extensiones del C++:
- Existen 2 tipos de sistemas paralelos
  - **Shared memory**
  - **Distributed memory**

Sistemas paralelos





Herramientas

- Shared memory
  - Núcleos pueden compartir el acceso la memoria principal de la computadora
  - Podemos coordinar los núcleos para observar y modificar la memoria
- Distributed memory
  - Cada núcleo tiene su propia memoria
  - Los núcleos deben de comunicar de forma explícita a través de red mandando “mensajes”
- Threads y OpenMP son diseñado para sistemas de shared-memory.
  - OpenMP hi-level, accesible
  - Threads coordinación entre núcleos
- MPI está diseñado para trabajar con sistemas distribuidos
  - Mandar mensajes

Parallel software
Parallel hardware

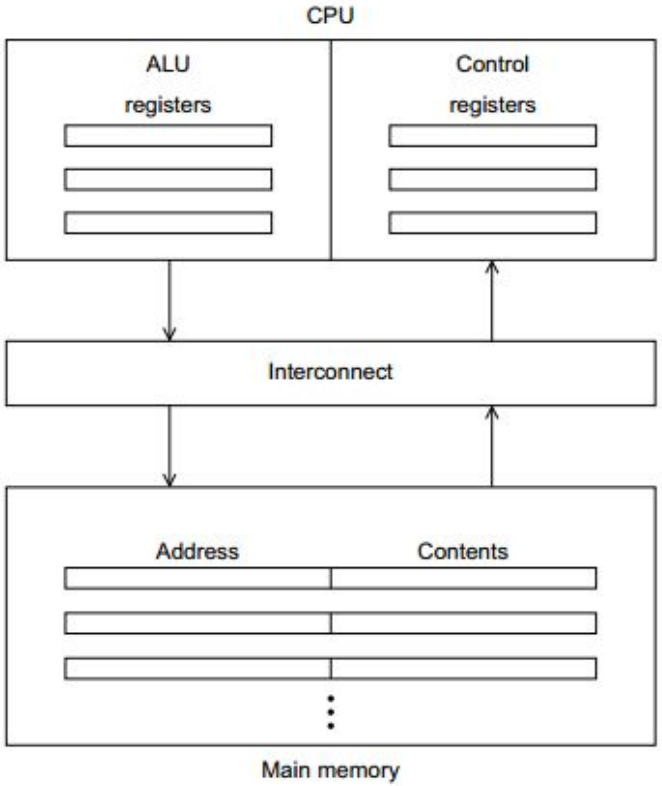
Software y Hardware paralelo - Objetivos

- Diferentes tipos de software
- Sistemas paralelos
- Evaluación de programas paralelos
- Método para el desarrollo de software paralelo

Sistema Serial - Arquitectura Von Neumann

- Main Memory
  - Localizaciones(datos, instrucciones)
  - Direcciones
- CPU
  - Control Unit (Que instrucción debe ejecutarse)
  - Arithmetic and Logic Unit (ALU) (Ejecutar la instrucción)
  - Registers (datos almacenados en el CPU)
    - Program counter (CU): almacena la dirección de la siguiente instrucción
    - Instrucción y datos
- Interconnection
  - Bus (parallel wires)

Sistema Serial - Arquitectura Von Neumann



Sistema Serial - Arquitectura Von Neumann

- Memoria transferida Memoria - CPU
  - Read/Fetch
- Memoria transferida CPU - Memoria
  - Write/stored
- Bus -> Von Neumann bottleneck
  - 2010 CPU puede ejecutar una instrucción más de 100 veces más rápido que leer un dato de la memoria
  - analogía a una fábrica que produce rápidamente y se encuentra esperando para materia prima
- **Veremos modificaciones hechas al modelo de Von Neumann para mejorar este problema**

Threads - multitasking

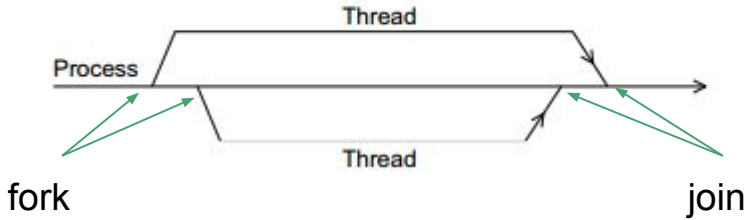
- Operating system
  - Software que maneja el hardware y recursos del software
- Process (Programa manejado por el OS)
  - Ejecutable (lenguaje máquina)
  - Bloque de memoria que contendrá
    - Código ejecutable
    - Call Stack (Variables temporales funciones)
    - Heap (asignación dinámica)
  - Información sobre el proceso (listo, esperando recursos etc.)

Threads - multitasking

- La mayoría de los sistemas son multitasking
  - Sistema maneja la ejecución de varios programas a la vez
  - Posible con un solo núcleo
    - **Time slice** : proceso corre sobre una pequeña unidad de tiempo
    - El OS puede cambiar muchas veces de proceso en un solo minuto
  - Un sistema operativo puede **bloquear** un proceso cuando le falta un recurso
    - Deja de ejecutarse
    - Hay una forma de seguir ejecutando un programa que espera un recurso
      - **Threading**

Threads - multitasking

- Threading
  - Divide un programa en múltiples tareas independientes
  - Un thread bloqueado mientras otro puede correr
  - **Es más rápido cambiar de thread** (lighter weight) que de proceso
    - Pertenecen al mismo proceso
    - Comparten la mayoría de los recursos del proceso
    - Call stack / Program counter independiente



Arquitectura Von Neumann - Modificaciones

- **Caching**
- **Virtual Memory**
- **Low level parallelism**

Caching

- Ej. Fábrica que produce rapido:
  - mejoramos el acceso a la fábrica
  - movemos el almacenamiento junto a la fábrica
- Mejor interconexión que permite transportar más datos en un solo acceso
- Dejamos de utilizar solo la main memory sino bloques de memoria cerca los registros del procesador
- Cache
  - Colección de localización de memoria que pueden ser accedidos rápidamente

Caching

- **Que datos poner en caché ?**
  - Datos e instrucciones físicamente cerca
  - Ej.: Después de ejecutar una instrucción generalmente ejecutamos la instrucción siguiente
  - Ej.: Después de acceder a una localización en memoria accedimos a la localización siguiente (v.at(i)<v.at(i+1))
- **Los arrays son asignados como bloques consecutivos en memoria**
  - **Locality (spatial/temporal locality)**
  - **Cache block 8x o 16x el tamaño de una locación en memoria**
  - **v[0] -> v[15]**

Caching

- Arquitectura
  - L1 - pequeño pero rápido
  - L2/L3 - grande y más lentos
- Cache hit
  - Cuando se encontró una variable en el caché
- Cache miss
  - Primero L1, después L2 etc. hasta la main memory (jerarquía)
- Inconsistency
  - CPU escribe en el cache -> cache != main memory
  - **write through** caches (escribimos en el caché + main memory)
  - **write back** caches (localización en el cache es marcada “dirty” así que antes de reemplazarla se escribirá en la main memory)

Cache mapping

- Fully associative
- Direct mapped
- n-way set associative

- Como saber cual reemplazar (evicted)
  - Se escoge la localización con el último acceso más antiguo

Memory Index	Cache Location		
	Fully Assoc	*Direct Mapped*	*2-way*
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Cache - Consecuencias en programación

```
double A[MAX][MAX], x[MAX], y[MAX];
/* 1*/
for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
        y[i] += A[i][j]*x[j];

 /* 2*/
for (j = 0; j < MAX; j++)
      for (i = 0; i < MAX; i++)
          y[i] += A[i][j]*x[j];
```

Cache - Consecuencias en programación

```
double A[MAX][MAX], x[MAX], y[MAX];
/* 1*/
for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
        y[i] += A[i][j]*x[j];

 /* 2*/
for (j = 0; j < MAX; j++)
     for (i = 0; i < MAX; i++)
        y[i] += A[i][j]*x[j];
```

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

Ej. MAX = 1000 (1) es 3 veces más rápido

Memoria Virtual

- En sistema multi-tasking la main memory no podrá almacenar todos los datos e instrucciones de todos los programas
- Memoria Virtual
  - La main memory es como un caché para la memoria secundaria
  - Se guarda en la main memory solamente las partes activas de los programas en ejecución
  - Lo demás está almacenado en el **swap space** de la memoria secundaria
  - Trabaja sobre bloques en memoria llama **pages (tamaño fijo 8-16kb)**

Instruction level parallelism

- **Functional units(o execution units)** en el CPU
  - Ejecutar varias instrucciones simultáneamente
- Pipelining
  - functional units organizados en etapas
  - ej. $9.87 \times 10^4 + 6.54 \times 10^3$

Time	Operation	Operand 1	Operand 2	Result
0	Fetch operands	9.87×10^4	6.54×10^3	
1	Compare exponents	9.87×10^4	6.54×10^3	
2	Shift one operand	9.87×10^4	0.654×10^4	
3	Add	9.87×10^4	0.654×10^4	10.524×10^4
4	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
5	Round result	9.87×10^4	0.654×10^4	1.05×10^5
6	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Instruction level parallelism

- Pipelining
  - functional units organizados en etapas
  - Cada etapa demora 1 x 10^{-9} seconds

```
float x[1000], y[1000], z[1000];
. . .
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

- Demoraria 7000 nanoseconds

Time	Operation	Operand 1	Operand 2	Result
0	Fetch operands	9.87×10^4	6.54×10^3	
1	Compare exponents	9.87×10^4	6.54×10^3	
2	Shift one operand	9.87×10^4	0.654×10^4	
3	Add	9.87×10^4	0.654×10^4	10.524×10^4
4	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
5	Round result	9.87×10^4	0.654×10^4	1.05×10^5
6	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Instruction level parallelism

- Pipelining
  - Si cada etapa representa un componente
  - La salida de cada componente es la entrada del siguiente (pipeline)
  - x[1] y y[1] puede ser leído mientras se compara los exponentes de x[0] y y[0]
  - Entonces podemos hacer 7 operaciones simulataneas
  - De 7000 pasamos a 1006 nanoseconds

Time	Operation	Operand 1	Operand 2	Result
0	Fetch operands	9.87×10^4	6.54×10^3	
1	Compare exponents	9.87×10^4	6.54×10^3	
2	Shift one operand	9.87×10^4	0.654×10^4	
3	Add	9.87×10^4	0.654×10^4	10.524×10^4
4	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
5	Round result	9.87×10^4	0.654×10^4	1.05×10^5
6	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Instruction level parallelism

- **Functional units(o execution units)** en el CPU
- Multiple Issue
  - Réplica funcional unit
  - **Static (compilation)**
  - **Dynamic (Super scalar) runtime**
- Ejemplo si podría replicar un adder en 2 adders podríamos reducir por dos este bucle

```
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

Instruction level parallelism

- **ILP es ejecutado en casos donde el sistema puede predecir las posibilidades de paralelismo**
  - **Casos no son evidentes (números de Fibonacci)**

Hardware multithreading

- **El sistema debe ser capaz de cambiar de threads muchommas rápido que de proceso**

- **Thread level parallelism (TLP)**
  - **Coarsed-grained parallelism vs finer-grained (ILP)**
  - **Partes del programa que seran ejecutado en paralelo**

Parallel Hardware

- Clasificación de Flynn
- Von Neumann : Single Instruction Stream, Single Data Stream SISD
  - Solamente ejecuta un instrucción a la vez
  - lee y escribe un dato a la vez

Parallel Hardware

- Single Instruction, Multiple Data SIMD (sistema paralelo)
  - Aplicar la misma instrucción sobre múltiples datos
  - Ej.: Un CPU con múltiples ALU, una instrucción llega los diferentes ALU

```
for (i = 0; i < n; i++)
        x[i] += y[i];
```

- Si tenemos n datos y m ALUs (m<n)
  - Podemos ejecutar bloques de m datos a la vez
  - Ej.: m=4 n=15, primero: 0-3,4-7,8-11, 12-14(uno de los ALU esperara)

Parallel Hardware

- Se requiere que todos los ALUs ejecuten las mismas instrucciones
    - Este puede degradar el rendimiento del sistema

```
for (i = 0; i < n; i++)
    if (y[i] > 0.0)
        x[i] += y[i];
```

- Si la condición no se cumple entonces el ALU esperará hasta que los demás hayan terminado
- SIMD -> ejecución sincronizada de las instrucciones entre los ALUs
- SIMD funcionan muy bien para procesar grandes array de datos
- **DATA-PARALLELISM**

Parallel Hardware

- Graphics processing units (GPU)
- Pipeline permite convertir geometría en un buffer de píxeles
- Utiliza tradicionalmente shaders
  - miniprogramas implícitamente paralelos
- SIMD, gran cantidad de ALUs
- No SIMD puro porque los GPUs actuales pueden ejecutar varios flujos de instrucciones a la vez.
- Varios lenguajes de programación han sido creados dedicados al GPGPU

High Performance Computing

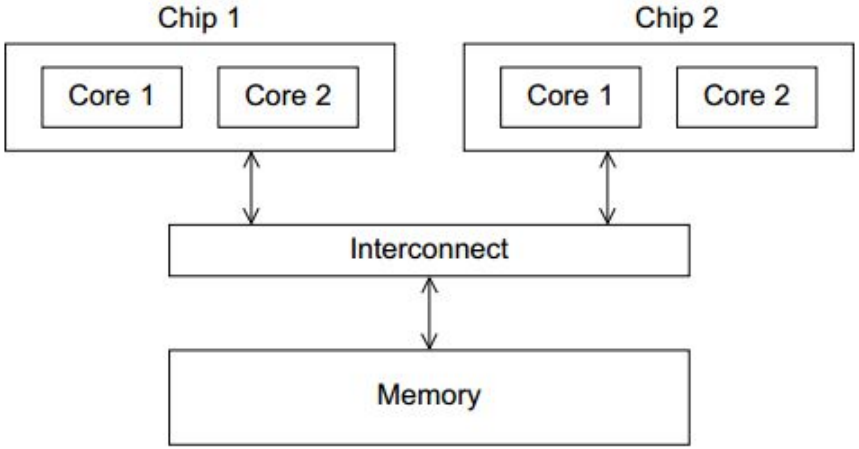
- Arquitectura heterogénea CPU-GPU
- Computación paralela consiste en llevar a cabo cálculos al mismo tiempo
  - Arquitectura de la computadora
  - Programación paralela

Parallel Hardware

- Multiple Instruction, Multiple Data MIMD
- Núcleos totalmente independientes
- Al contrario de SIMD, los sistemas MIMD son generalmente asíncronos
  - A un tiempo dado pueden ejecutar codigo completamente distintos aun así los forzamos
- MIMD shared memory
  - Todos los procesadores están conectados a una memoria global que todos pueden acceder
- MIMD Distributed memory
  - Todos los procesadores tienen su propia memoria y comunicarán a través de una red

MIMD Shared memory

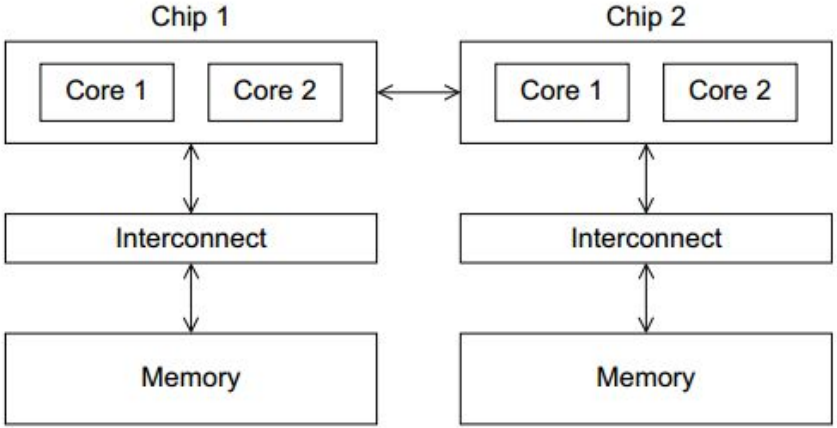
- uno o varios procesadores multi-núcleos
- Cores pueden tener Caché L1 privados y los otros niveles pueden ser compartidos



Uniform Memory Access

MIMD Shared memory

- Cada procesador tiene una main memory
  - Acceso más rápido que UMA
  - Programación es más compleja que UMA
  - Sigue posible acceder a la memoria de otro núcleo



Non-Uniform Memory Access

MIMD Distributed-memory systems

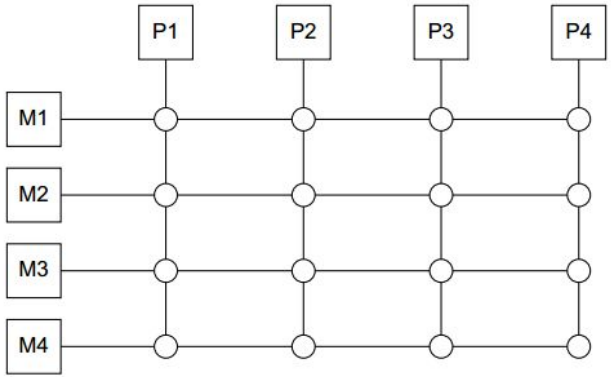
- Clusters !
  - En general PCs interconectados por una red (Ethernet)
  - Cada entidad individual generalmente un sistemas shared memory llamados **nodos**
  - **Sistemas híbridos**

Redes de interconexión

- La interconexión es crítica tanto en sistemas “shared” que “distributed”
- Puede degradar el rendimiento global

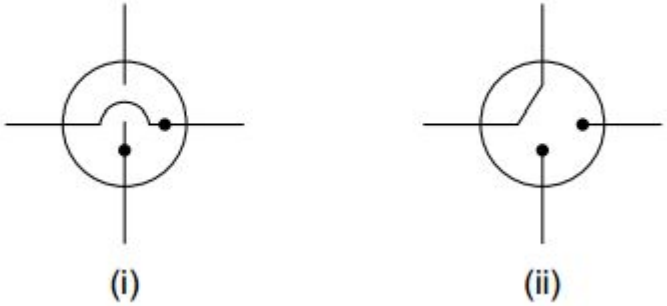
Shared-memory interconnects

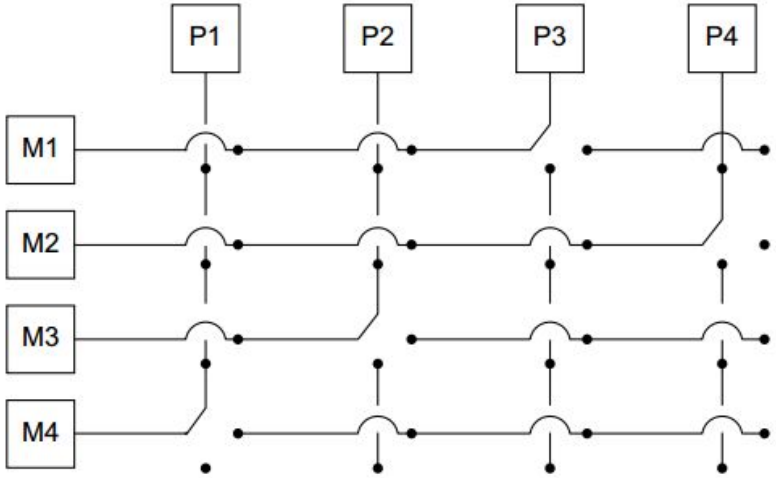
- Buses
  - Cables de comunicación paralelos
  - Compartidos entre los devices que están interconectados
  - Costo bajo y flexible (agregar un device)
  - Entre más devices más lento (los procesadores podrían esperar su turno)
- Crossbars (switched interconnects)



Shared-memory interconnects

- Crossbars (switched interconnects)
  - Los switches impiden el acceso de dos procesadores a la misma memoria
  - Permiten un acceso mucho más rápido que los buses



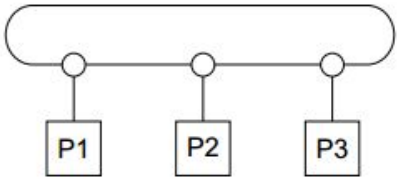


Distributed-memory interconnects

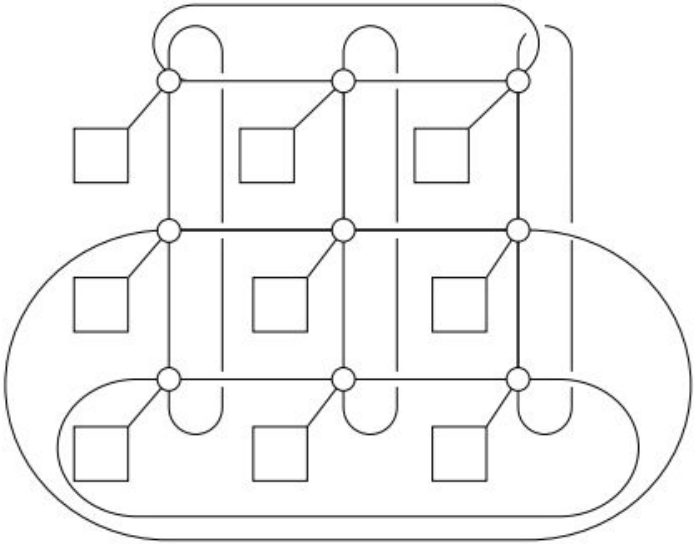
- Direct interconnect
- Indirect interconnect

Distributed-memory Direct interconnects

- Cada switch es conectado a un par processor-memoria
- Los switch están interconectados entre ellos (**ring** or **toroidal mesh**)



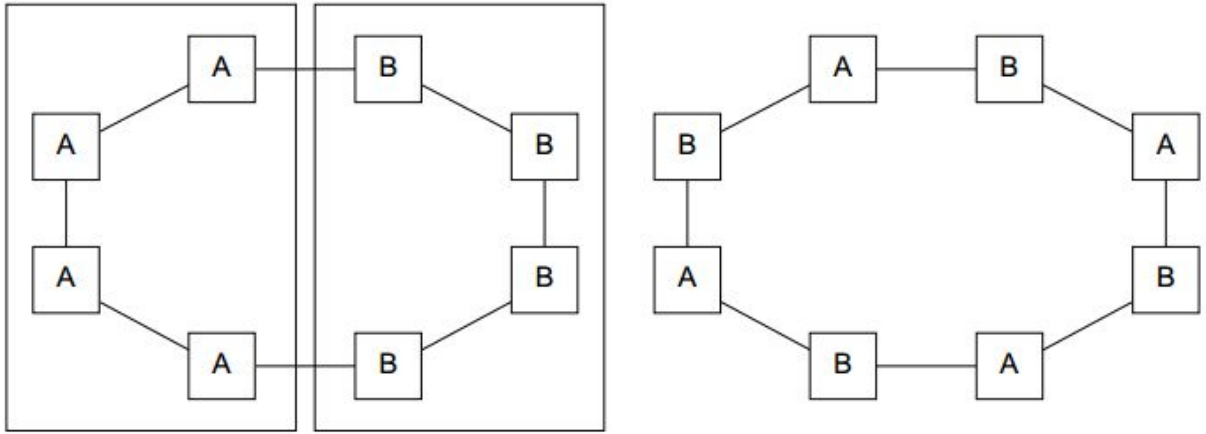
2p links,
Switch 3
puertos



3p links
Switch 5
puertos

Distributed-memory Direct interconnects

- Para medir la cantidad de conexiones simultáneas
  - Bisection width
  - Dividir el sistema en dos partes
  - Contar cuantas conexiones simultaneas puede haber entre los grupos
  - **En un ring -> 2**

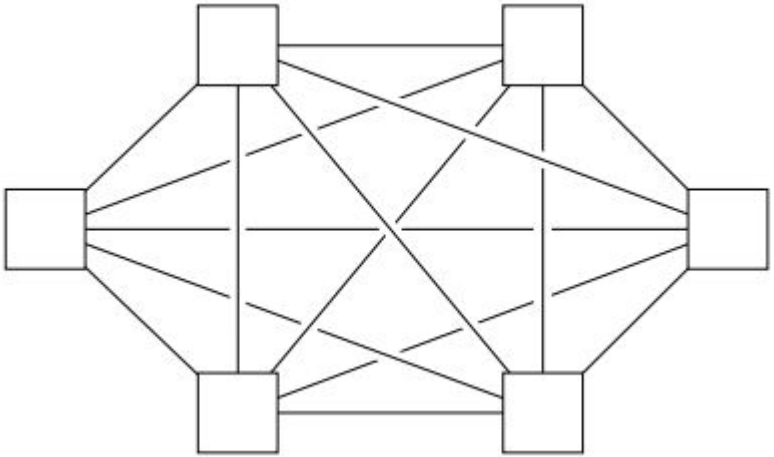


Distributed-memory Direct interconnects

- **Bandwidth**
  - Frecuencia de la transmisión de datos
- **Bisection** bandwidth representa la relación entre la calidad de la red y el ancho de banda
  - Si el bandwidth est 10 MB/s y el Bisection Width es 2
  - Entonces el bandwidth de la red es 20 MB/s !

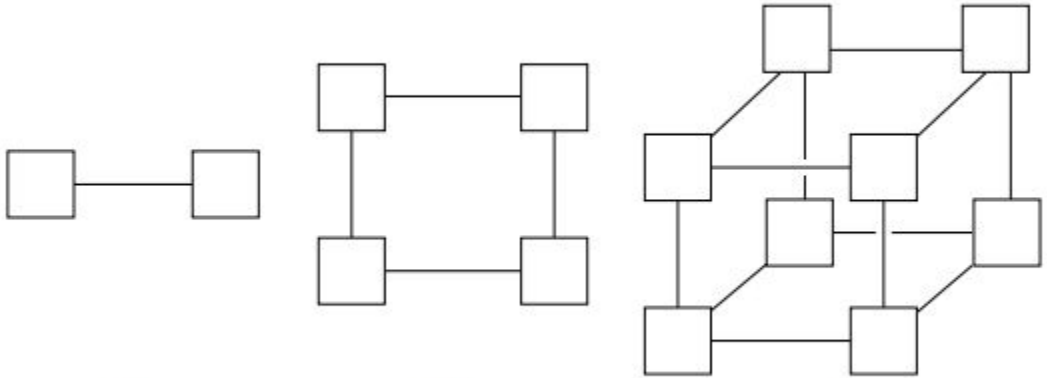
Distributed-memory Direct interconnects

- Fully interconnected network
- Bisection width $P^2/4$



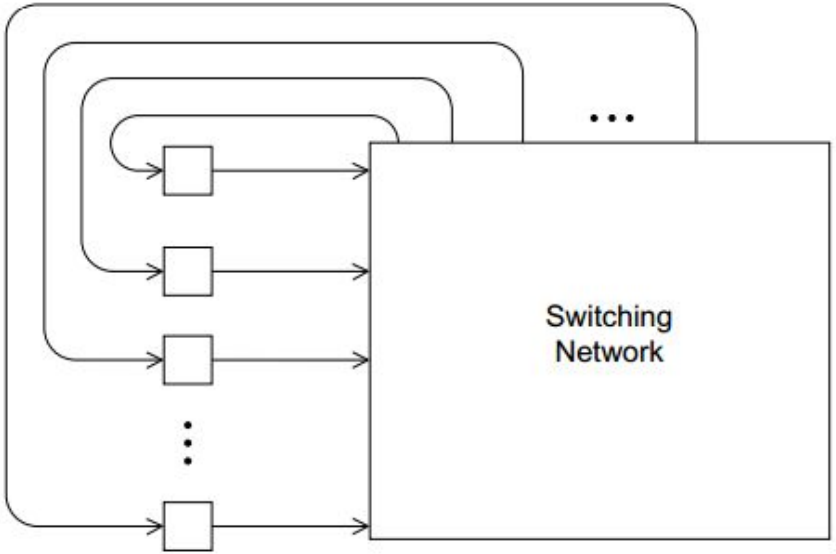
Distributed-memory Direct interconnects

- Hypercube
- d dimensión tiene $p = 2^d$
- Mejor que ring o toroidal (bisection p/2)
- Problema los switch deben ser muy potentes



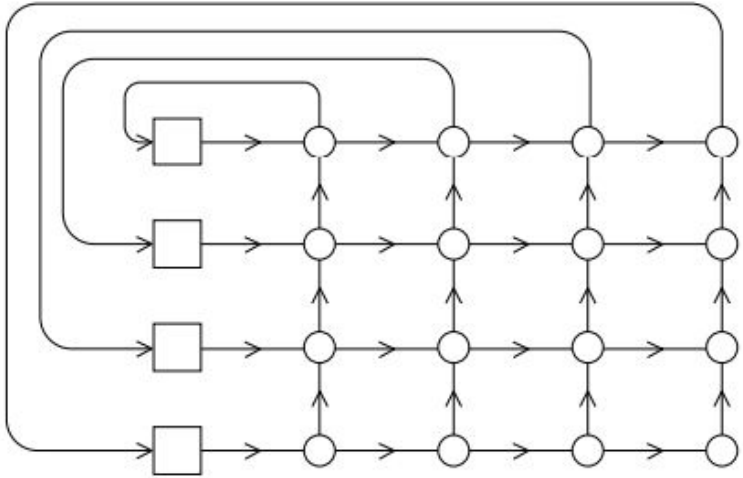
Distributed-memory Indirect interconnects

- Alternativa a la conexión directa
- Cada procesador tiene una entrada y salida hacia un switch



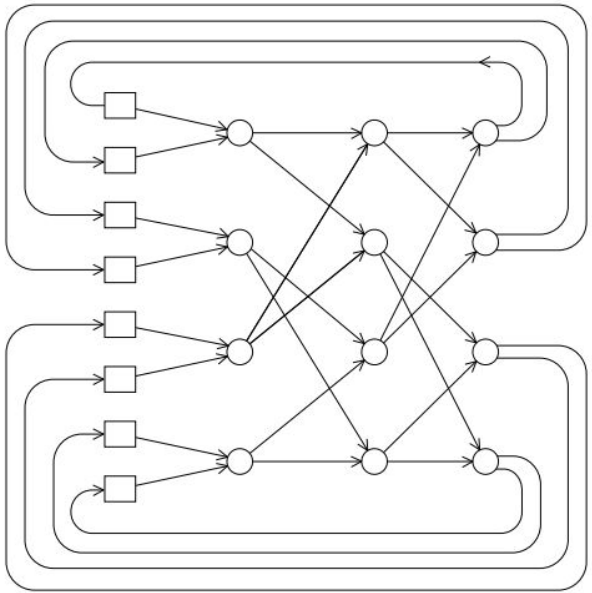
Distributed-memory Indirect interconnects

- **Crossbar** similar al shared memory en su estructura
- Puede haber un comunicación simultánea entre procesadores mientras dos procesadores no se quieren conectar con un mismo procesador



Distributed-memory Indirect interconnects

- **Omega** similar al crossbar (2-by-2 crossbar)
- Menos costoso que crossbar
- no permite siempre una comunicación simultánea
- Si procesador 0 manda a 6, **1 no puede mandar a 7**



Latencia y ancho de banda

- Queremos saber cuanto tiempo demora en viajar un dato
- main memory - cache, cache - registro o entre nodos de un sistema distribuido
- Latency
  - Cuanto tiempo demora en llegar
- Bandwidth
  - A qué frecuencia la destinación recibe los datos
  - message transmission time = l + n/b.
  - l latencia, n la cantidad de bytes, b bandwidth

Un poco mas de cache

- Coherencia
- 2 núcleos -> 2 cachés, x compartida inicializada a 2

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving x
2	Statement(s) not involving x	`z1 = 4*x;`

- Qué valor se almacenará en z1 ?
- 28 ? ya que core 0 cambia x
- 8 ?

- Lo más probable es 8 ya que x estará en el caché de core 0 y core 1 y cuando se modifica a 7, x tendría que regresar al main y a ser cargado en el cache del core 1.

Un poco mas de cache

- No importa si write through o write back
- No hay forma de ver dentro del caché de otro procesador
  - **Cache coherence problem**

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving x
2	Statement(s) not involving x	`z1 = 4*x;`

Un poco mas de cache

- **snooping cache coherence**
  - asegurar la coherencia
- Idea
  - basada en el bus, cada core conectado a un bus puede ver lo se transmite vía el bus
  - core 0 modifica **x** y lo manda a través del bus entonces core 1 podría verlo y marcar su **x** en caché como **inválido**
- Actualmente
  - se informa a los otros núcleos que x fue cambiado en el caché
  - Ya no es necesario un bus, solamente que exista un broadcast posible entre los núcleos
  - **write-through** es más simple el core tiene que ver las escrituras para detectar un cambio
  - **write-back** una comunicación extra es necesaria ya que no se escriba directamente en la main memory

Un poco mas de cache

- **Snooping cache coherence**
  - El broadcast entre todos los núcleos puede ser caro
  - No es escalable (distributed memory)
- **Directory-based cache coherence**
  - Estructura de datos -> Directory
  - Almacena el estado de cada cache line
  - Cada núcleo mantiene su parte del directory
  - Sí Core 0 lee un dato, en el directory es marcado como presente en el caché
  - Si el dato es cambiado entonces miramos en el directory y los cores que tenían el dato verán su caché invalidado
  - La ventaja es que solamente se contacta los cores que tienen el dato

False sharing

- Paralelizar el bucle exterior

```
int i, j, m, n;
double y[m];

/* Assign y = 0 */
. . .

for (i = 0; i < m; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);
```

```
/* Private variables */
int i, j, iter_count;

/* Shared variables initialized by one core */
int m, n, core_count
double y[m];

iter_count = m/core_count

/* Core 0 does this */
for (i = 0; i < iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);

/* Core 1 does this */
for (i = iter_count+1; i < 2*iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);
```

False sharing

- Paralelizar el bucle exterior m = 8
- double = 8 bytes - cache line = 64 bytes
- entonces **y** está puesto en caché enteramente
- y[i] += f(i,j) invalida el cache line de los demás cores
- entonces si **n** es grande se hará muchos intercambios con el main memory a pesar de que core 0 y core 1 **nunca** quieren escribir o leer en un mismo elemento de **y**
- **No hay resultado incorrecto pero sí una pérdida en el rendimiento**
- **Para evitar eso se puede creer una variable temporal que almacene la suma para cada core**
- **Veremos más adelante como resolver eso**

```
/* Private variables */
int i, j, iter_count;

/* Shared variables initialized by one core */
int m, n, core_count
double y[m];

iter_count = m/core_count

/* Core 0 does this */
for (i = 0; i < iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);

/* Core 1 does this */
for (i = iter_count+1; i < 2*iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);
```

Shared Memory VS Distributed Memory

- Shared memory
  - Es más simple de programar
  - Es mucho más costoso a escalar
  - Entre más procesadores más probabilidad de crear conflictos de accesos a la memoria
  - La comunicación es mucho más rápida entre los núcleos
- Distributed memory
  - Resulta ser mucho menos costoso
  - Se adapta muy bien a grandes cantidades de datos

Parallel software

Parallel Software

- Los software no son todos paralelos
- Excepto
  - Servidores web
  - Sistemas de BDD
  - OS
- Tenemos que aprender a programar de forma paralela
- Shared-memory
  - Single process -> fork multiple thread
- Distributed-memory
  - Multiples procesos para resolver tareas

Coordinación de threads/procesos

- A veces paralizar un algoritmo es fácil **(embarrassingly parallel)**
  - ej. bucle for
- Dividir el trabajo a los diferentes threads
  - De forma equitativa (**load balancing**)
  - Minimizando la comunicación entre threads
- Sincronizar los threads
- Comunicar entre threads

Shared memory

- Variables privadas
  - Cada threads tiene sus propias variables
- Variables compartidas
  - Comunicación entre threads
- Comunicación **implícita**
  - **Lo hacemos a través de variables compartidas**

Shared memory - Threads dinamicos y estaticos

- Threads dinámicos
  - en cualquier momento el master puede crear un grupo de threads
  - Ej. un máster espera un trabajo, cuando llega crea un thread para realizarlo
  - Los recursos son utilizados solamente durante la “vida” del thread
- Threads estáticos
  - Los threads son creados al setup por el master y terminas al fin del trabajo
  - los recursos son utilizados hasta que todos los threads hayan terminado
  - mejor rendimiento que dynamic thread

Shared memory - Non determinístico

- Sistemas MIMD
  - Non determinístico
  - Una misma entrada puede resultar en una diferente salida
- Normalmente no es un problema
- A veces puede llevar a resultados incorrectos
  - Dos threads acceden a un mismo dato para cambiarlo
  - **Race condition**
  - **Critical section**
  - Es la responsabilidad del programador de asegurar una exclusión mutua entre threads

Shared memory - Non determinístico

- **exclusión lock or mutex or lock (Similar a semaphores, monitor)**

```
my_val = Compute_val(my rank);
Lock(&add_my_val_lock);
x += my_val;
Unlock(&add_my_val_lock)
```

- Con un mutex forzamos la **serialización**

Shared memory - Non determinístico

- **busy-waiting**

```
my_val = Compute val(my_rank);
if (my_rank == 1)
        while (!ok_for_1); /* Busy−wait loop */
        x += my_val; /* Critical section */
if (my rank == 0)
        ok_for_1 = true; /* Let thread 1 update x */
```

- puede ser costoso ya que un núcleo estará ocupado en verificar constantemente la condición del while

Shared memory - Thread safety

- **Se trata de calificar una función thread safe o no**
- **Variable Static C**
  - Compartida entre la llamadas de una función
  - Compartida entre los threads
- Existen funciones que no son seguras de usar con threads ya que podrían producir un comportamiento no definido
- Listado

Distributed memory

- Message passing
- Procesos se identifican con Ids

```
char message[100]; . . .
my rank = Get rank();
if (my rank == 1) {
        sprintf(message, "Greetings from process 1");
        Send(message, MSG CHAR, 100, 0);
}
else if (my rank == 0) {
        Receive(message, MSG CHAR, 100, 1); printf("Process 0 > Received: %s\n", message);
}
```

Distributed memory

- Código SPMD
  - Comparten el mismo código
  - Lo que harán dependen de su rango
  - message se refiere a dos diferentes block en memoria

```
char message[100]; . . .
my rank = Get rank();
if (my rank == 1) {
        sprintf(message, "Greetings from process 1");
        Send(message, MSG CHAR, 100, 0);
}
else if (my rank == 0) {
        Receive(message, MSG CHAR, 100, 1); printf("Process 0 > Received: %s\n", message);
}
```

Distributed memory

- Es posible que *send* no retorne hasta que no se empieza a recibir (*receive()*)
- Receive no retorna antes de haber recibido el mensaje
- No son las únicas posibilidades o comportamiento (MPI)

```
char message[100]; . . .
my rank = Get rank();
if (my rank == 1) {
        sprintf(message, "Greetings from process 1");
        Send(message, MSG CHAR, 100, 0);
}
else if (my rank == 0) {
        Receive(message, MSG CHAR, 100, 1); printf("Process 0 > Received: %s\n", message);
}
```

Distributed memory

- Los API para mandar mensajes proveen otras funciones
- Broadcast
  - Un solo proceso transmite datos a los demás
- Reduction
  - Los resultados calculados independientemente por los procesas son combinados en un resultado
- *Veremos esto con MPI*

Distributed memory

- Los API para mandar mensajes proveen otras funciones
- Broadcast
  - Un solo proceso transmite datos a los demas
- Reduction
  - Los resultados calculados independientemente por los procesas son combinados en un resultado
- *Veremos esto con MPI*

Distributed memory

- API de Message-Passing es muy potente
- Es de bajo nivel, el programador tiene que manejar muchos aspectos al detalle
  - El lenguaje ensamblador de la programación paralela
  - Existen muchos intentos de crear API de alto nivel

Distributed memory

- One-sided communication
  - Escribir directamente en la memoria de otro proceso
  - Comunicación poco costosa ya que involucra un solo proceso
  - Problema de sincronización antes de copiar el dato en la memoria de otro proceso
  - Como avisar al otro proceso que dispone de un nuevo dato
    - Variable flag
    - Overhead del proceso que mira el flag para ver si cambia

Sistemas híbridos

- Cluster de procesadores multi-núcleos
- Se puede utilizar un API de shared-memory junto a un API de messages passing para comunicar entre nodos
- Puede resultar en programas bastante complejos

Input/output

I/O

- Problema bastante complejo para todos los sistemas paralelos
- La mayoría de las funciones I/O son pensadas secuenciales
- Non deterministic (ej. printf)
- Que pasaria con scanf ?
    - Deberíamos de autorizar el uso de scanf a un solo proceso o thread

I/O

- Reglas
  - DM, solo el proceso 0 puede acceder a stdin
  - SM solo el master puede acceder a stdin
  - Todos los procesos o threads pueden acceder a stdout (útil para debug)
  - Lo mejor (para evitar los problemas non deterministic) solo un thread lo usara
  - No leer/escribir dentro de un mismo archivo
  - debug -> incluir en rank del thread

Performance

Performance

- El objetivo principal de paralelismo es mejorar el rendimiento de los programas
- Speedup
- Eficiencia
- Amdahl’s law
- Escalabilidad

Eficiencia y speedup

- Lo ideal es de dividir equitativamente el trabajo en los núcleos sin agregar complejidad al trabajo
- P cantidad de núcleos
- Tparallel=Tserial/p <- speedup lineal
- Es muy poco probable ya que utilizar el paralelismo siempre implica algún costo

    - SM -> Secciones críticas (secuencial)
    - DM -> mandar datos a través de la red
    - Entre mas threads o procesos más probable de incrementar el costo

Eficiencia y speedup

- Speedup
- Speedup lineal S=p
- Entre mas incrementamos p -> S deberia de alejarse mas y mas de speedup ideal lineal
- S/p es lo que llamamos la eficiencia

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

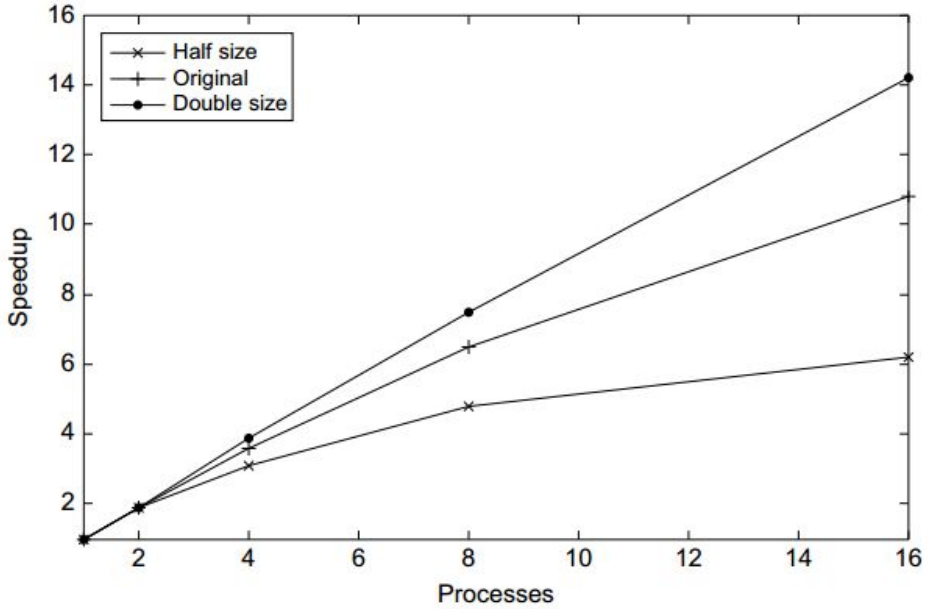
$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}.$$

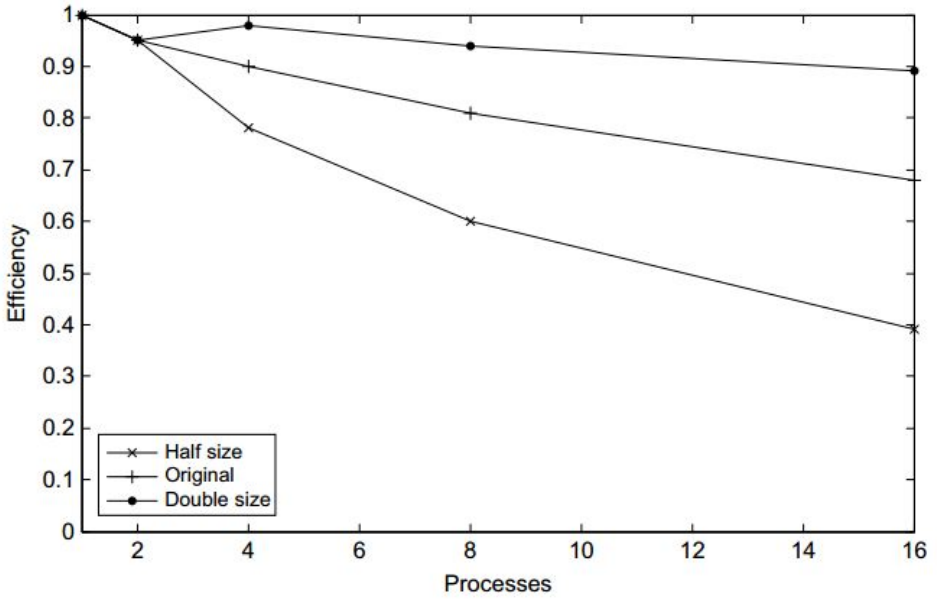
p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
E = S/p	1.0	0.95	0.90	0.81	0.68

Eficiencia y speedup

- También depende del tamaño del problema

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89





Eficiencia y speedup

- También depende del tamaño del problema
- Tparallel = Tserial/p + Toverhead.
- Toverhead crece lentamente comparado con el tamaño del problema
- **Cada thread trabaja más** así que el tiempo de coordinación etc, se vuelve **menos importante**

Ley de Amdahl

- Gene Amdahl en los 60
  - Si todo un programa no es paralelizado al 100% entonces el speedup será limitado y eso sin importar la cantidad de núcleos
- Ej. Programa perfectamente paralelizado al 90%
  - Tserial = 20 s
  - **Tparallel = 0.9 * Tserial / p + 0.1 * Tserial = 18/p + 2**

$$S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}}} = \frac{20}{18/p + 2}.$$

Ley de Amdahl

$$S \leq \frac{T_{\text{serial}}}{0.1 \times T_{\text{serial}}} = \frac{20}{2} = 10$$

- El límite de S entre más crece P es 10
- De forma general si r es la fracción del programa secuencial
  - Limite es 1/10
- Esto es un problema real ?
  - No, porque no se toma en cuenta el tamaño de problema
  - La parte serial se vuelve en general menos importante
  - En general obtener un speedup de 5, 10 etc es ya bastante interesante !

Escalabilidad

- Un programa escalable
  - Aumentar la cantidad de thread y del tamaño del problema de tal manera que podemos conservar la misma eficiencia
- Si **no necesitamos** cambiar el tamaño del problema el programa es
  - **Strongly scalable**
- Si **necesitamos** cambiar el tamaño del problema a la misma frecuencia que p el programa es
  - **Weakly scalable**

Medir tiempos

- Es importante monitorear el tiempo para confirmar el comportamiento de un programa
- DM, monitorear el tiempo que un proceso para esperando
- Tiempo de ejecución de partes del programa
- Tiempo global de la ejecución de un programa no es muy interesante ya que muchas veces incluye la lectura de los datos etc.
- **Wall time**
  - **El tiempo de ejecución de la parte interesante**

```
double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

Medir tiempos

- **Wall time**
  - **MPI_Wtime**
  - **omp_get_wtime**
- **Cuidado multiples threads**

```
/* Synchronize all processes/threads */
Barrier();
my_start = Get_current_time();

/* Code that we want to time */
. . .

my_finish = Get_current_time();
my_elapsed = my_finish - my_start;

/* Find the max across all processes/threads */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
   printf("The elapsed time = %e seconds\n", global_elapsed);
```

Medir tiempos

- **Los tiempos pueden variar de un reporte a otro en el mismo sistema**
- **Se guarda el menor de los tiempos**

Diseño de programa paralelo

Diseño de programa paralelo

- Cómo paralelizar un programa secuencial ?
- La idea es repartir el trabajo de forma equitativa sobre los threads o procesos de un sistema
- No existe un proceso predefinido para realizar eso
    - No hay solucion universal

Diseño de programa paralelo

- Grandes etapas (Ian Foster)
- Particionar
  - Identificar las partes paralelizables
  - Dividir una tarea grande en pequeñas tareas simultáneas
- Comunicar
  - Determinar cuando los procesos tienen que comunicar entre ellos
- aglomeración o agregación
  - Combinar tareas y comunicación en secuencias de tareas más grandes
- Mapear
  - Asignar las tareas identificar a grupos de procesos o threads para que cada threads tenga la misma cantidad de trabajo