



# Computación Paralela y Distribuida

2022-II

---

José Fiestas

7 de octubre de 2022

Universidad de Ingeniería y Tecnología  
jfiestas@utec.edu.pe

# Unidad 4: Comunicación y coordinación

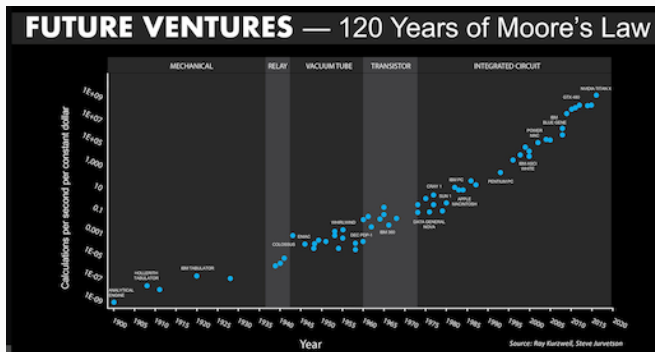
## Objetivos:

1. Pasos de Mensaje: MPI, Mensajes Punto a Punto, MPI, Comunicación Colectiva, Blocking vs non-blocking
2. Memoria Compartida: OMP, Constructores y cláusulas
3. Programacion Híbrida

## MPI+OpenMP

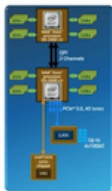
# Ley de Moore

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



<b>Haswell CPU</b>	From 4 to 18 cores TDP: 55 W to 145 W (SVR); 160 W (WS)
<b>Socket</b>	Socket-R3
<b>Scalability</b>	2S capability 4-QDR4 channels 1333, 1600, 1866 (2 DPC); 2133 (1 DPC)
<b>Memory</b>	RDIMM, LRDIMM
<b>QPI</b>	2xQPI 1.1 channels 6.4, 8.0, 9.6 GT/s
<b>PCIe*</b>	PCIe 3.0 (2, 5, 8, 16 GT/s) PCIe Extensions: Dual Cast, Atomics 40xPCIe 3.0
<b>Wellburg PCH</b>	DM2 - 4 lanes; Up to 6xUSB3, 8x USB2 ports, 10xSATA3 ports; GbE MAC (+ External PHY)
<b>LAN</b>	Fortville (40GbE), Springville (1GbE)
<b>Firmware</b>	Servers: Intel® Server Platform Services (SPS) Workstations: ME 9.x

EP = Efficient Performance  
Excellent balance of performance and power efficiency



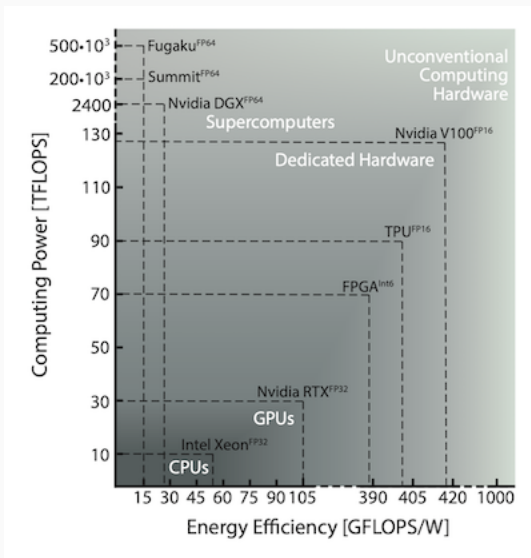
Sin embargo, problemas debido a que:

1. Poder de procesadores crece mas rápido que mejoras en acceso a memoria. Latencias de memoria decrecen lentamente. I.e. la distancia entre velocidad de memoria y la eficiencia teórica de los núcleos, crece
2. Número de núcleos por módulo de memoria crece

Por ello, procesadores pierden tiempo (ciclos) mientras esperan la información

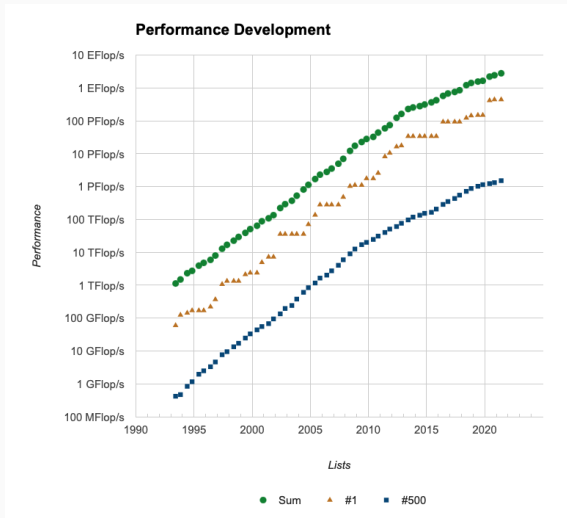
# Solución: paralelismo

## Paralelismo en memoria distribuída



# Solución: paralelismo

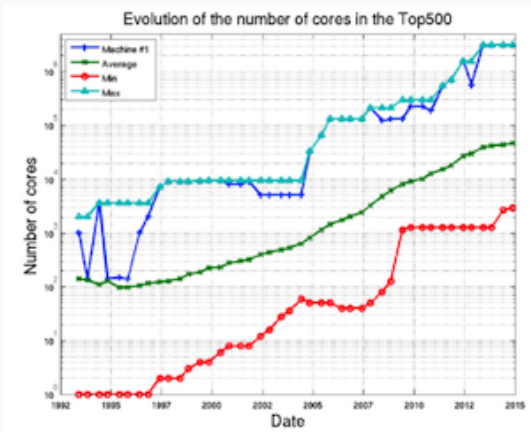
El poder computacional en Supercomputadoras se dobla cada año





# Solución: paralelismo

El número de núcleos se incrementa rápidamente (paralelismo masivo y arquitecturas multi-núcleo) Arquitecturas híbridas aparecen (GPU o XeonPhi con standard CPUs)



Por consecuencia:

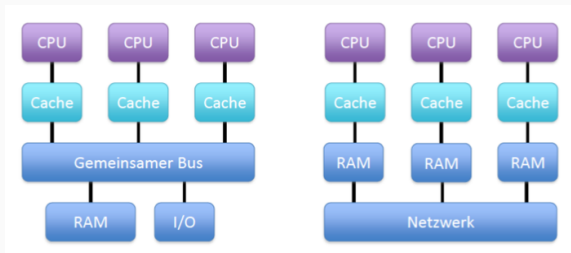
1. Es necesario explotar un número grande de núcleos “lentos”
2. La memoria individual de procesadores no aumenta, por ello no hay que desperdiciarla (paralelismo híbrido)
3. Se necesita niveles altos de paralelismo para arquitecturas modernas

Consecuencias para desarrolladores de software:

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

# Métodos de programación:

1. MPI aún predomina y se mantendrá por buen tiempo (tanto en desarrollo como en aplicaciones)
2. El híbrido MPI-OPENMP-GPU está siendo mas usado, sobre todo en supercomputadores
3. Nuevos lenguajes de programación paralela (UPC, Coarray-Fortran, PGAS, X10, Chapel, ...) aunque en etapa experimental.



**Figura 1:** MPI vs OpenMP

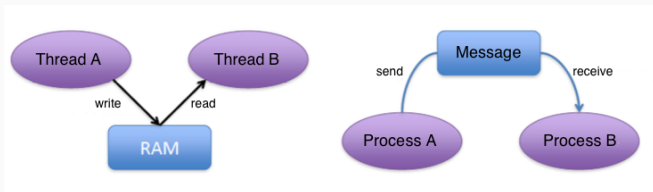
# Métodos de programación:

## Memoria compartida

- a) Hilos escriben y leen del RAM
- b) Se sincroniza de acuerdo a competencia entre tareas
- c) hilos pertenecen a un proceso

## Memoria compartida

- a) Procesos envían y reciben mensajes
- b) Solo se utilizan variables locales
- c) poseen uno o más hilos



**Figura 2:** Comunicación

# 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

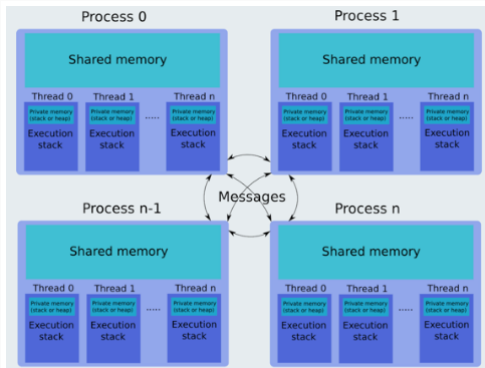


Figura 3: Comunicación

## Ventajas:

1. Mayor escalabilidad al reducir a la vez el número de mensajes MPI y el número de procesos envueltos en comunicación colectiva. Sobre todo cuando MPI solo ya no escala. Se busca explotar más niveles de paralelismo.
2. Mejor balance de cargas. Más fácil entre hilos (balance de tareas y no de data) que entre procesos (balance de tareas y data)
3. Optimiza el consumo de memoria (OpenMP), menos copia de información en procesos MPI
4. Aumenta granularidad (ratio computacion y comunicación) y por consiguiente escalabilidad. Granularidad gruesa (cada proceso hace más trabajo y disminuye la comunicación)

## Desventajas:

1. Mayor complejidad, y requiere mas experiencia en programación
2. Rendimiento en ambos, MPI y OpenMP, debe ser buena
3. Rendimiento final no está garantizado. Depende de costos adicionales de proceso
4. OMP agrega overheads a MPI (sincronización, regiones secuenciales). Sincronización exige más barreras.
5. Efectos NUMA. Se espera que procesos MPI ocupen distintos sockets (RAM) e hilos OMP pertenezcan al mismo socket de su proceso MPI

## Códigos beneficiados:

1. Aquellos con limitada escalabilidad MPI
2. Aquellos que requieren balance de cargas dinámico
3. Aquellos con espacio de memoria limitada y requieren copiar gran cantidad de información durante comunicación MPI
4. Aplicaciones de paralelismo masivo



# Optimización en el uso de memoria

**Regiones de halo** son copias locales de información remota que es necesaria para procesamiento de datos.

Deben ser

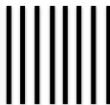
copiadas frecuentemente. Utilizando OpenMP se reduce el tamaño de las copias de información de los halos que deben ser almacenadas.

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

**Procesos MPI**



**Hilos OpenMP**



# Descomposición de dominio

Malla en dos

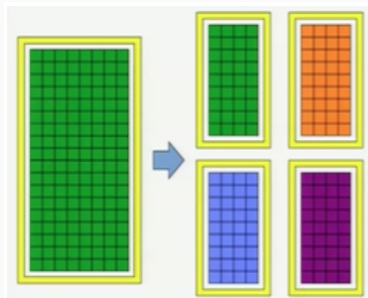
dimensiones: Incrementando el número de procesos a 4 conlleva a que cada proceso tenga:

- 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

orden  $O(P)$  para  $P$  procesos  
(problema de escalamiento menor)  
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, la data por proceso  
en un cubo de  $40^3$  elementos  
con 1 elemento de halo, se  
transformaría en un cubo de  $42^3$   
elementos (14 % halo), o  $20^3$  en  
un cubo de  $22^3$  (25 % halo), o  $10^3$   
en un cubo de  $12^3$  (42 % halo)

**Computación:**



**Comunicación:**



# Tecnología Hyper-threading (HT)

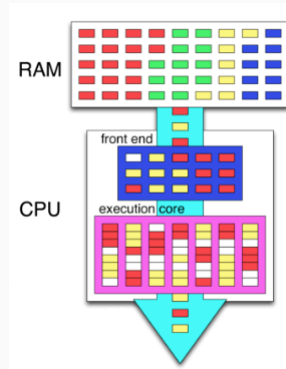
## Multithreading

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)

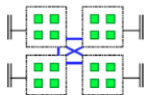


# Modos de operación MPI-threads

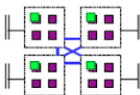
Pure MPI Node

Pure SMP Node

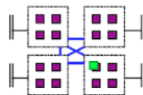
16 MPI Tasks






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

# 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

## Program

```
MPI_Init
...
MPI_call
    OMP Parallel
    ...
    MPI_call
    ...
    end parallel
...
MPI_call
...
MPI_Finalize
```

# Código híbrido OpenMP-MPI

```
1 #include <stdio.h>
2 #include "mpi.h"
3 #include <omp.h>
4 int main(int argc, char *argv[]) {
5     int numprocs, rank, namelen;
6     char processor_name[MPI_MAX_PROCESSOR_NAME];
7     int iam = 0, np = 1;
8     MPI_Init(&argc, &argv);
9     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
10    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11    MPI_Get_processor_name(processor_name, &namelen);
12    #pragma omp parallel num_threads(4) default(shared) private(iam, np)
13    {
14        np = omp_get_num_threads();
15        iam = omp_get_thread_num();
16        printf("Peekaboo desde el hilo %d de un total de %d y desde el proceso
17 %d de un total de %d en %s\n", iam, np, rank, numprocs, processor_name);
18    }
19    MPI_Finalize(); }
```

# Código híbrido OpenMP-MPI: compilación

---

```
1 mpicc -fopenmp peekaboo_mpi_omp.c
2 mpirun -np 2 ./a.out
```

---

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



# Máquinas híbridas:

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

# 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



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`

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

# 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  $n$

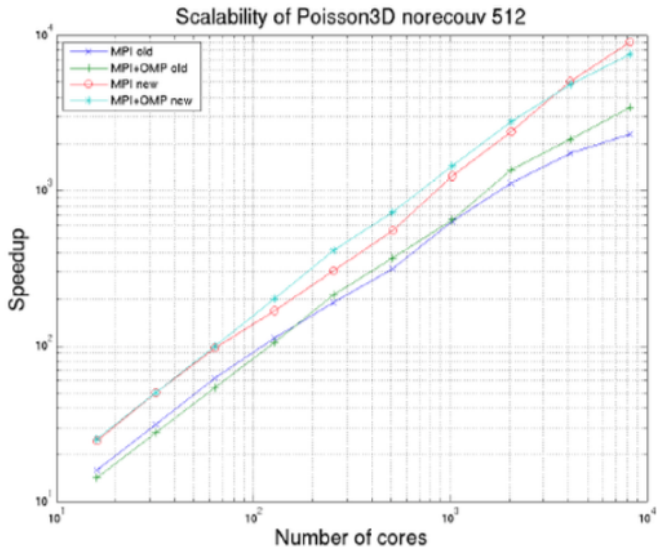
$$u_{ijk}^{n+1} = \frac{1}{6}(u_{i+1jk}^n + u_{i-1jk}^n + u_{ij+1k}^n + u_{ij-1k}^n + u_{ijk+1}^n + u_{ijk-1}^n - h^2 f_{ijk})$$

# Ecuacion de Poisson en 3D

El dominio estará separado en 3 direcciones espaciales






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

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



¿Por qué no necesariamente mas eficiente?

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

-  David B. Kirk and Wen-mei W. Hwu *Programming Massively Parallel Processors: A Hands-on Approach*. 2nd. Morgan Kaufmann, 2013. isbn: 978-0-12-415992-1.
-  Norm Matloff. *Programming on Parallel Machines*. University of California, Davis, 2014.
-  Peter S. Pacheco. *An Introduction to Parallel Programming*. 1st. Morgan Kaufmann, 2011. isbn: 978-0-12-374260- 5.
-  Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. 1st. McGraw-Hill Education Group, 2003. isbn: 0071232656.
-  Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Program- ming*. 1st. Addison-Wesley Professional, 2010. isbn: 0131387685, 9780131387683.