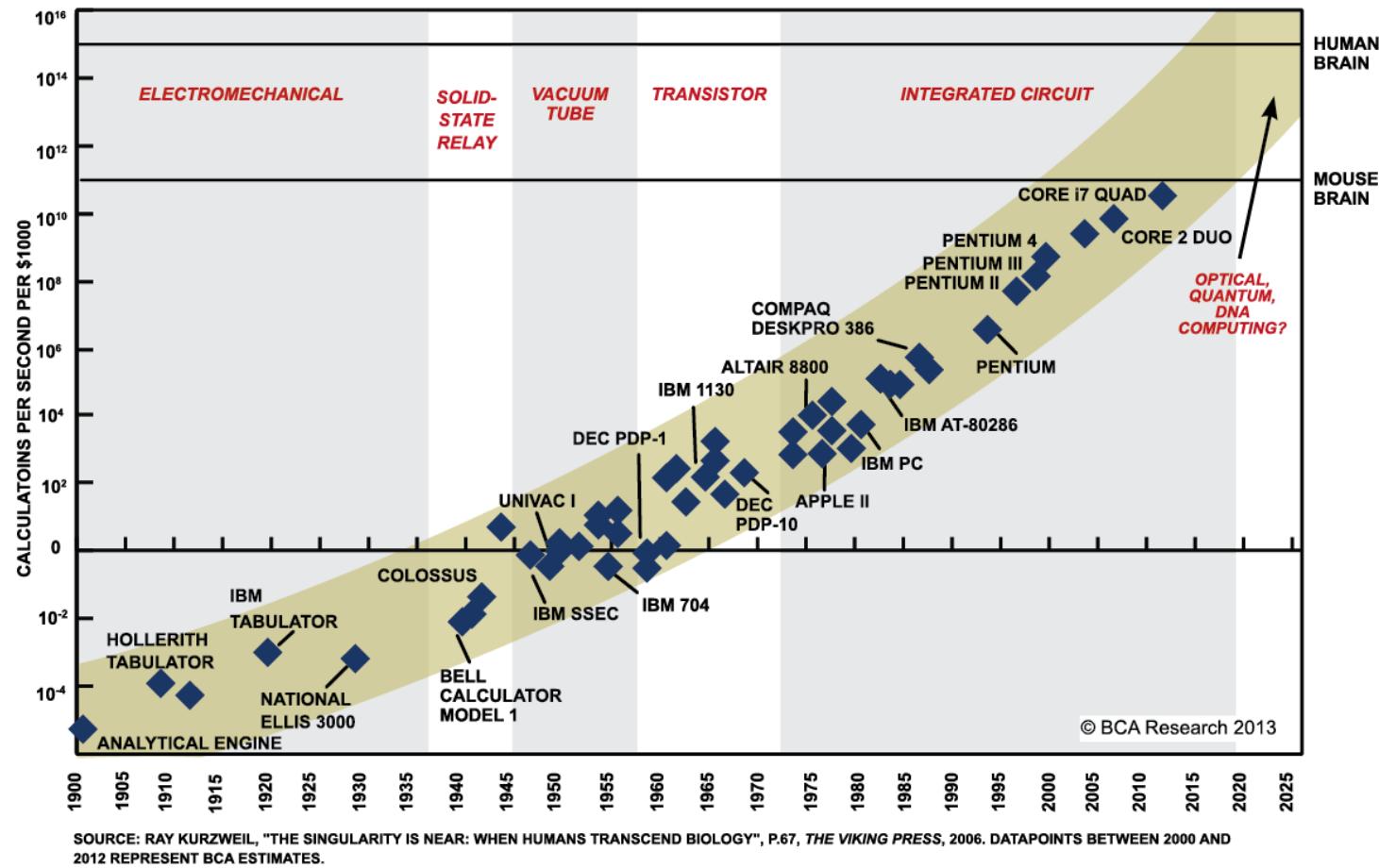


# Algoritmos Paralelos

MPI vs OPENMP  
(clase 24.11.15)

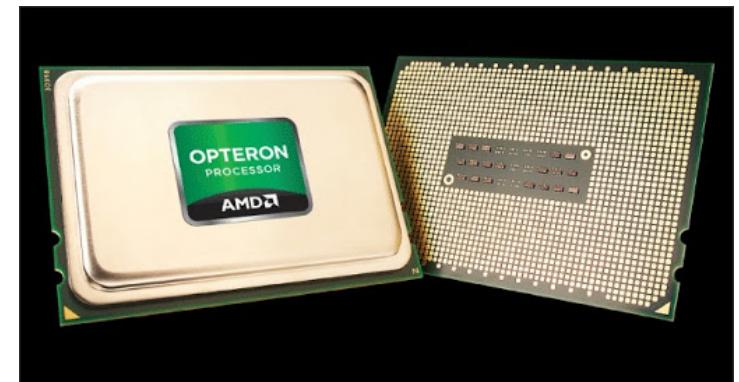
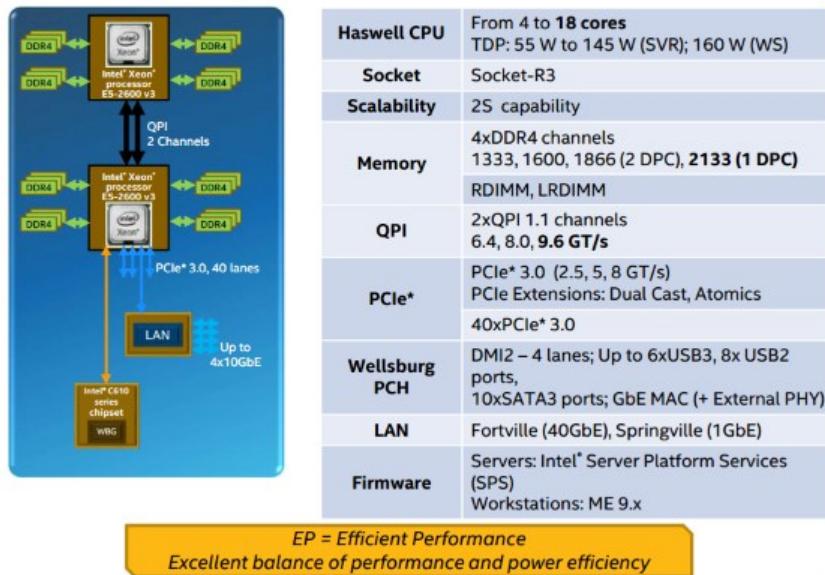
Prof. J.Fiestas

**Ley de Moore:** el número de transistores que puede contener un circuito integrado a un costo razonable, se duplica cada dos años.



**El número de núcleos por chip sigue creciendo:** Intel Haswell-EP chips tienen hasta 18 núcleos; AMD Abu Dhabi chips tienen 16 núcleos; IBM Blue Gene tiene núcleos de baja frecuencia pero en gran número

## Intel® Xeon® E5-2600 v3 Platform Summary



## **Sin embargo, problemas debido a que:**

- Poder de procesadores crece mas rápido que mejoras en acceso a memoria
- Latencias de memoria decrecen lentamente
- Número de núcleos por módulo de memoria crece

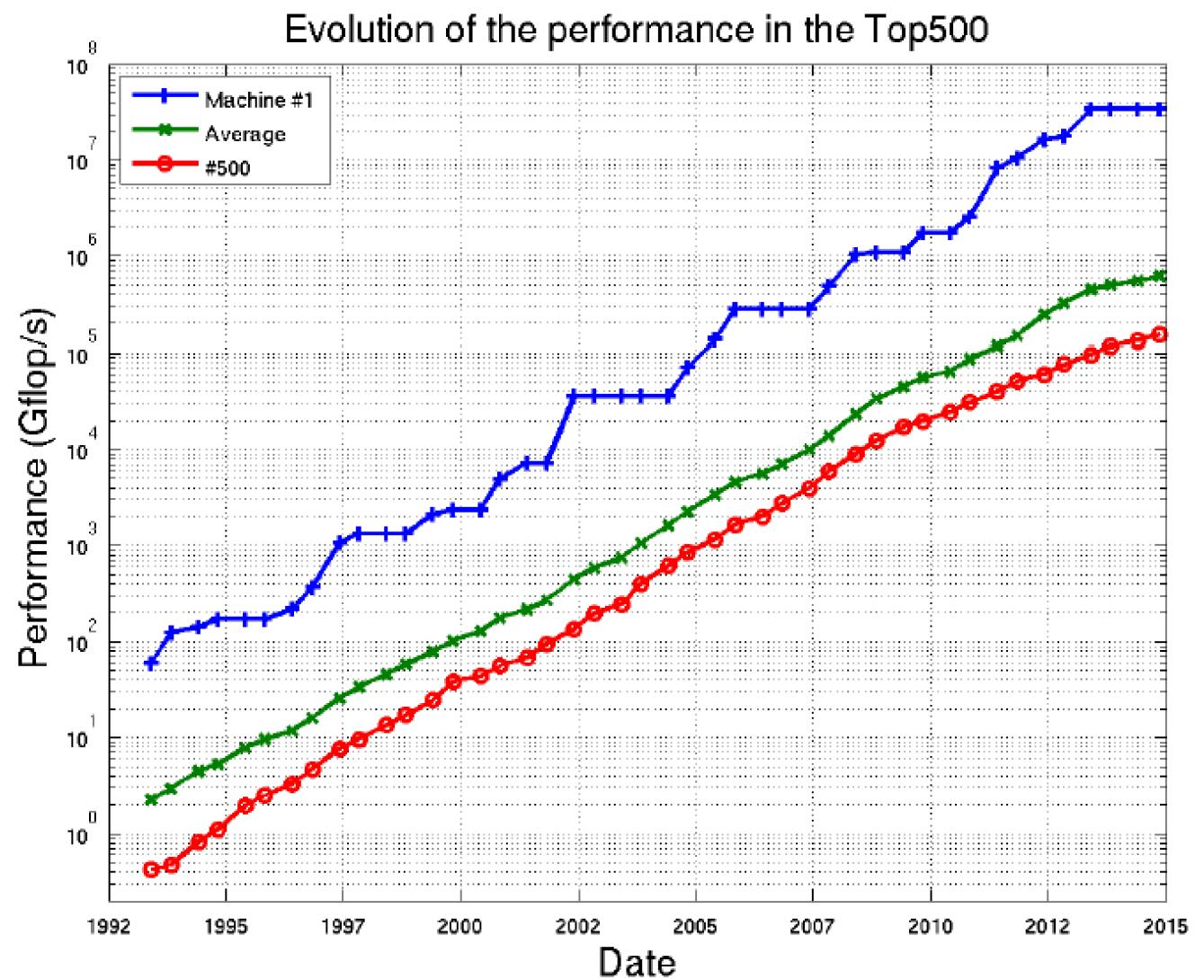
## **Por ello:**

- La distancia entre velocidad de memoria y la eficiencia teórica de los núcleos, crece
- Procesadores pierden tiempo (ciclos) mientras esperan la información
- Dificultad en explotar al máximo eficiencia teórica de procesadores

## **Soluciones:**

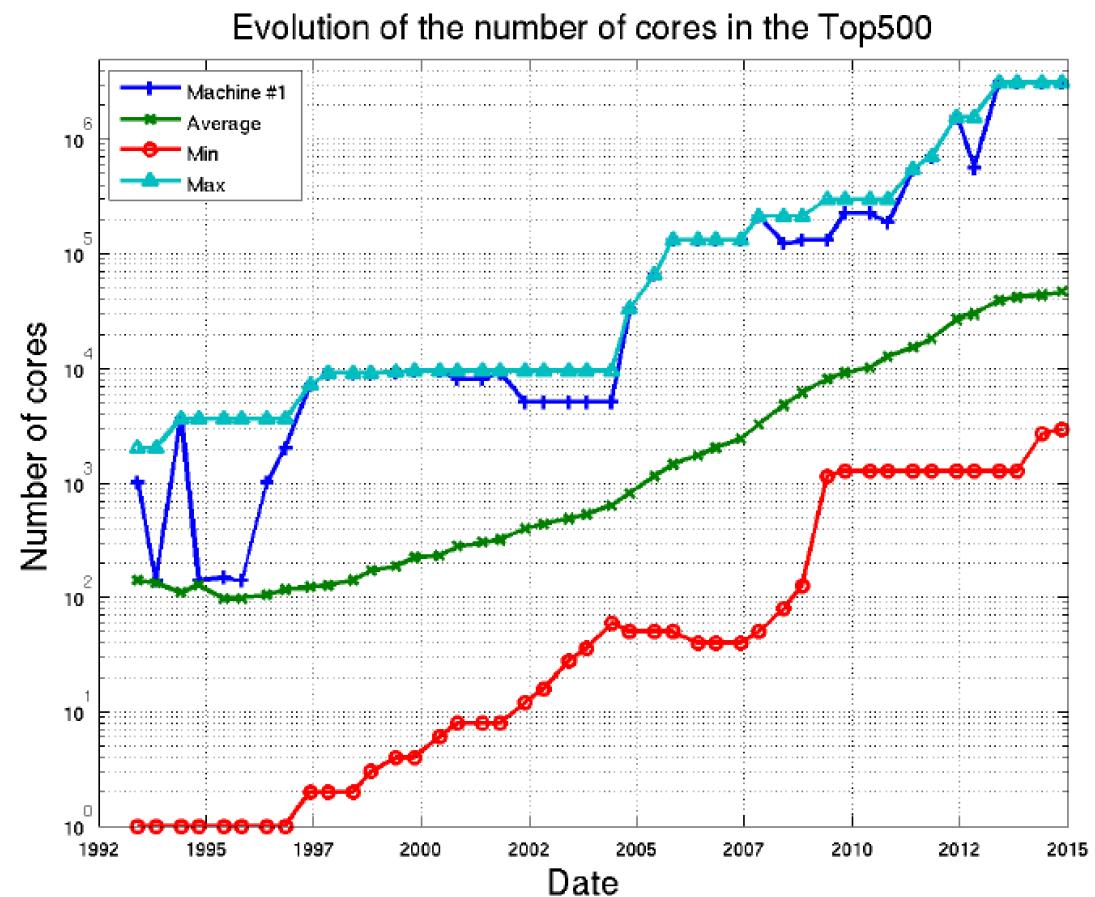
- Añadir mas memoria cache es esencial
- Paralelización a través de bancos de memoria (DDR-SDRAM, como en arquitecturas vectoriales)

El poder computacional en **Supercomputadoras** se dobla cada año

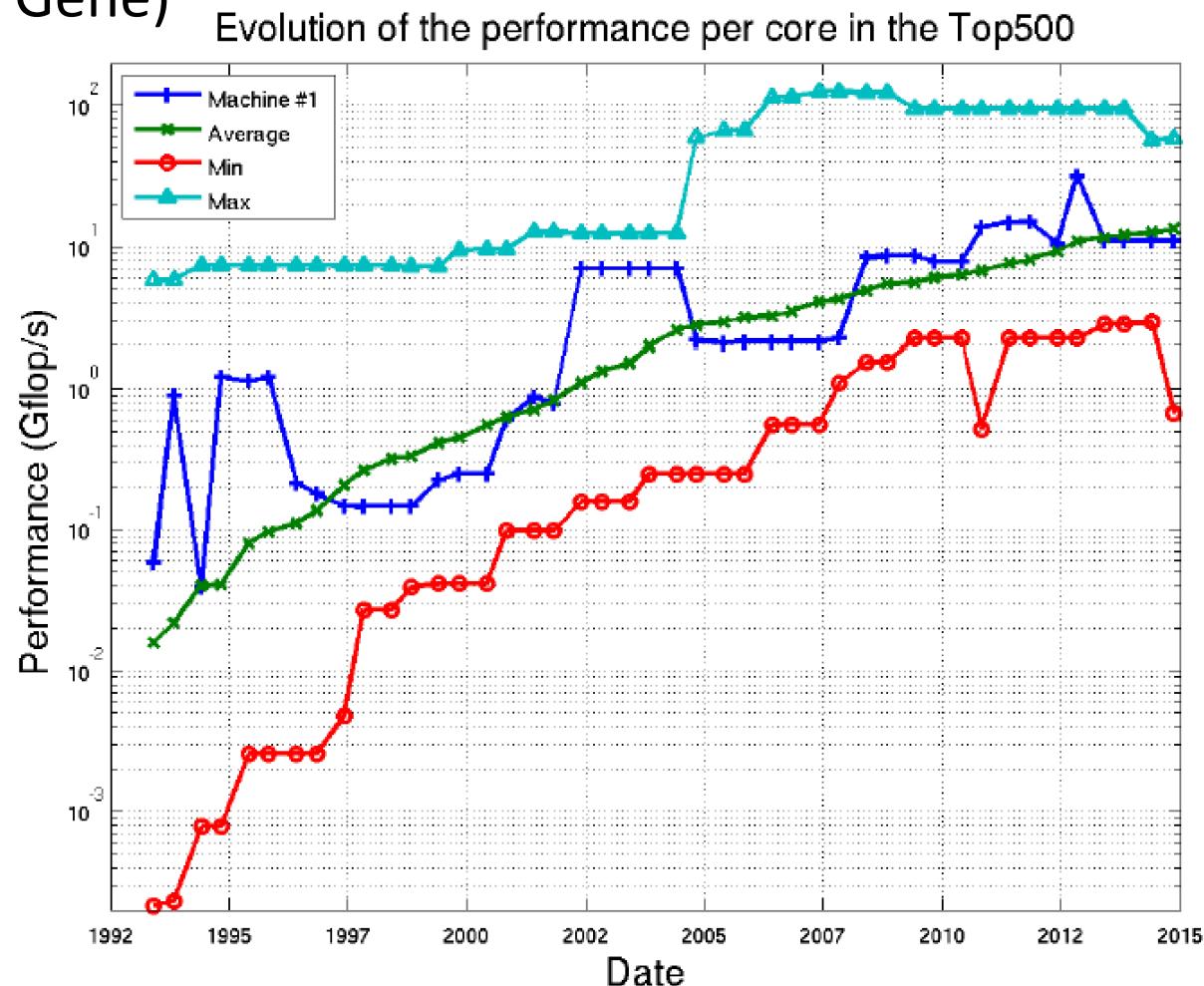


El número de núcleos se incrementa rápidamente (parallelismo masivo y arquitecturas multi-núcleo)

Arquitecturas híbridas aparecen (GPU o XeonPhi con standard CPUs)



Arquitecturas de computadores se vuelven mas complejas (procesadores/núcleos, acceso de memoria, network e I/O)  
Memoria por núcleo empieza a saturarse, asi como el performance por núcleo (IBM Blue Gene)



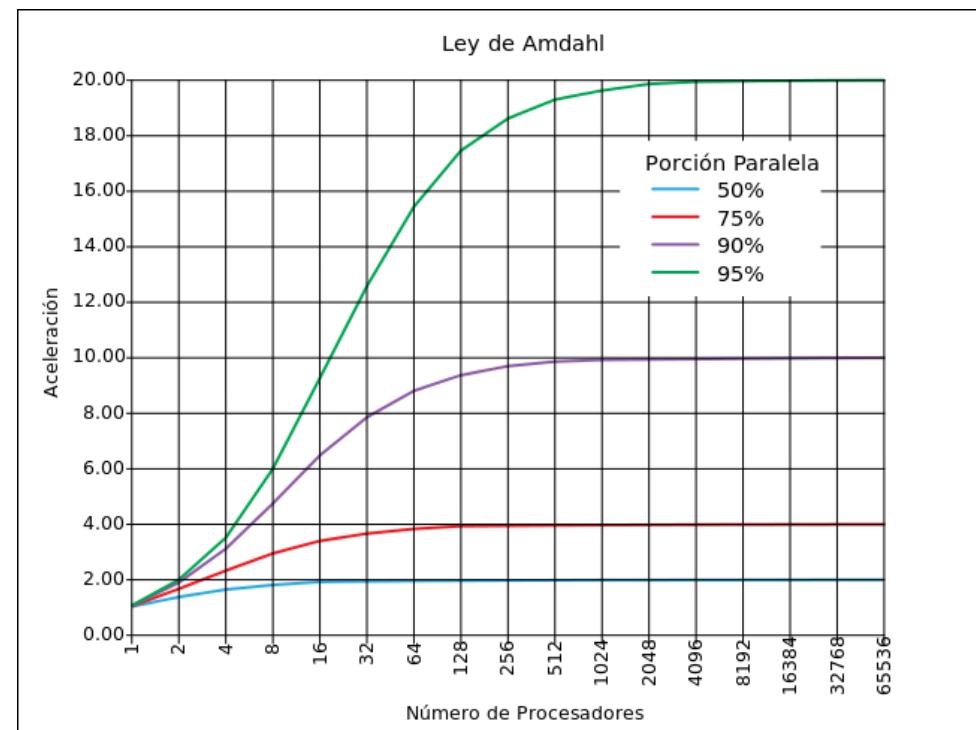
La **Ley de Amdahl**, establece que

“La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente”

$$Sp(P) = \frac{T_s}{T_{//}(P)} = \frac{1}{\alpha + \frac{(1-\alpha)}{P}} < \frac{1}{\alpha} \quad (P \rightarrow \infty)$$

Con  $S_p$  como la aceleración del rendimiento,  $T_s$  el tiempo de ejecución secuencial (1 procesador),  $T_{//}(P)$  el tiempo de ejecución de un código ideal paralelizado en  $P$  núcleos, y con  $\alpha$  como la parte no paralelizable de la aplicación

Es decir, independientemente de número de núcleos, la aceleración es siempre menor que la inversa del porcentaje representado por la parte secuencial



## Por consecuencia:

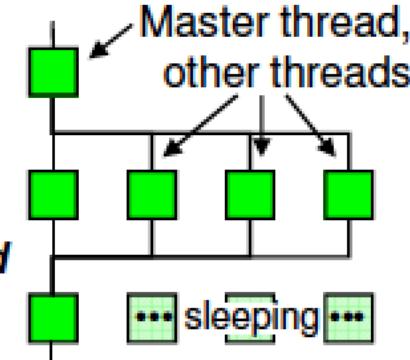
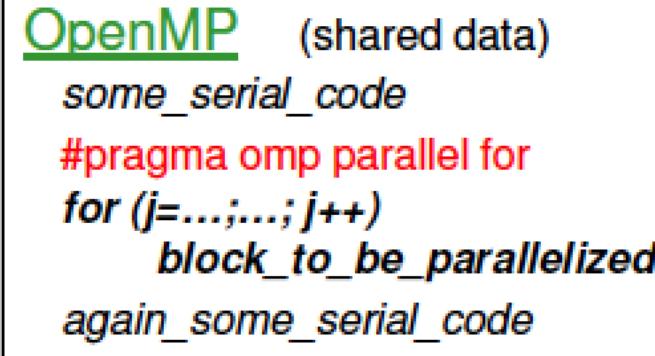
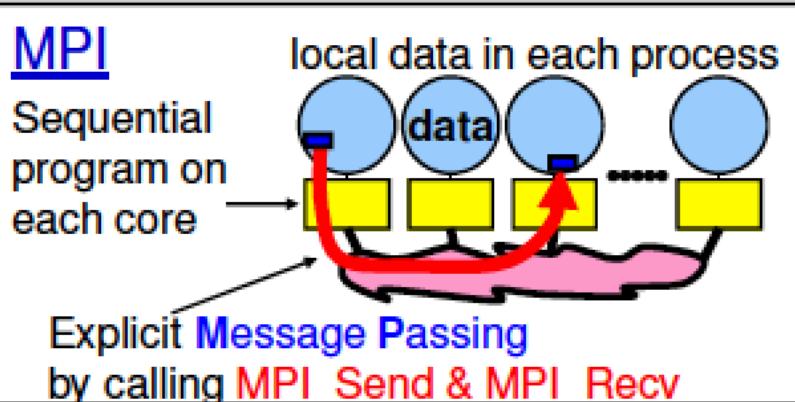
- Es necesario explotar un **número grande de núcleos “lentos”**
- La memoria individual de procesadores no aumenta, por ello no hay que desperdiciarla
- Se necesita niveles altos de **paralelismo para arquitecturas modernas**
- **Ingreso/Salida** de información cobrando importancia recientemente

## Consecuencias para desarrolladores de software:

- No es suficiente “esperar” para obtener mejor performance (poder computacional por núcleo se está saturando)
- Mayor necesidad de comprender la **arquitectura de hardware**
- Desarrollar códigos se vuelve **mas complejo** (trabajo en equipo)

## Evolución de métodos de programación:

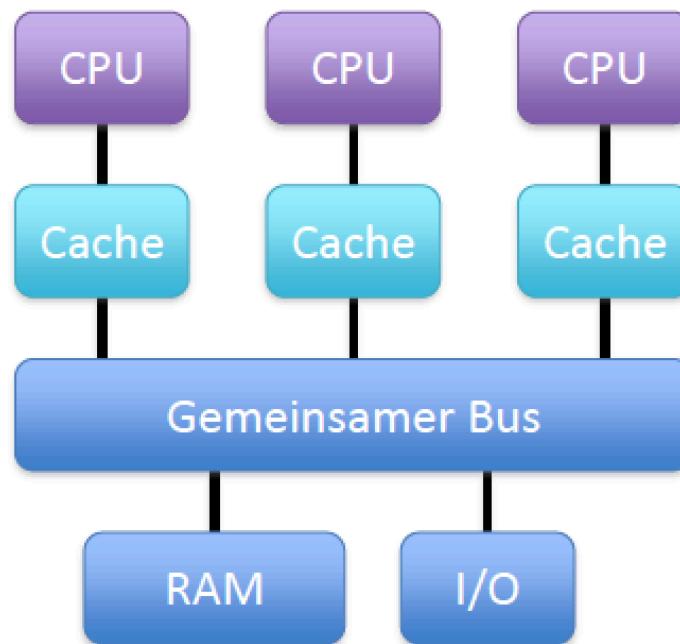
- MPI aún predomina y se mantendrá por buen tiempo (tanto en desarrollo como en aplicaciones)
- El híbrido MPI-OPENMP está siendo mas usado, sobre todo en supercomputadores
- Programación en GPU se incrementa, aunque las técnicas necesitan desarrollo
- Otros tests de programación híbrida (MPI+GPU,...), generalmente siempre usando MPI
- Nuevos lenguajes de programación paralela (UPC, Coarray-Fortran, PGAS, X10, Chapel, ...) aunque en etapa experimental.



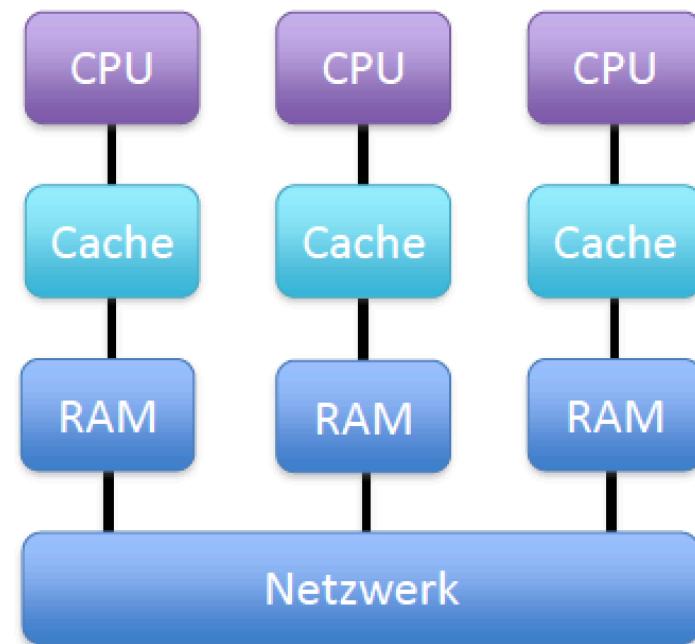
# MPI (Message Passing Interface)

VS

# OPENMP (OPEN MultiProcessing)



OPENMP: memoria compartida

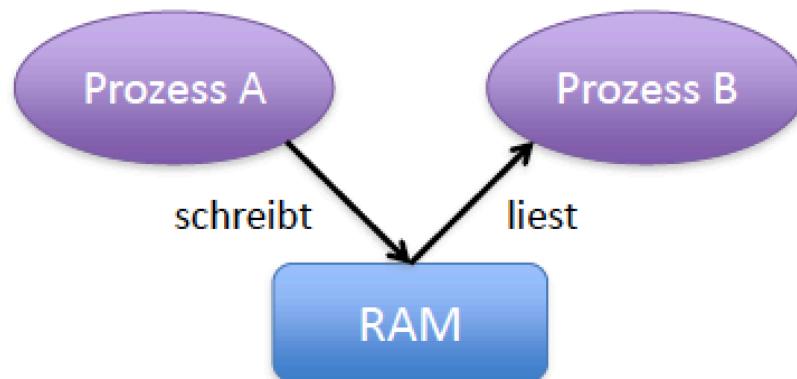


MPI: memoria distribuída

# Comunicación:

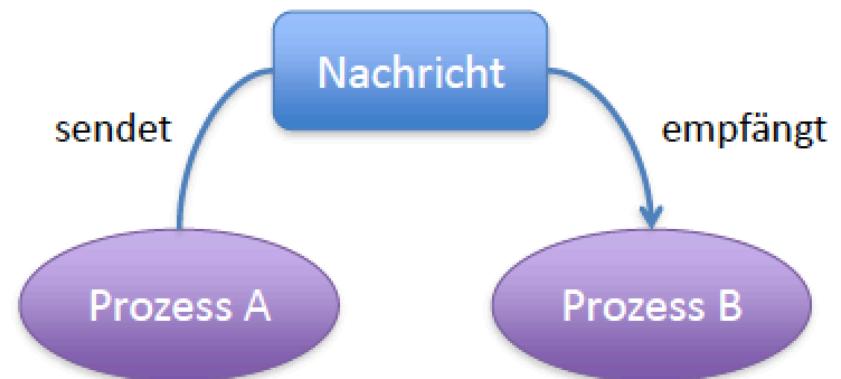
## Memoria compartida:

- Se escribe y lee del RAM
- Se sincroniza de acuerdo a competencia entre tareas



## Memoria distribuída:

- Se envia y reciben mensajes
- Solo se utilizan variables locales



## Proceso:

- Consiste de su propia dirección (para pilas (stack), códigos, etc)
- Puede alojar y reservar memoria
- Posee uno o mas threads, que ejecutan el programa

## Thread:

- Pertenece a un proceso
- Consiste en un Stack propio y registro en CPU
- Tiene la dirección del proceso al que pertenece (threads del mismo proceso no estan protegidos entre ellos)

## Problemas:

- **Competencia:** situaciones en las que la competencia de procesos al acceso a fuentes comunes tiene repercusiones en la ejecución del programa. I.e., cuando threads escriben en la misma unidad de memoria

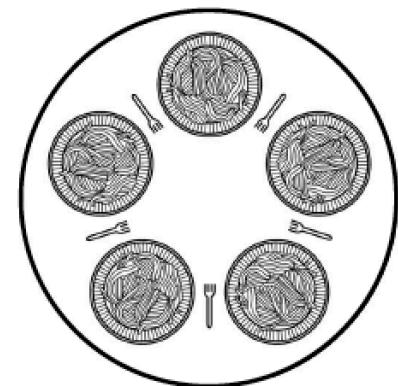
**Solución:** sincronizar las tareas (Mutex, Semaphore, Barrier)

## Problemas:

- **Sincronización desadecuada:** origina tiempos de espera a las fuentes de otros (deadlock), o el caso en que algunos threads no acceden a memoria por mucho tiempo.

### Ej. Problema de los filósofos:

Un filósofo solo piensa y come, y necesita dos tenedores para comer, pero puede tomar solo uno a la vez. Por ello, cada uno toma el tenedor izquierdo y espera por el otro.

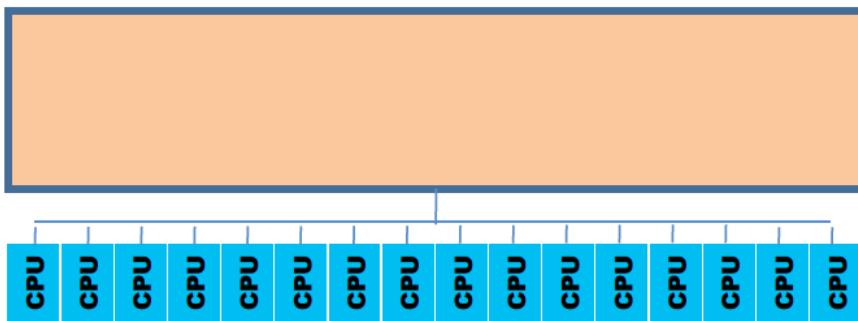


### Solución:

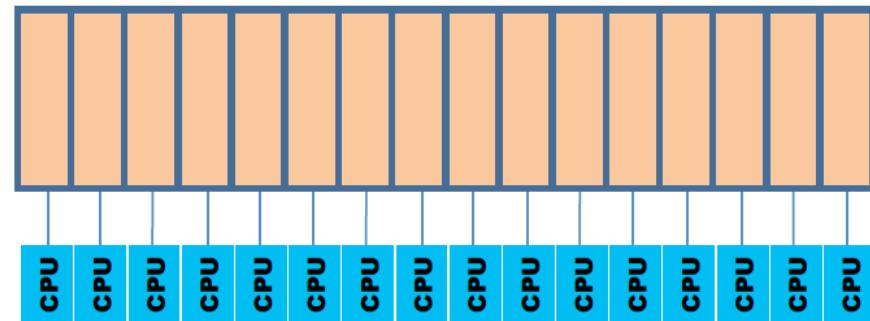
Aplicar Mutex (Mutual Exclusion) para comprobar y tomar dos tenedores

# Paralelismo híbrido

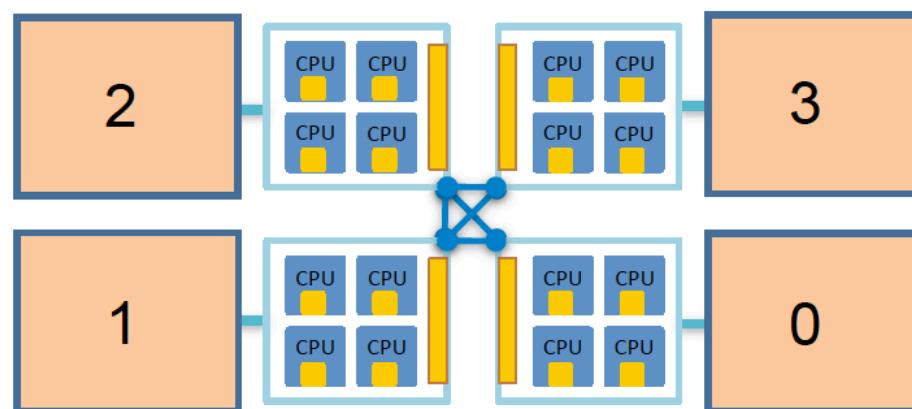
OpenMP



MPI

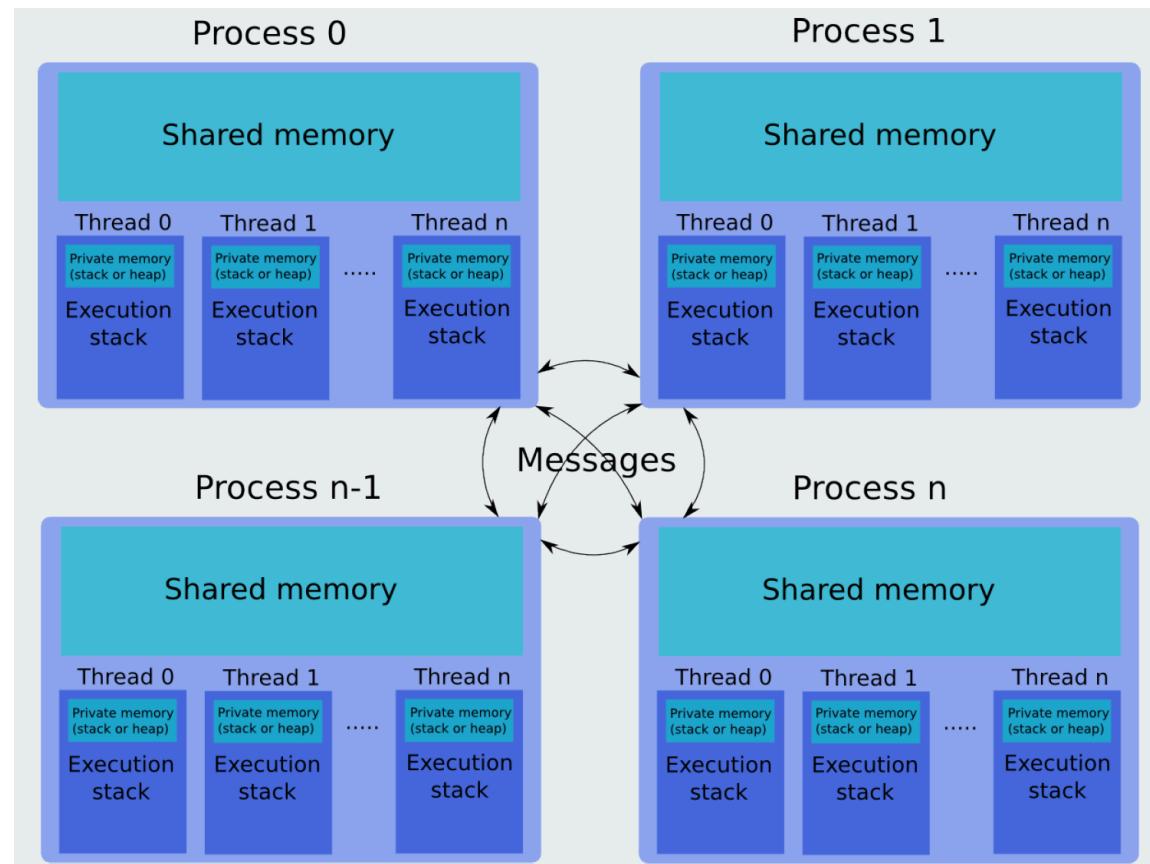


Process-Affinity  
Memory-Allocation



# Paralelismo híbrido

Consiste en la mezcla de paradigmas en paralelismo para explotar las ventajas de cada una de ellas. Es decir, **MPI** se usará para **comunicación entre procesos**, y **OpenMP** dentro de **cada proceso**



# Paralelismo híbrido

## Ventajas:

- Mayor escalabilidad al reducir a la vez el número de mensajes MPI y el número de procesos envueltos en comunicación colectiva
- Mejor balance de cargas
- Mejor adaptable a computadoras modernas (Multithreading), mientras que solo MPI es una aproximación menos eficiente
- Optimiza el consumo de memoria (memoria compartida OpenMP), menos copia de información en procesos MPI
- Mejor adaptable a limitaciones del algoritmo
- Aumenta granularidad (ratio computacion y comunicación) y por consiguiente escalabilidad

# Paralelismo híbrido

## Desventajas:

- Mayor complejidad, y requiere mas experiencia en programación
- Rendimiento en ambos, MPI y OpenMP, debe ser buena
- Rendimiento final no está garantizado. Depende de costos adicionales de proceso

# Paralelismo híbrido

## Códigos beneficiados con ...

- Aquellos con limitada escalabilidad MPI
- Aquellos que requieren balance de cargas dinámico
- Aquellos con espacio de memoria limitada y requieren copiar grande cantidad de información durante comunicación MPI
- Aplicaciones de paralelismo masivo

## Uso de hilos (threads) en MPI

MPI provee una directiva en caso la aplicación sea multi-thread: **`MPI_Init_Thread`**

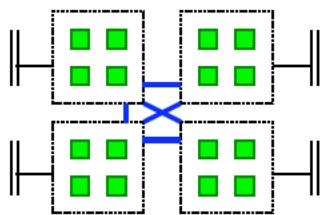
- Usualmente no se requiere de este soporte, por lo que no se activa
- No estan manipuladas en ranks, asi que los threads no se especificarán en las comunicaciones
- Cualquier thread puede hacer llamadas MPI (dependiendo del nivel de soporte)
- Cualquier thread de un proceso MPI puede recibir un mensaje enviado a este proceso
- El llamado de `MPI_Finalize` debe ser hecho por el mismo thread que llamo `MPI_Init_Thread`, y solo cuando todos los demás threads del proceso han culminado sus llamados MPI

# Modos de operación MPI-threads

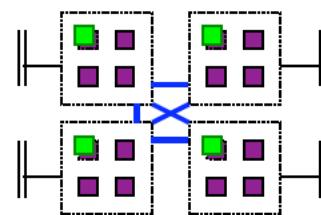
Pure MPI Node

Pure SMP Node

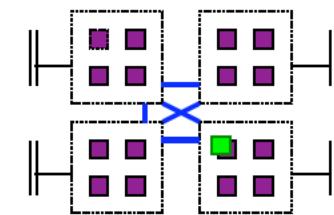
16 MPI Tasks



4 MPI Tasks  
4 Threads/Task



1 MPI Task  
16 Threads/Task



Master Thread of MPI Task

- █ MPI Task on Core
- █ Master Thread of MPI Task
- █ Worker Thread of MPI Task

## Uso en supercomputadores

### HOPPER

Current Status: Up

**Target retirement date: Dec 15, 2015**

Hopper is NERSC's first petaflop system, a Cray XE6, with a peak performance of 1.28 Petaflops/sec, 153,216 compute cores, 212 Terabytes of memory, and 2 Petabytes of disk. Hopper placed number 5 on the November 2010 Top500 Supercomputer list.



- n procesos en paralelo
- N procesos por nodo
- d cuantos threads
- OpenMP por proceso MPI

```
01. #PBS -N jacobi
02. #PBS -q debug
03. #PBS -l mppwidth=48
04. #PBS -l walltime=00:10:00
05. #PBS -e jacobijob.out
06. #PBS -j eo
07.
08. cd $PBS_O_WORKDIR
09.
10. setenv OMP_NUM_THREADS 6
11. aprun -n 8 -N 4 -S 1 -d 6 ./jacobi_mpiomp
```

## Uso en supercomputadores



- n procesos en paralelo
- N procesos por nodo
- d cuantos threads

OpenMP por proceso MPI

Status: **Up**

Edison is NERSC's newest supercomputer, a Cray XC30, with a peak performance of 2.57 petaflops/sec, 133,824 compute cores, 357 terabytes of memory, and 7.56 petabytes of disk.

Use #PBS -l mppwidth=192 for all

192 MPI: `aprun -n 192 a.out`

96 MPI + 2 OpenMP threads per MPI, 1/2 as many MPI tasks per node: `aprun -n 96 -N 12 -S 6 -d 2 a.out`

48 MPI + 4 OpenMP threads per MPI, 1/4 as many MPI tasks per node: `aprun -n 48 -N 6 -S 3 -d 4 a.out`

16 MPI + 12 OpenMP threads per MPI, 1/12 as many MPI tasks per node: `aprun -n 16 -N 2 -S 1 -d 12 a.out`

# Tecnología Hyper-threading (HT)

Multithreading implementado por Intel con el fin de mejorar procesos en paralelo en microprocesadores x86

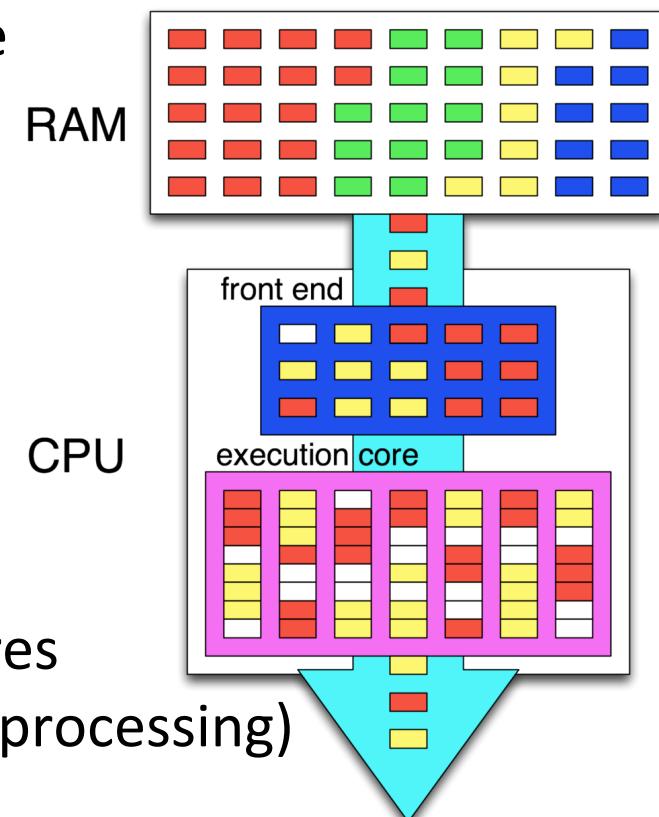
Por cada **núcleo físico** el sistema operativo direcciona **dos núcleos lógicos** y comparte las tareas de los mismos.

La principal función es de **incrementar el número de instrucciones independientes** en la pipeline (segmentación)

El tener dos tareas (hilos) por núcleo, permite **maximizar el trabajo hecho por procesador cada ciclo** (clock cycle).

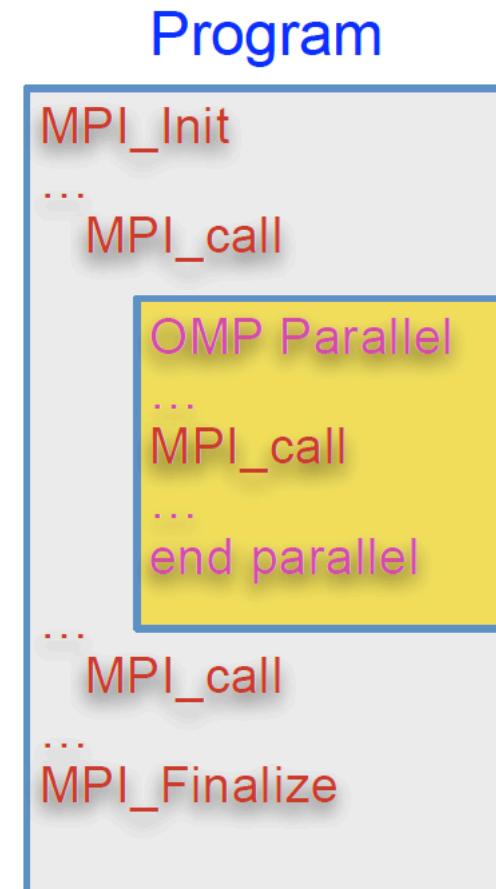
Para el SO, aparecen como dos procesadores

El SO debe soportar **SMP** (symmetric multiprocessing)



# Programa híbrido

- Se inicializa **MPI**
- Se crea regiones paralelas **OpenMP** dentro de procesos **MPI**
- Se pueden llamar procesos **MPI** en regiones en paralelo y en serie
- Se finaliza **MPI**



## OPENMP Hello

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
    int nthreads, tid;
    /* Fork a team of threads giving them their own copies of variables */
    #pragma omp parallel private(nthreads, tid)
    {
        /* Get thread number */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    } /* All threads join master thread and disband */
    exit(0);
}
```

## MPI Hello

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define MASTER      0

int main (int argc, char *argv[])
{
    int numtasks, taskid, len;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
    MPI_Get_processor_name(hostname, &len);
    printf ("Hello from task %d on %s!\n", taskid, hostname);
    if (taskid == MASTER)
        printf("MASTER: Number of MPI tasks is: %d\n",numtasks);
    MPI_Finalize();

}
```

# Código MPI-OpenMP

```
#include <mpi.h>
int main(int argc, char **argv){
    int rank, size, ierr, i;

    ierr= MPI_Init(&argc,&argv[]);
    ierr= MPI_Comm_rank (...,&rank);
    ierr= MPI_Comm_size (...,&size);
    //Setup shared mem, compute & Comm

    #pragma omp parallel for
    for(i=0; i<n; i++){
        <work>
    }
    // compute & communicate

    ierr= MPI_Finalize();
```

# Código MPI-OpenMP:

‘Funneling’ por el master. Barrier es necesario

```
#include <mpi.h>
int main(int argc, char **argv){
    int rank, size, ierr, i;

    #pragma omp parallel
    {
        #pragma omp barrier
        #pragma omp master
        {
            ierr=MPI_{Whatever}(...)
        }
        #pragma omp barrier
    }
}
```

## **MPI\_Init\_thread:**

inicializa MPI con soporte de hilos. Cualquier hilo puede llamar procesos MPI

```
int MPI_Init_thread(int *argc, char ***argv, int required, int *provided )
```

**required** es el nivel de soporte requerido

**provided** es el nivel de soporte obtenido

**MPI\_THREAD\_SINGLE**: solo ejecuta un thread (no OpenMP)

**MPI\_THREAD\_FUNNELED**: de todos los threads del proceso, solo el que llamo **MPI\_INIT\_THREAD** puede llamar MPI

**MPI\_THREAD\_SERIALIZED**: todos los threads pueden llamar MPI, pero solo uno a la vez (critical)

**MPI\_THREAD\_MULTIPLE**: todos los threads pueden llamar MPI sin restricciones

# Código MPI-OpenMP:

Llamada MPI en  
omp single

mpi\_init\_thread  
inicializa soporte  
de threads en  
MPI

```
#include <mpi.h>
int main(int argc, char **argv){
    int rank, size, ierr, i;
    mpi_init_thread(MPI_THREAD_SERIALIZED,
iprovided)
    #pragma omp parallel
    {
        #pragma omp barrier
        #pragma omp single
        {
            ierr=mpi_<Whatever>(...)
        }
        //pragma omp barrier
    }
}
```

## Máquinas híbridas:

e.g. Una máquina con 268 nodos

Cada nodo tiene 8 núcleos

Se pueden combinar:

- 1 proceso MPI y 8 hilos OpenMP
- 2 procesos MPI y 4 hilos OpenMP
- 4 procesos MPI y 2 hilos OpenMP

## 2 nodos, 1 proceso MPI/nodo, 4 hilos

0 running on compute-2-25.local thread= 0 of 4

0 running on compute-2-25.local thread= 1 of 4

0 running on compute-2-25.local thread= 2 of 4

0 running on compute-2-25.local thread= 3 of 4

1 running on compute-3-14.local thread= 0 of 4

1 running on compute-3-14.local thread= 1 of 4

1 running on compute-3-14.local thread= 2 of 4

1 running on compute-3-14.local thread= 3 of 4

## Ejercicio 1:

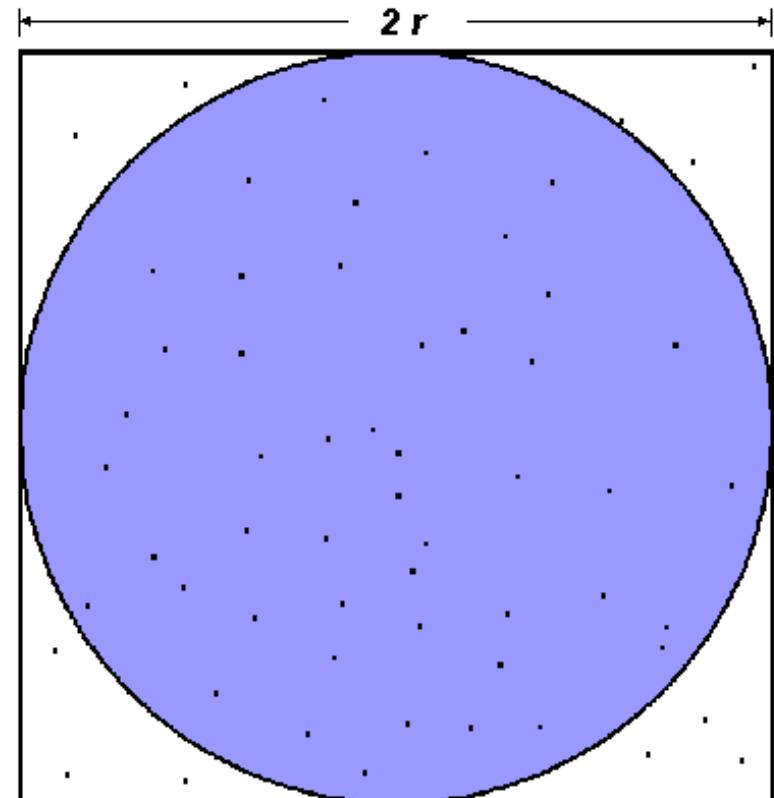
Paralelizar el método de Monte Carlo para calcular PI en MPI y OPENMP

- Generar N puntos random en el cuadrado exterior
- Determinar el número de puntos n dentro del círculo
- Siendo el área del círculo  $\pi r^2$ , y el área del cuadrado  $4r^2$
- Entonces  $n/N = \pi/4$ , o  $\pi=4n/N$
- Generando un número suficiente de puntos se logra un resultado mas exacto

Utilize [reduction](#) en OPENMP y [AllReduce](#) en MPI

Medir tiempos de ejecución y comparar la eficiencia utilizando un procesador con 4 threads y el mismo con 4 procesos MPI.

Incrementar para ello el número de iteraciones tanto como sea necesario para que la medición del tiempo sea sensible al proceso.



$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

**Ejercicio 2.** Parallelizar la multiplicación de dos matrices en OPENMP y MPI  
Medir y comparar tiempos de ejecución

```
for(i=0;i<NA;i++)  
    for(j=0;j<NB;j++)  
        for(k=0;k<NC;k++)  
            C[i][j] += A[i][k] * B[k][j];
```

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

$$\text{Task 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

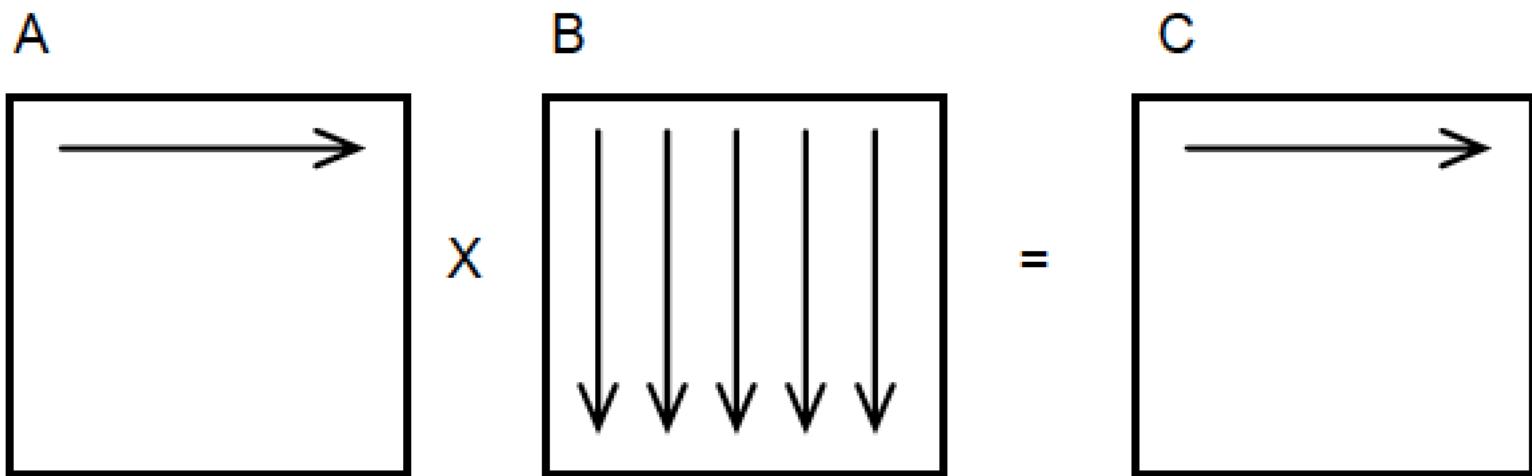
$$\text{Task 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$\text{Task 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$\text{Task 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

(b)

## Método de comunicación P2P (MPI):



Algoritmo ejecuta líneas de la matriz C en forma secuencial

Cada iteración, una línea de A y todas las columnas de B son procesadas

Complejidad del problema:  $O(ijk)$ , donde las matrices son  $i \times j$

La operación básica es calcular un elemento de C

$$c_{ij} = (a_i, b_j^T), \quad a_i = (a_{i0}, a_{i1}, \dots, a_{in-1}), \quad b_j^T = (b_{0j}, b_{1j}, \dots, b_{n-1j})^T$$

En general, el número de procesos es  $p < n^2$  d, por lo que sera necesario realizar sub-operaciones que alberguen una fracción de la matriz

## Método de comunicación P2P (MPI):

```
if (rank == 0) {
    // initialize matrices
    // Send matrix data to the worker tasks
    for (dest=1; dest<=np-1; dest++)
    {
        .....
        rows = .....
        MPI_Send(&a[offset][0], rows*NA, MPI_DOUBLE, dest, mtype,MPI_COMM_WORLD);
    }
    for (i=1; i<np; i++)
    {
        .....
        MPI_Recv(&c[offset][0], rows*NB, MPI_DOUBLE, i, mtype,MPI_COMM_WORLD, &status);
    }
}
if (rank > 0) {
    MPI_Recv(&a, rows*NA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD,&status);
    // calculate sub-matrix
    MPI_Send(&c, rows*NB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
.....
}
```

## Método de asignación de tareas (OPENMP):

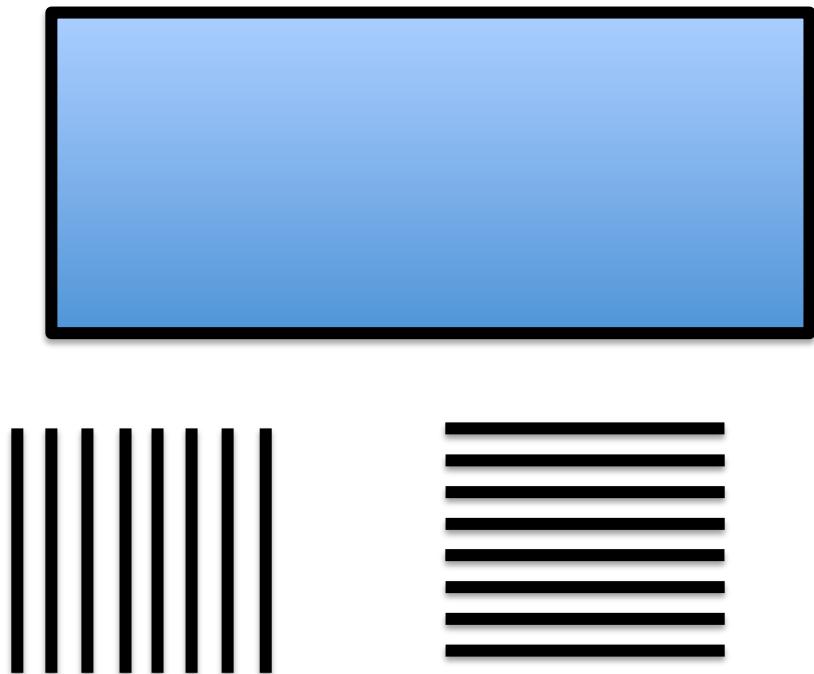
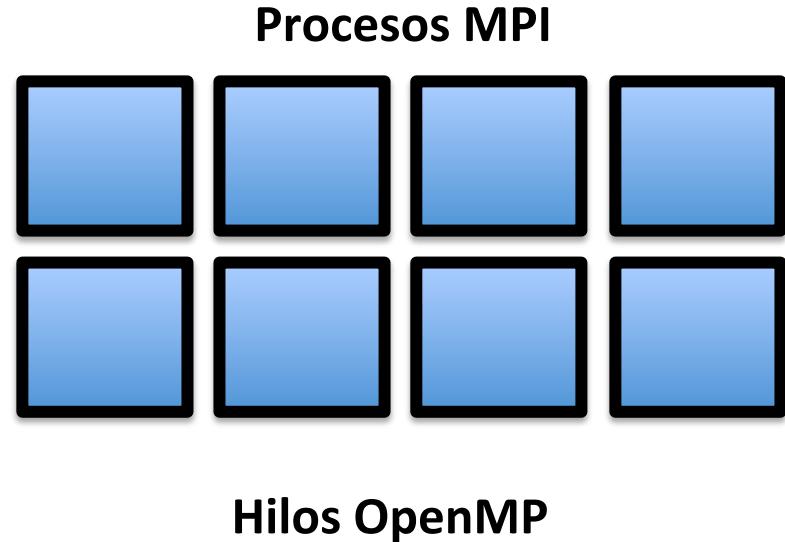
Multiplicación de matrices ( $N \times N$ ) utilizando bloques de tareas en bloques ( $M \times M$ )

La declaración OPENMP podría ser:

```
#pragma omp task private(ii, jj, kk) depend ( in: A[i:M][k:M], B[k:M][j:M] )
depend ( inout: C[i:M][j:M] )
```

## Optimización en el uso de memoria:

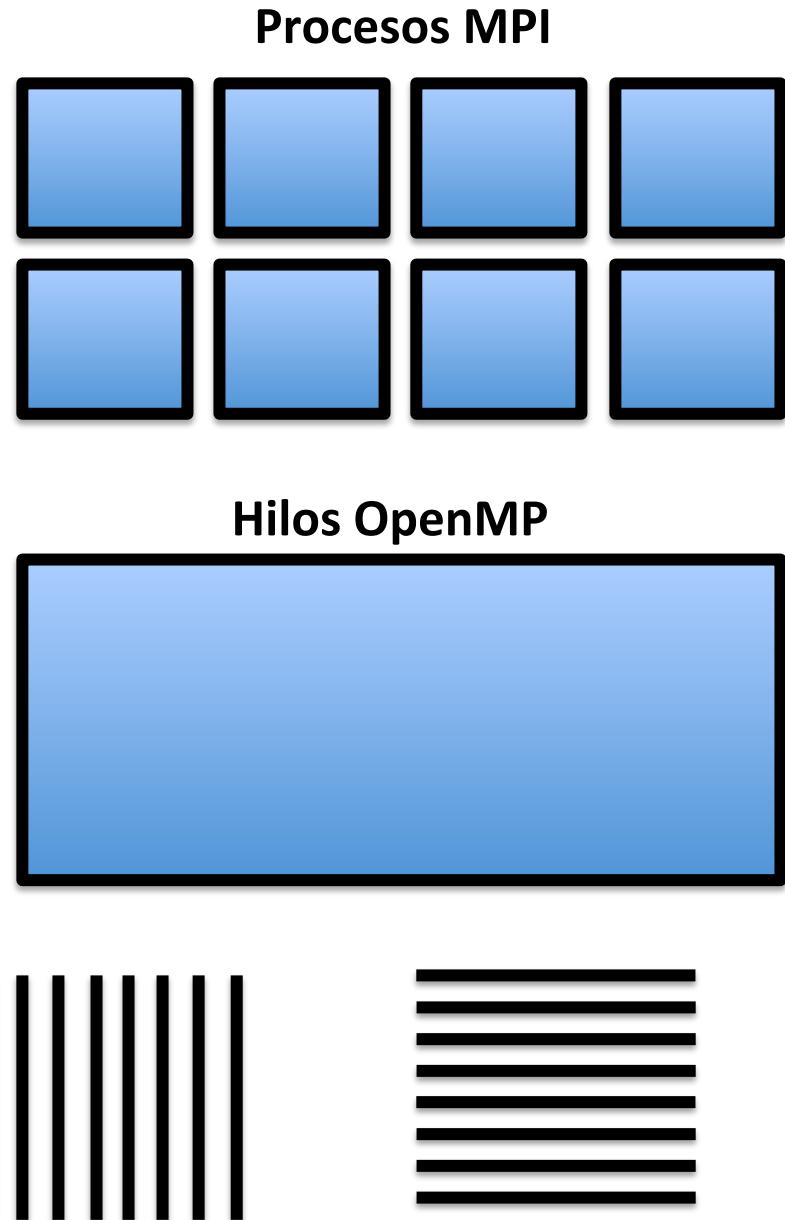
- **Regiones de halo** son copias locales de información remota que es necesaria para procesamiento de datos. Deben ser copiadas frecuentemente
- Utilizando OpenMP se **reduce el tamaño de las copias de información** de los halos que deben ser almacenadas



Ahorro de memoria de almacenamiento

## Optimización en el uso de memoria:

- Dependiendo de la estructura de datos, se puede aprovechar de la combinación **MPI/OpenMP** para reducir requerimiento de memoria
- Reduciendo regiones de halo también **reduce requerimientos de comunicación**



Ahorro de memoria de almacenamiento

## Optimización en el uso de memoria:

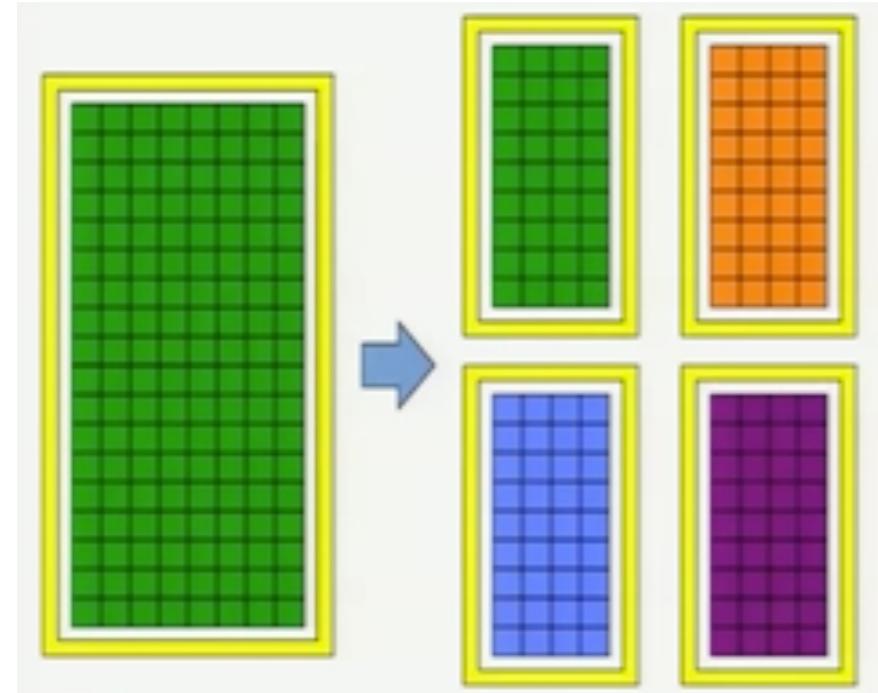
- **Programación híbrida** permite optimizar el código a la arquitectura de hardware (normalmente compuesta de nodos de memoria compartida, interconectados). La ventaja de tener memoria compartida en un nodo es que no necesita duplicar información para compartirla. Cada thread accede (escribe o lee) shared data
- Las **áreas de halo (*ghosts*)**, que aparecen en programación MPI con descomposición de dominios, no se necesitan en nodos OpenMP, solo las celdas fantasma asociadas con la comunicación entre nodos son necesarias
- El **ahorro de memoria** debido a la eliminación de celdas fantasma dentro del nodo puede ser considerable. Este depende del orden del método usado, del tipo de dominio (2D, 3D), de la descomposición de dominio (uni- o multi-dimensional), y en el número de núcleos del nodo.
- El **buffer crece** con el número de procesos en MPI.

## Descomposición de dominio

Malla en dos dimensiones:  
Incrementando el número de procesos a 4 conlleva a que cada proceso tenga:

- Un cuarto del número de celdas que computar
- La mitad del número de celdas en halos para comunicar.

Las partes secuenciales del código no se ven modificadas

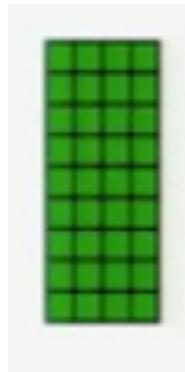


P procesos,  
cada uno con  
 $M \times N$  celdas y  
 $2M+2N$  celdas  
en halo

4P procesos,  
cada uno con  
 $(M/2) \times (N/2)$   
celdas y  $(M+N)$   
celdas en halo

## Descomposición de dominio

Computación:



orden  $O(P)$  para  $P$  procesos  
(problema de escalamiento menor)

Comunicación:



orden  $O(\sqrt{P})$  para  $P$  procesos  
la región de halo disminuye más  
lentamente al incrementar el  
número de procesos (problema  
de escalamiento mayor)

Por ejemplo, un cubo de  $8 \times 8 \times 8$  elementos con 1 elemento de halo,  
se transformaría en un cubo de  **$10 \times 10 \times 10$  (50% halo)**, o  $16 \times 16 \times 16$   
en un cubo de  $18 \times 18 \times 18$  (**30% halo**), o  $32 \times 32 \times 32$  en un cubo de  
 **$34 \times 34 \times 34$  (17% halo)**

## Ejemplo simple de código híbrido OpenMP+MPI:

```
export SET_NUM_THREADS=4  
mpicc -fopenmp hello.c  
mpiexec -np 2 ./a.out
```

En ocasiones, el número óptimo de threads es una característica de la arquitectura de hardware, y debe ser sujeto a tests.

```
#include <stdio.h>  
#include "mpi.h"  
#include <omp.h>  
int main(int argc, char *argv[]) {  
    int numprocs, rank, namelen;  
    char processor_name[MPI_MAX_PROCESSOR_NAME];  
    int iam = 0, np = 1;  
  
    MPI_Init(&argc, &argv);  
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    MPI_Get_processor_name(processor_name, &namelen);  
    #pragma omp parallel default(shared) private(iam, np)  
    {  
        np = omp_get_num_threads();  
        iam = omp_get_thread_num();  
        printf("Hello from thread %d out of %d  
from process %d out of %d on %s\n",  
               iam, np, rank, numprocs, processor_name);  
    }  
    MPI_Finalize();  
}
```

## Ecuación de Poisson en 3D

Este método resuelve la ecuación de Poisson en un dominio cúbico  $[0,1] \times [0,1] \times [0,1]$  usando una discretización finita y el método de Jacobi

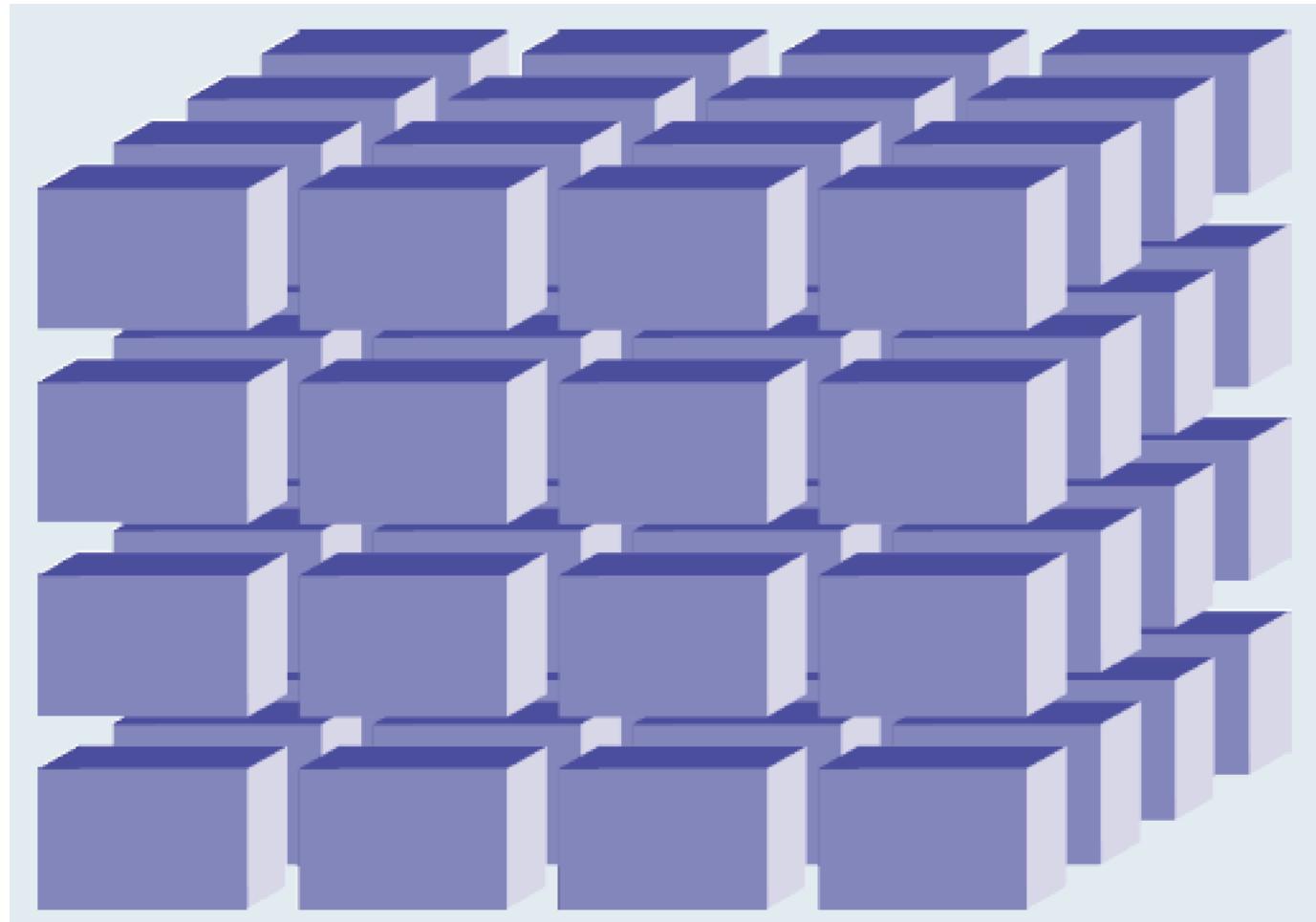
$$\begin{cases} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = f(x, y, z) & \text{in } [0, 1] \times [0, 1] \times [0, 1] \\ u(x, y, z) = 0. & \text{on the boundaries} \\ f(x, y, z) = 2yz(y - 1)(z - 1) + 2xz(x - 1)(z - 1) + 2xy(x - 1)(y - 1) \\ u_{\text{exact}}(x, y) = xyz(x - 1)(y - 1)(z - 1) \end{cases}$$

Se discretiza en una malla regular en las tres direcciones del espacio ( $h=h_x=h_y=h_z$ )  
La solución se calcula usando el método de Jacobi donde la solución de la iteración  $n+1$  se calcula usando la inmediata adyacente iteración  $n$

$$u_{ijk}^{n+1} = \frac{1}{6}(u_{i+1jk}^n + u_{i-1jk}^n + u_{j+1ik}^n + u_{j-1ik}^n + u_{jk+1}^n + u_{jk-1}^n - h^2 f_{ijk})$$

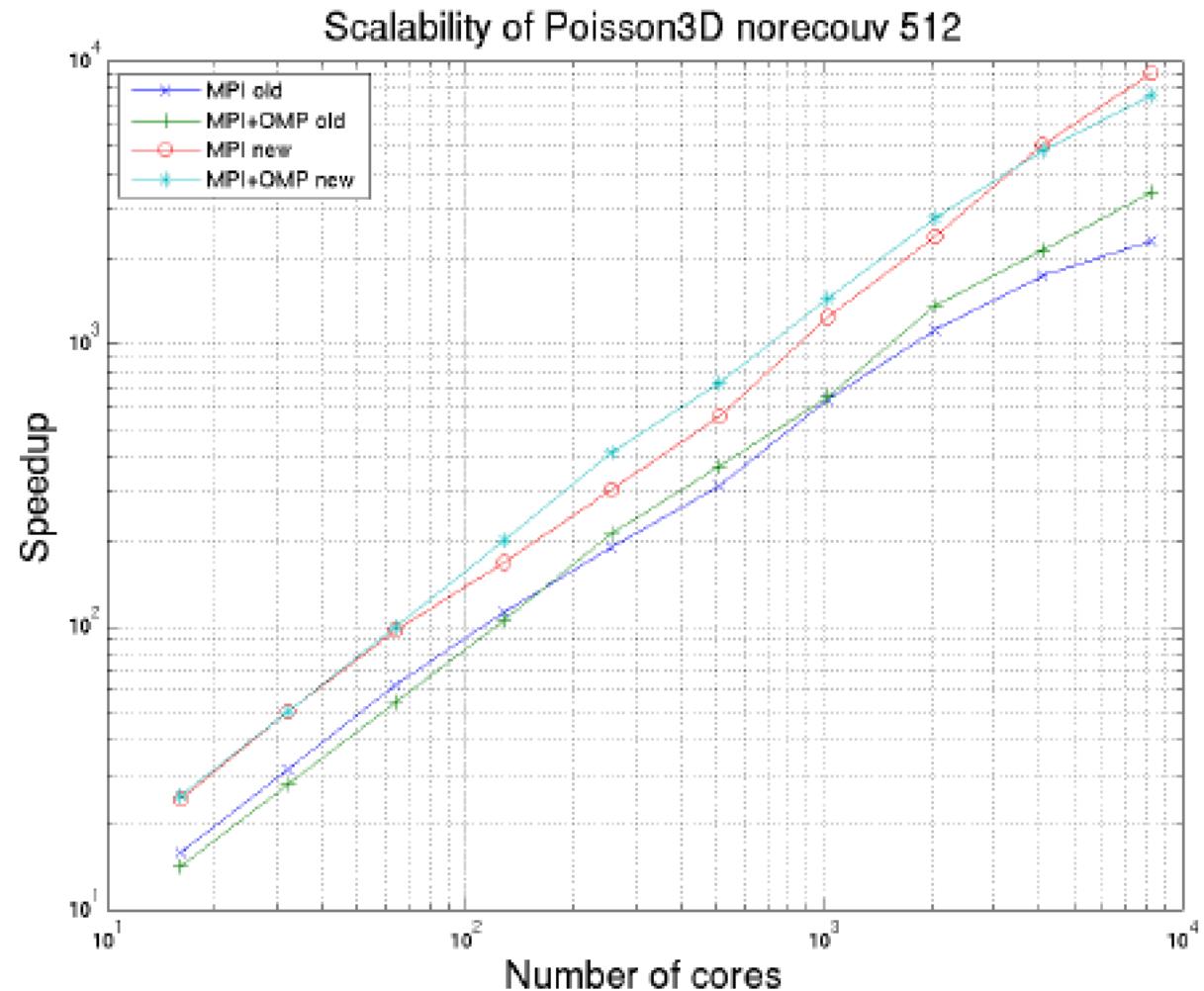
## Ecuacion de Poisson en 3D

El dominio estará separado en 3 direcciones espaciales



## Ecuacion de Poisson en 3D

Tests realizados en un supercomputador IBM Blue Gene, de 10240 nodos con 4 núcleos cada uno



## OpenMP + MPI

Por qué no necesariamente mas eficiente?

- OpenMP tiene mayor escalabilidad debido a paralelismo implícito
- Durante la comunicación MPI todos los threads esperan (idle) menos uno
- Costo excesivo en casos cuando se crean threads
- Tener en cuenta la coherencia de la memoria cache en métodos distintos, y el lugar donde se almacena información
- OpenMP puro es en muchos casos menos eficiente que MPI puro en un nodo
- No hay librerías/compiladores optimizados en OpenMP

Por ello, para encontrar el método mas eficiente hay necesidad de aplicar los métodos y comprobar eficiencias antes de empezar a producir resultados a mayor escala.

## Ejercicio 3:

- Utilizar el método de cuadratura para calcular el valor de pi, en un código híbrido OpenMP+MPI

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

La variable de integración x sera igual a  $(i-0.5)/N$ ,

Donde i es el número de intervalo y N el número total de intervalos

Implementar el programa con directivas de pre-procesamiento para activar/desactivar compilación con OpenMP/MPI/serial

Ejecutar el programa en paralelo en 2,4,8,16 procesadores, y 2,4,6 threads, usando  $N=10^4, 10^5, 10^6$  y comparar resultados con el método en serie

**Sugerencia:** aplique OpenMP en el ciclo for del cálculo de la suma, y MPI fuera del ciclo para primero distribuir variables a los nodos, y finalmente recolectar el valor de pi.

Comente que método es más eficiente para paralelizar este problema

# Algoritmos Paralelos

---