

# Taller de Sistemas Operativos

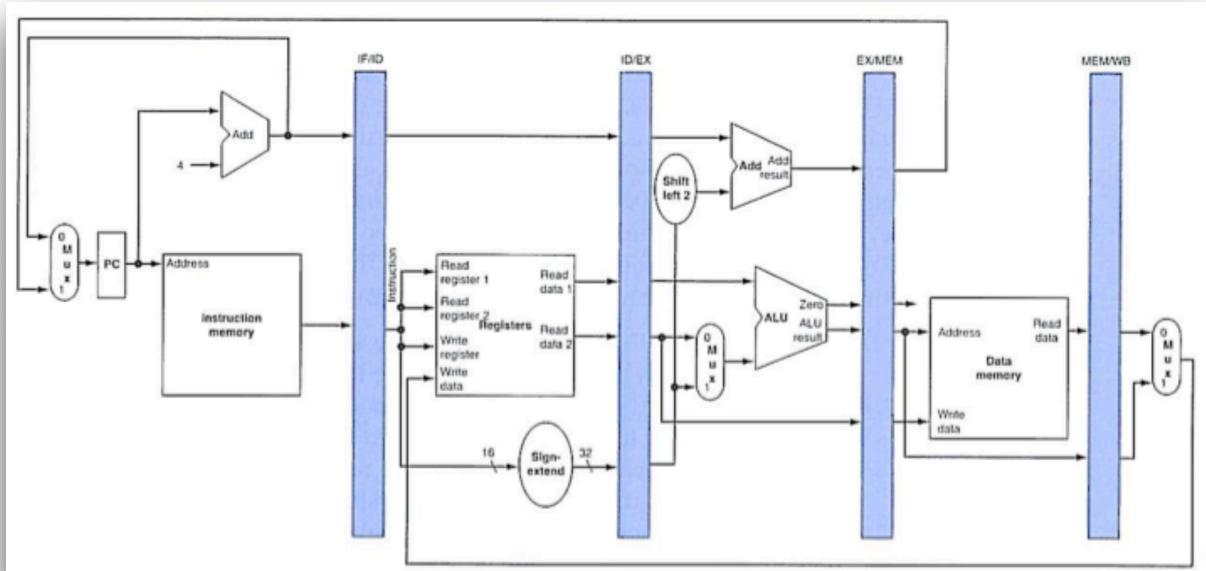
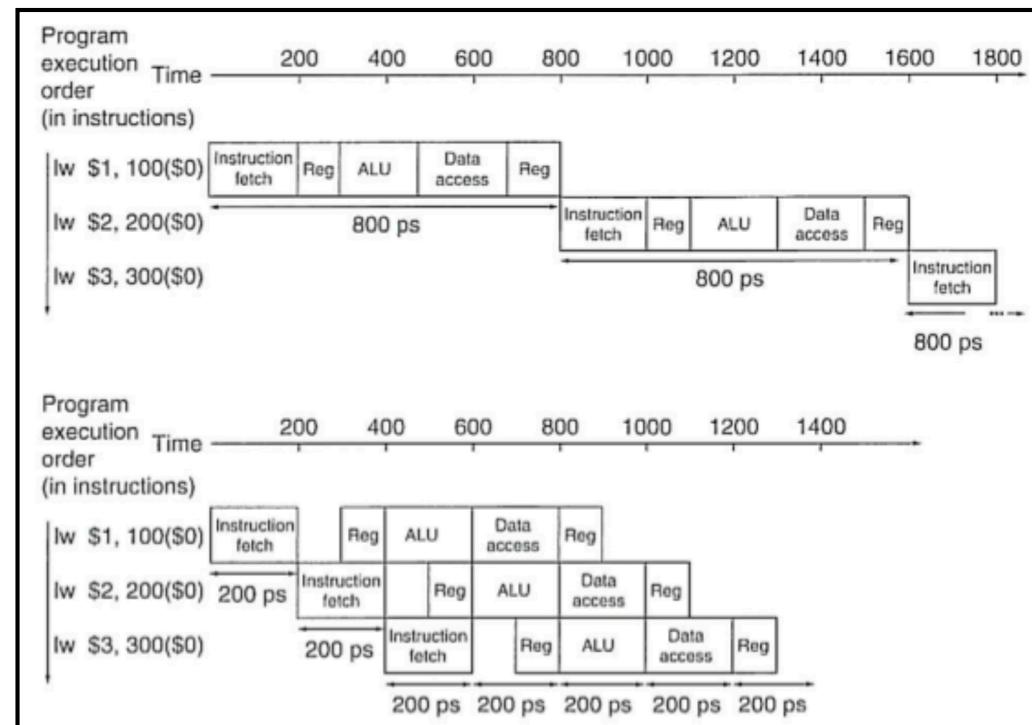
Gabriel Astudillo Muñoz

# Procesador escalar

## Modelo tradicional

Una instrucción por ciclo de reloj

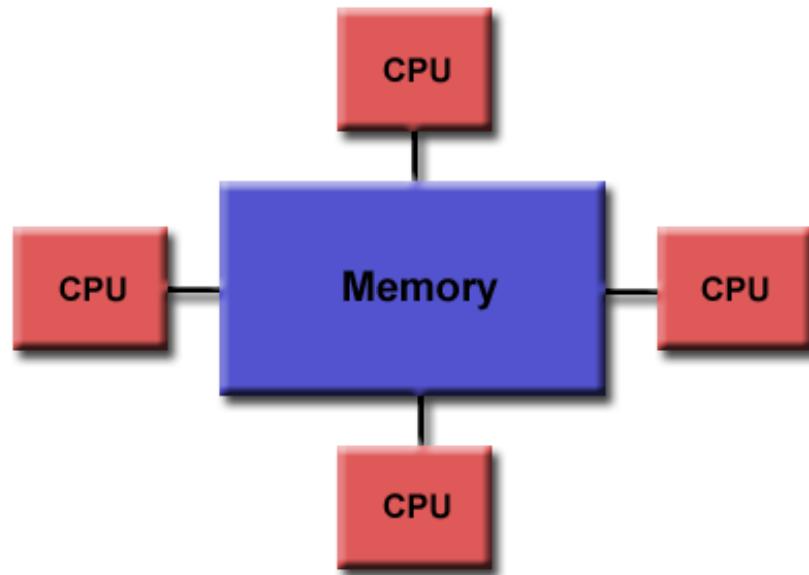
Los datos “viajan” a través de buses



Su construcción se basa en el concepto de pipelining

# Multiprocesadores de memoria compartida

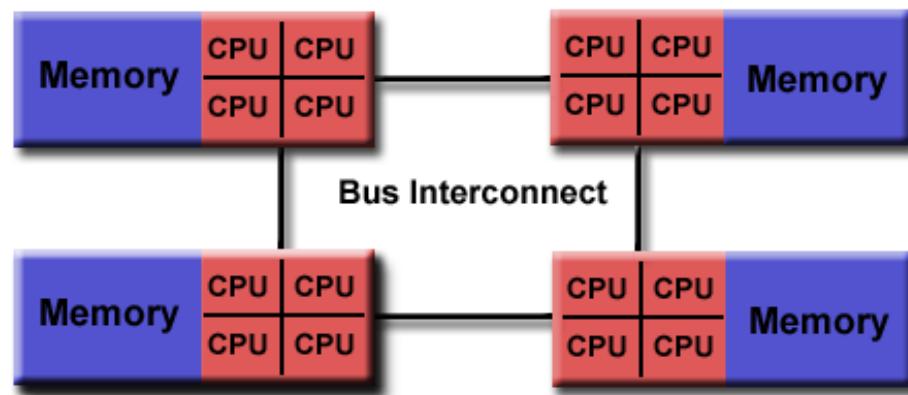
2+ procesadores que comparten el espacio de memoria



## UMA (Uniform Memory Access)

Memoria física es compartida por todos los procesadores.  
» tiempo de acceso uniforme «

## NUMA (Non Uniform Memory Access)



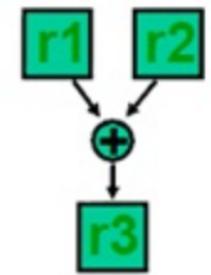
Memoria física es local a cada procesador.  
» tiempo de acceso no uniforme «

# Taxonomía de Flynn

	Una instrucción	Múltiples instrucciones
Un dato	SISD	MISD
Múltiples datos	SIMD	MIMD

Single Instruction Single Data

Modelo tradicional:  
escalar, pipelining



ADD r3, r1, r2



Single Instruction Multiple Data

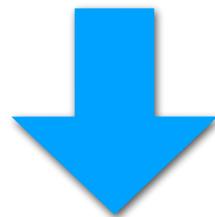
Arquitecturas vectoriales

ADDV v3, v1, v2

# Cores actuales

## Procesadores escalares

Tienen unidades dedicadas a operaciones del tipo SIMD



SSE (Intel)

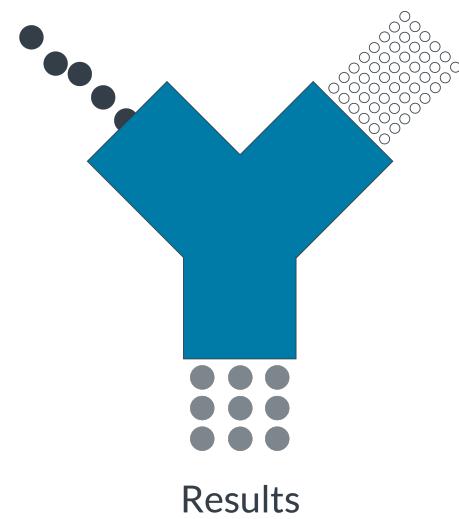
NEON (Arm)

(**S**treaming  
**S**IMD  
**E**xtensions)

SIMD Architecture

Instruction stream

Parallel data streams

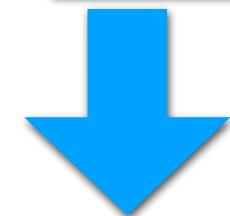


Results

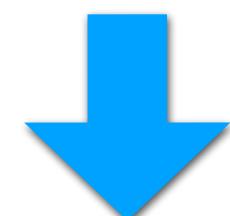
<https://developer.arm.com/technologies/neon>

# ¿Qué es programación concurrente/paralela?

Descomponer el trabajo en partes más simples



Distribución de las partes en un conjunto de procesadores



Cada parte se implementa en forma 100% secuencial

Coordinar el trabajo y la comunicación.

Consolidar los resultados parciales

No olvidarse de la infraestructura...

Seleccionar modelo de programación

# Infraestructura: Topologías de CPU

Más información en:

[https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/6/html/performance\\_tuning\\_guide/main-cpu#s-cpu-papr](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/main-cpu#s-cpu-papr)  
<http://queue.acm.org/detail.cfm?id=2513149&ref=qnh>

# Determinar Topología

```
lscpu
```

```
CPU(s): 8  
On-line CPU(s) list: 0-7  
Thread(s) per core: 1  
Core(s) per socket: 4  
Socket(s): 2
```

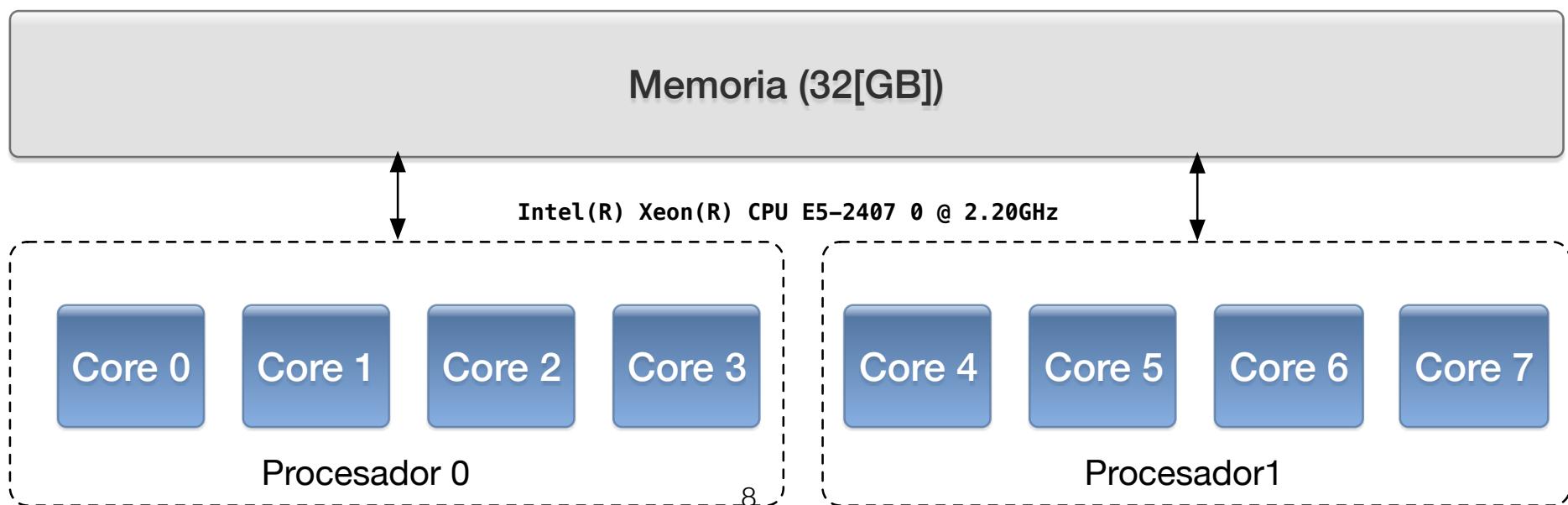


(extracto)

```
cat /proc/meminfo
```

```
MemTotal: 32899260 kB
```

## Vista General

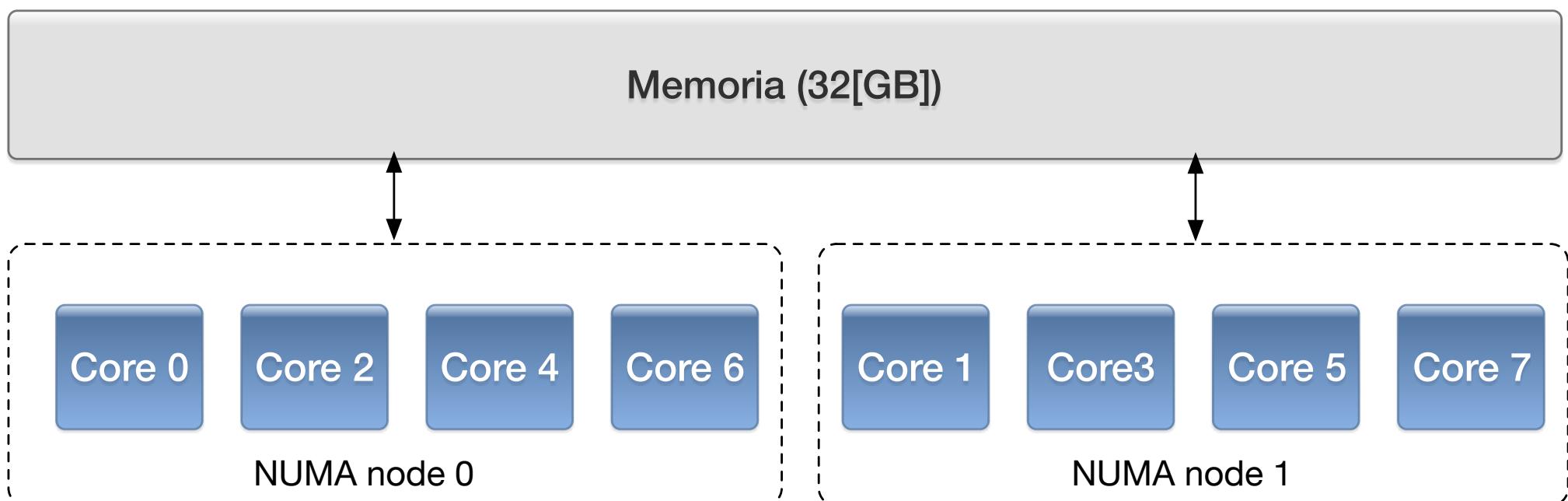
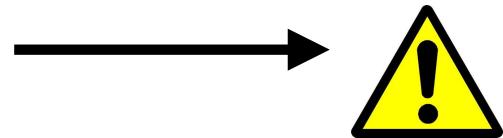


# Determinar Topología

lscpu

(extracto)

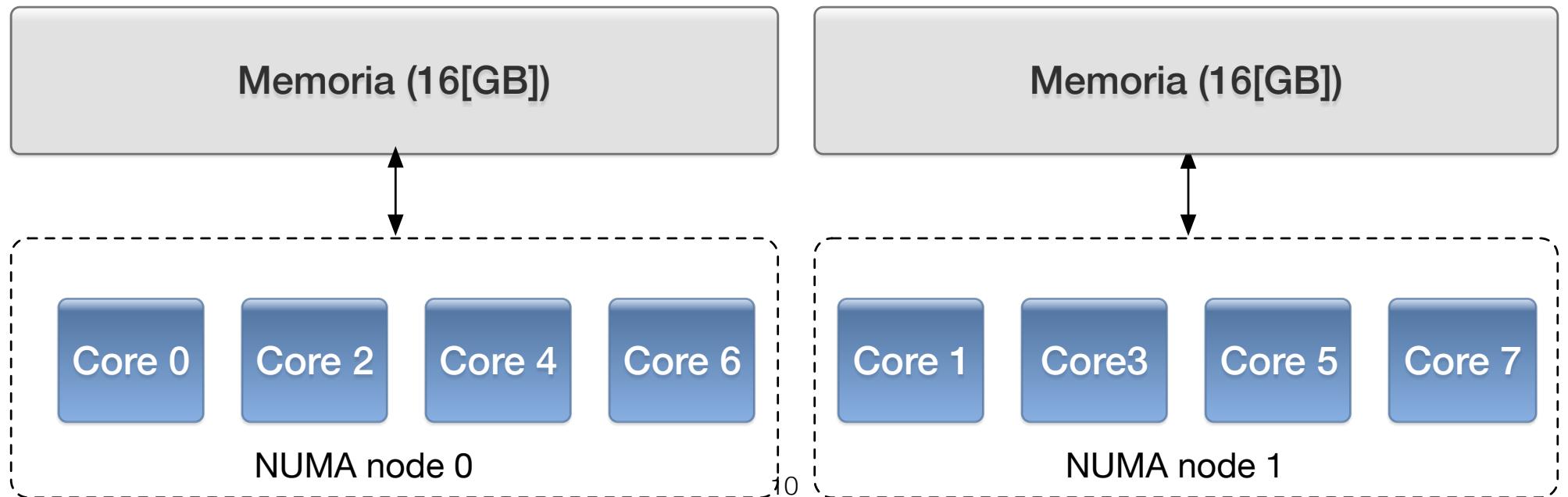
```
CPU(s):          8
On-line CPU(s) list: 0-7
Thread(s) per core: 1
Core(s) per socket: 4
Socket(s):        2
NUMA node(s):     2
NUMA node0 CPU(s): 0,2,4,6
NUMA node1 CPU(s): 1,3,5,7
```



# Determinar Topología

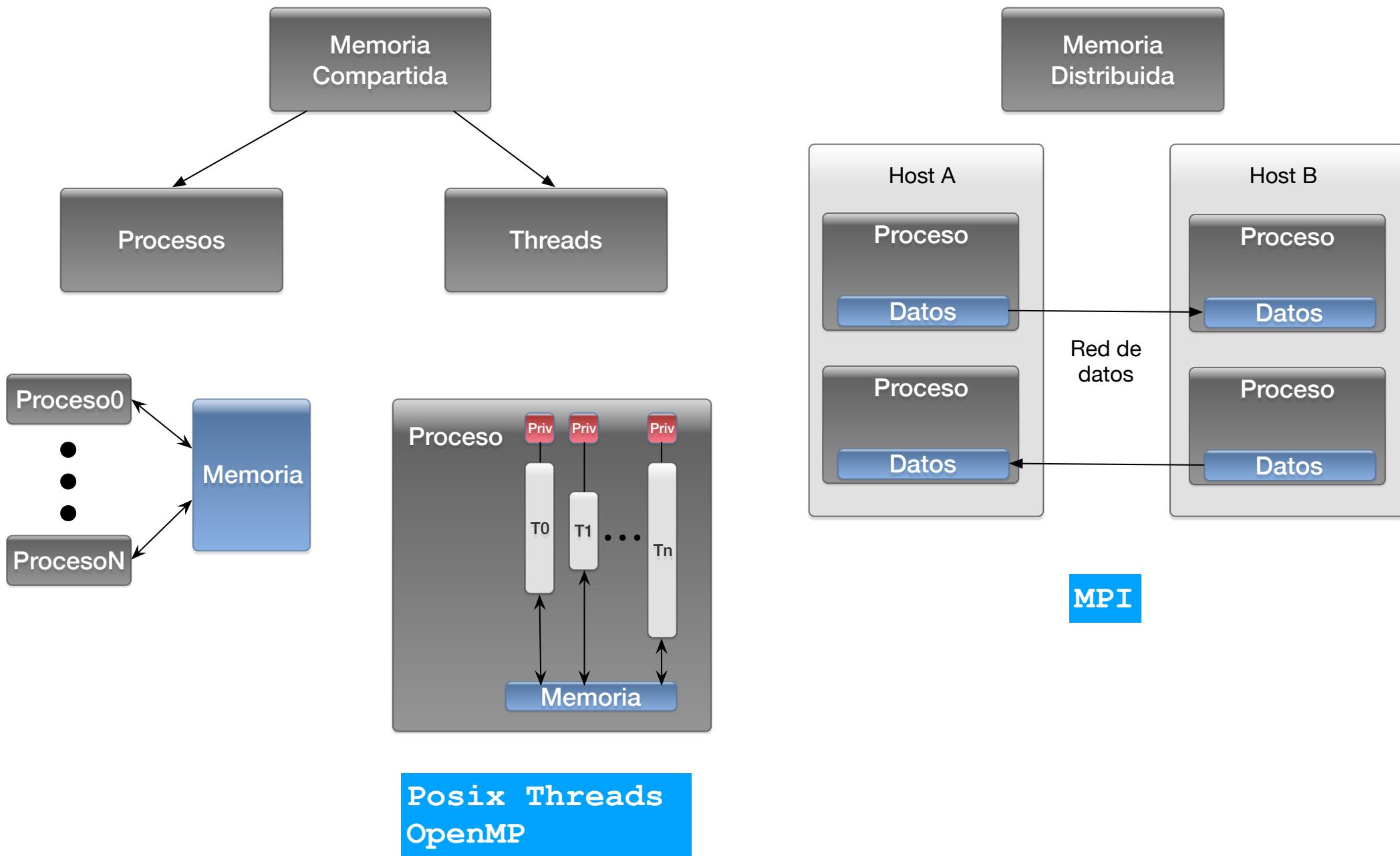
```
numactl --hardware
```

```
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6
node 0 size: 16015 MB
node 0 free: 15776 MB
node 1 cpus: 1 3 5 7
node 1 size: 16113 MB
node 1 free: 15979 MB
node distances:
node    0    1
 0:   10   20
 1:   20   10
```



# Modelos de programación

# Modelos de programación paralela



# Modelos de programación paralela

## Single Program Multiple Data (SPMD)

Single Program

Se ejecutan múltiples copias (procesos) del mismo programa

Multiple Data

Cada proceso puede utilizar diferentes datos

./ejecutable

Proceso0

./ejecutable

Proceso1

• • •

./ejecutable

ProcesoN

# **Elementos de Análisis de Rendimiento**

# Rendimiento

Costo computacional de un programa que se ejecuta en un procesador

tiempo



memoria

Interesa el **Tiempo de Ejecución**

$$T_{ejec} = \frac{I_c \cdot CPI}{f}$$

# de instrucciones de máquina

nro promedio de ciclos por instrucción

frecuencia de reloj del sistema computacional

MFLOPS

$$MFLOPS = \frac{nro \ de \ instrucciones \ en \ pto. \ flotante}{T_{ejec} \cdot 10^6}$$

# Aceleración (S)

## ABSOLUTO

$$S_{absoluto} = \frac{t_{sec}(I)}{t_p(I)}$$

tiempo para resolver I utilizando el mejor algoritmo secuencial y el procesador más veloz

tiempo para resolver I utilizando la solución paralela propuesta con **P** procesadores

## RELATIVO

$$S_{relativo} = \frac{t_{sec}(I, Q)}{t_p(I, Q)}$$

tiempo para resolver I utilizando la solución paralela propuesta Q y 1 procesador

tiempo para resolver I utilizando la solución paralela propuesta Q y **P** procesadores

# Límites a la Aceleración (S)

**$t_{sec}$**



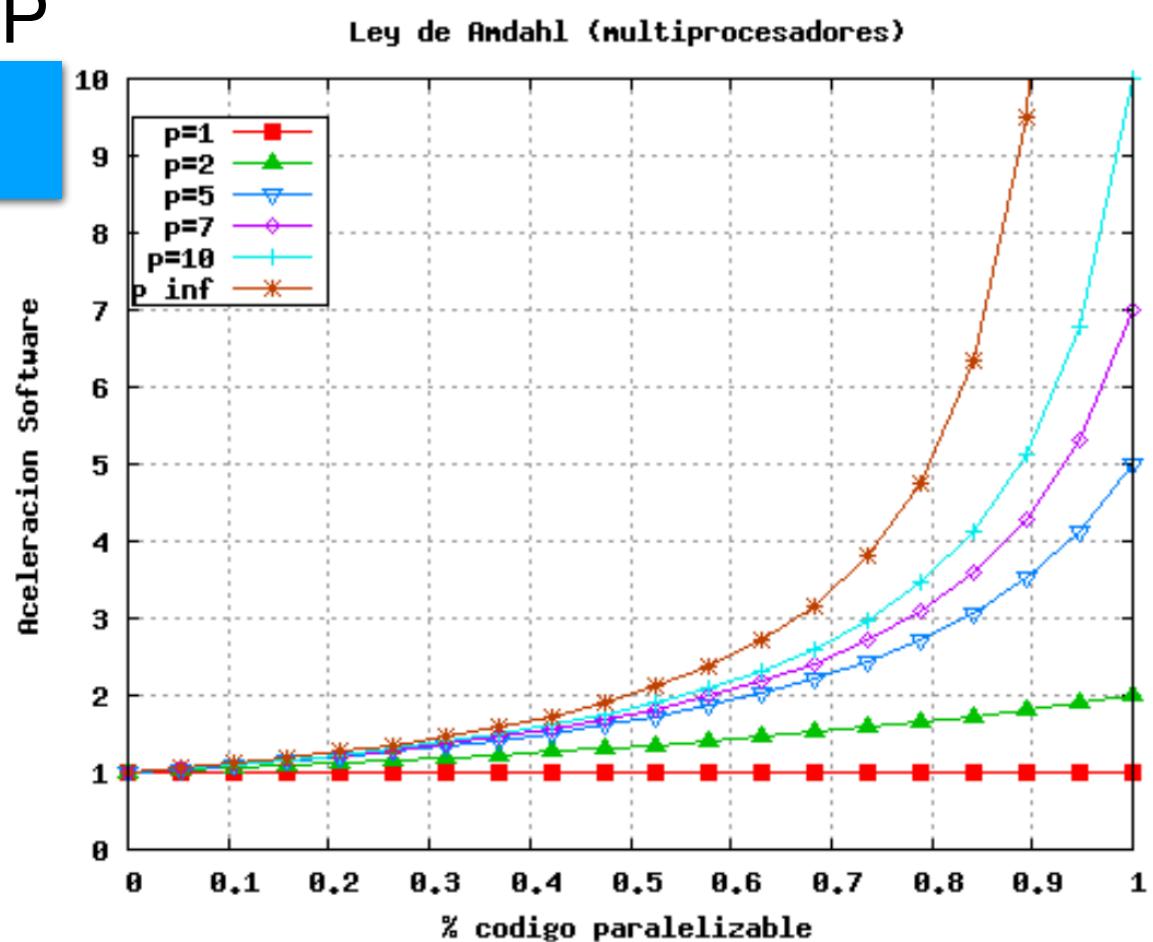
**$t_p$**



**Ley de Amdahl**

$$A = \frac{1}{(1 - f) + \frac{f}{p}}$$

f: fracción del programa que es paralelizable



# Eficiencia

Para un sistema de p  
procesadores:

$$E(p) = \frac{S(p)}{p}$$

$E(p) \rightarrow 0$  : todo el programa se ejecuta en un procesador en forma secuencial.

$E(p) \rightarrow 1$  : todo el programa se ejecuta en todos los procesadores

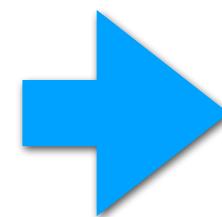
Sea:

$T_1$ : tiempo secuencial

$T_0$ : overhead del programa paralelo

$p$  : nro de procesadores

$T_{paralelo}$ : tiempo total paralelo



$$E(p) = \frac{1}{1 + \frac{T_0}{T_I}}$$

# y... ¿cómo medimos el tiempo?

comando `time`

```
$ time ./simulacion
real 0m7.103s
user 0m13.288s
sys  0m0.768s
$
```

`std::chrono`

```
#include <chrono>
auto start = std::chrono::high_resolution_clock::now();
```

BLOQUE a MEDIR

```
auto end      = std::chrono::high_resolution_clock::now();
auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start)
auto totalTime = elapsed.count();
```

# Ejemplo

## Aceleración de código

```
for(size_t i = 0; i < repeticiones; i++) {  
    the_dist += sqrt( pow(x2-x1,2) + pow(y2-y1,2) );  
}
```

Versión 1

```
for(size_t i = 0; i < repeticiones; i++) {  
    the_dist += sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) );  
}
```

Versión 2

Repeticiones: 200000000

=====Versión 1=====

Tiempo01 :9560[ms]

Distancia:1.56564e+09

=====Versión 2=====

Tiempo02 :563[ms]

Distancia:1.56564e+09

=====Versión 3=====

Tiempo03 :552[ms]

Distancia:1.56564e+09

=====Desempeño=====

Aceleración A(1,2) = 16.9805

Aceleración A(1,3) = 17.3188

```
double dx = x2-x1;  
double dy = y2-y1;
```

```
for(size_t i = 0; i < repeticiones; i++) {  
    the_dist += sqrt( dx*dx + dy*dy );  
}
```

Versión 3

# Ejemplo

## Aceleración de código

```
for(size_t i = 0; i < repeticiones; i++) {  
    the_dist += sqrt( pow(x2-x1,2) + pow(y2-y1,2) );  
}
```

Versión 1

```
for(size_t i = 0; i < repeticiones; i++) {  
    the_dist += sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) );  
}
```

Versión 2

Repeticiones: 200000000  
=====Versión 1=====  
Tiempo01 :9560[ms]  
Distancia:1.56564e+09

=====Versión 2=====  
Tiempo02 :563[ms]  
Distancia:1.56564e+09  
  
=====Versión 3=====  
Tiempo03 :552[ms]  
Distancia:1.56564e+09

=====Desempeño=====  
Aceleración A(1,2) = 16.9805  
Aceleración A(1,3) = 17.3188

```
double dx = x2-x1;  
double dy = y2-y1;  
  
for(size_t i = 0; i < repeticiones; i++) {  
    the_dist += sqrt( dx*dx + dy*dy );  
}
```

Versión 3

Algunas veces no es necesario paralelizar para obtener acelaración...

# Ejemplo

## Aceleración de código

```
for(size_t i = 0; i < repeticiones; i++) {  
    the_dist += sqrt( pow(x2-x1,2) + pow(y2-y1,2) );  
}
```

Versión 1

```
for(size_t i = 0; i < repeticiones; i++) {  
    the_dist += sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) );  
}
```

Versión 2

CXXFLAGS = -O1

Repeticiones: 200000000

====Versión 1====

Tiempo01 :187[ms]

Distancia:1.56564e+09

====Versión 2====

Tiempo02 :184[ms]

Distancia:1.56564e+09

====Versión 3====

Tiempo03 :184[ms]

Distancia:1.56564e+09

====Desempeño====

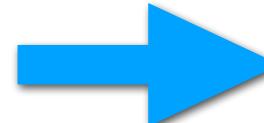
Aceleración A(1,2) = 1.0163

Aceleración A(1,3) = 1.0163

```
double dx = x2-x1;  
double dy = y2-y1;
```

```
for(size_t i = 0; i < repeticiones; i++) {  
    the_dist += sqrt( dx*dx + dy*dy );  
}
```

Versión 3

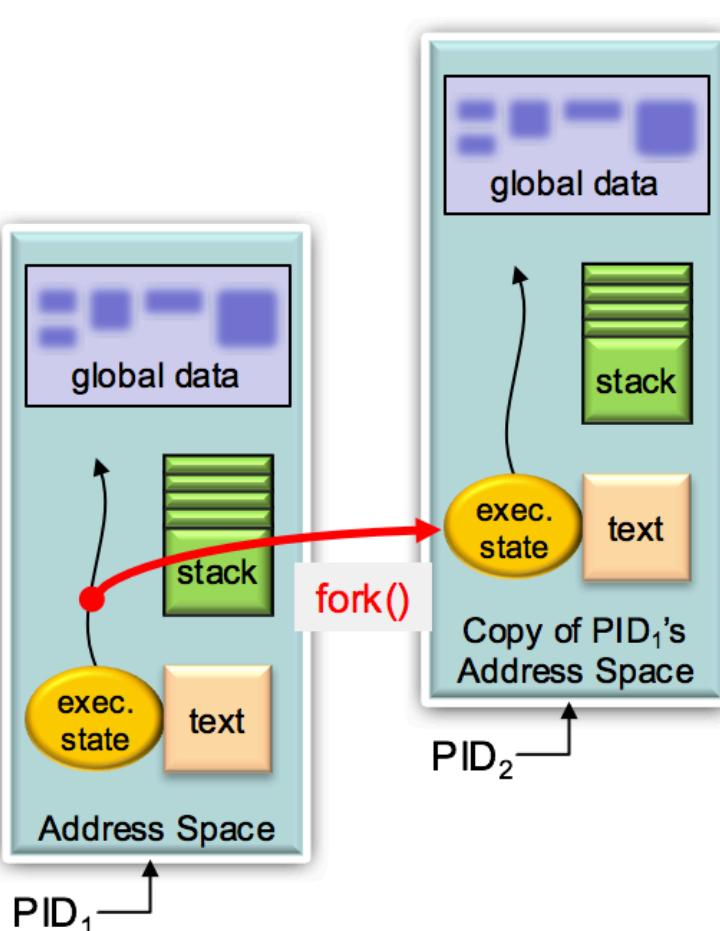
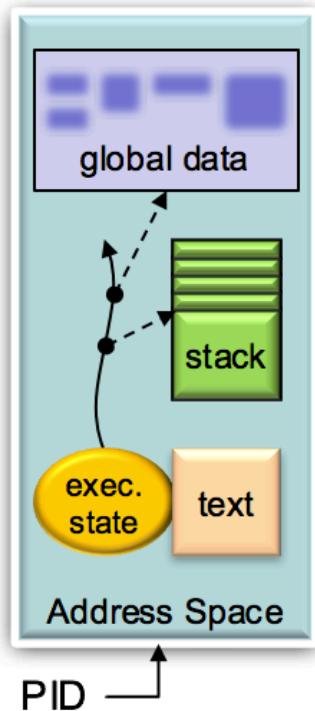


$A \approx 51.00$

Algunas veces no es necesario paralelizar para obtener acelaración...

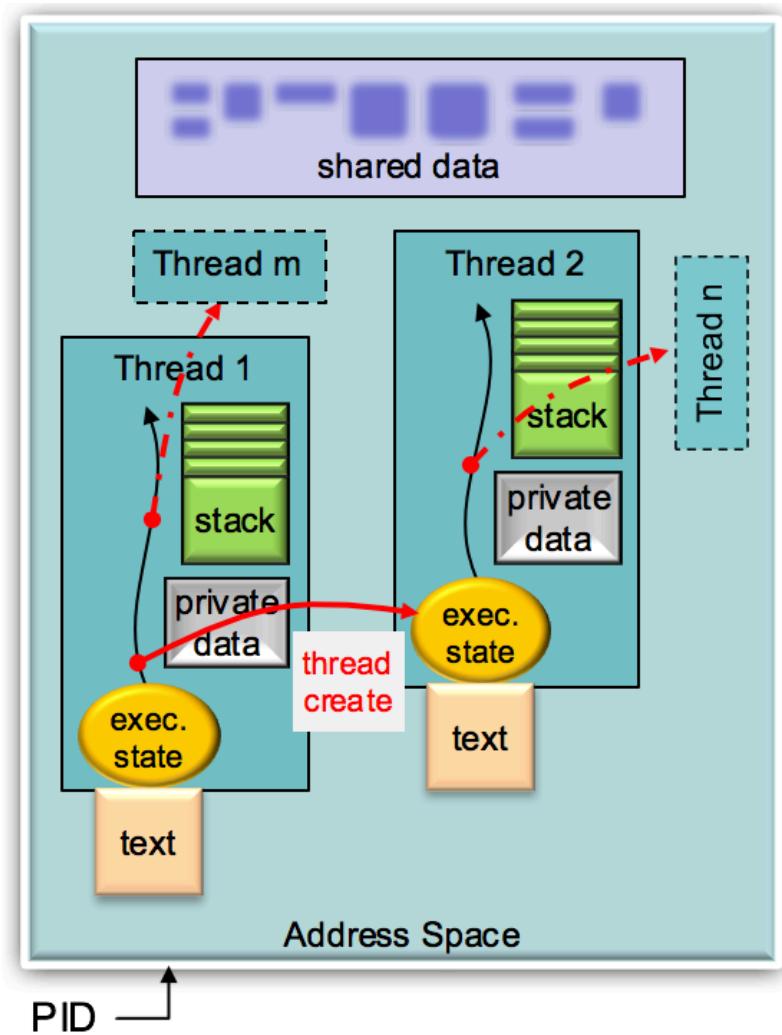
# **Procesos/Threads**

# Procesos / Threads



Standard  
UNIX process  
(single-threaded)

New process spawned via fork()



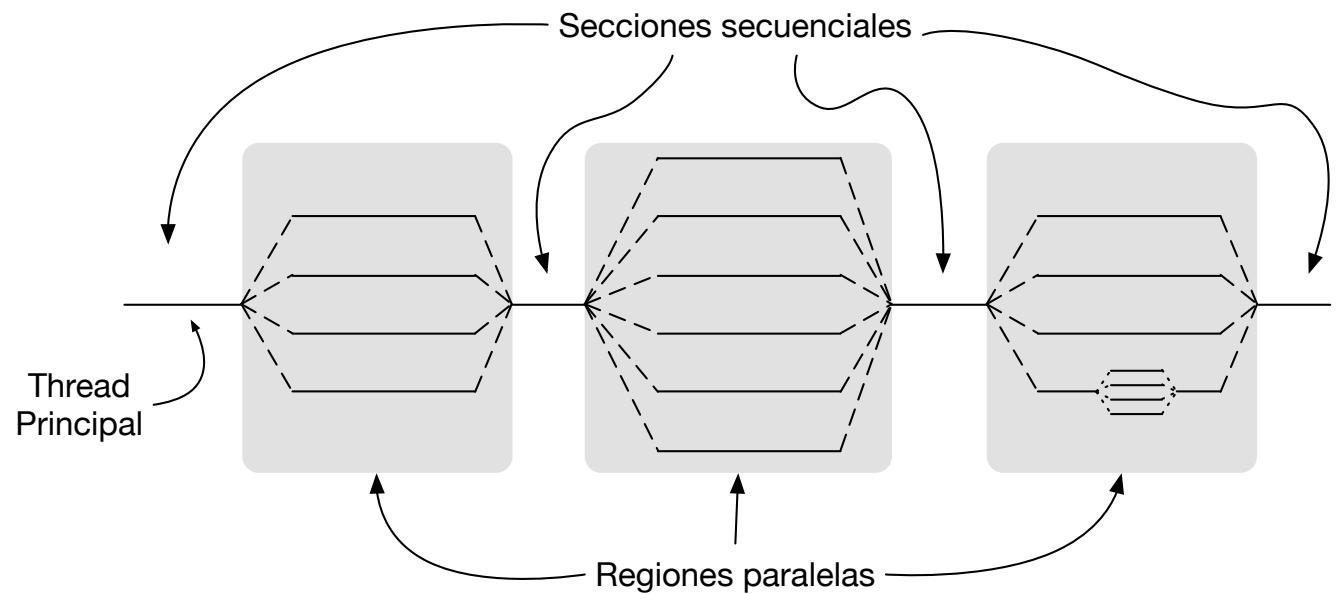
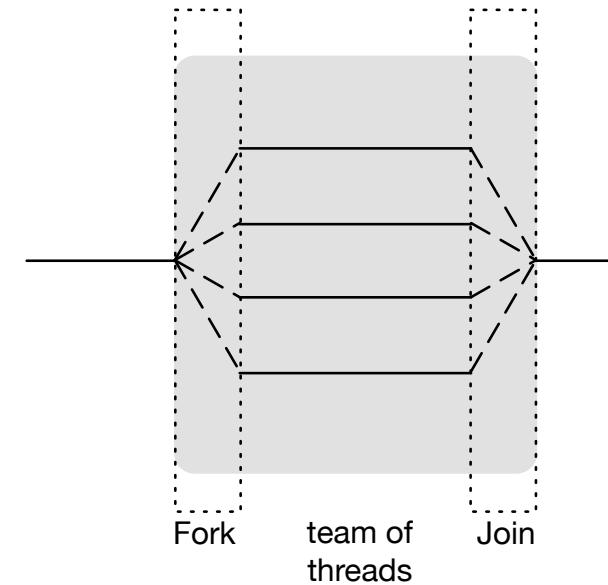
Multithreaded Application

# Procesos / Threads

Idea base

El thread principal genera un grupo (team) de threads (modelo FORK-JOIN)

El paralelismo se agrega incrementalmente a la solución secuencial



# Threads C++ (standard $\geq$ 2011)

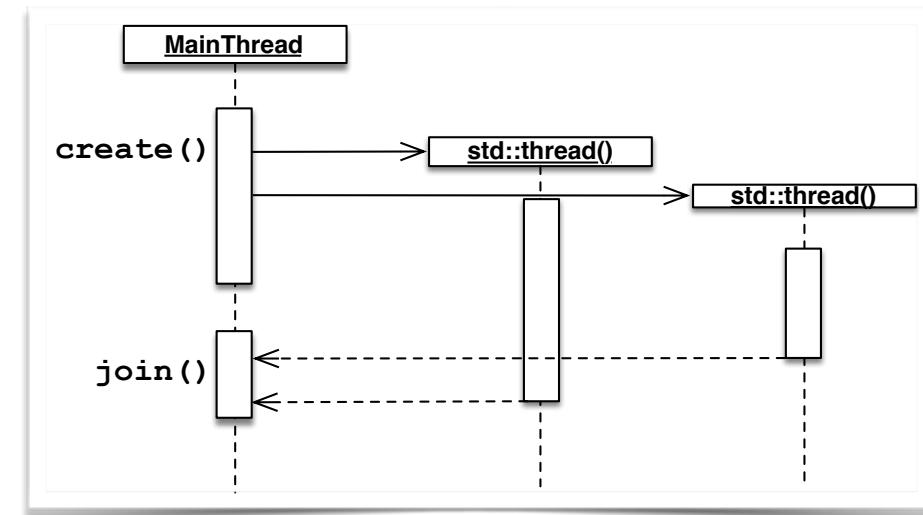
```
#include <thread>

void threadMain(){
    std::cout << "thread staff" << std::endl;
}

int main(int argc, char** argv){
    std::thread t01(threadMain);
    std::cout << "main thread" << std::endl;

    t01.join();

    return(EXIT_SUCCESS);
}
```



```
$ ./example
main thread
thread staff
```

```
$ ./example
main thread
thread staff
```

# Threads C++ (standard ≥ 2011)

```
int main(int argc, char** argv){  
  
    std::cout << "Total threads = " <<  
        std::thread::hardware_concurrency() <<  
        std::endl;  
  
    std::string msg = "hello!!!";  
    std::thread t01(threadMain, msg);  
  
    std::cout << "main thread" <<  
        std::endl;  
    std::cout << "main thread: id = " <<  
        std::this_thread::get_id() <<  
        std::endl;  
    std::cout << "main thread: child id = " <<  
        t01.get_id() <<  
        std::endl;  
  
    t01.join();  
  
    return(EXIT_SUCCESS);  
}
```

```
void threadMain(const std::string& msg){  
    std::cout << "thread child: start" <<  
        std::endl;  
  
    std::cout << "thread child: id = " <<  
        std::this_thread::get_id() <<  
        std::endl;  
  
    std::cout << "thread child: msg=" <<  
        msg <<  
        std::endl;  
}
```

**std::thread::hardware\_concurrency()**

Sugerencia para la cantidad máxima de threads

**std::this\_thread**

Referencia al thread actual

# Threads C++ (standard $\geq$ 2011)

```
void sumaParcial(const std::vector<uint32_t> &v,
                  uint32_t &suma,
                  size_t beginIndex,
                  size_t endIndex) {
    suma = 0;
    for (size_t i = beginIndex; i < endIndex; ++i){
        suma += v[i];
    }
}
```

Creación de threads (1)

Con punteros a funciones

```
=====THREADS=====
//(1) Separación del trabajo
std::thread t1(sumaParcial, std::ref(v), std::ref(suma1), 0, v.size() / 2);
std::thread t2(sumaParcial, std::ref(v), std::ref(suma2), v.size() / 2, v.size());

t1.join();
t2.join();

//(2) Reducción (Consolidación de resultados parciales)
sumaThreads = suma1 + suma2;
```

# Threads C++ (standard ≥ 2011)

```
void sumaParcial(const std::vector<uint32_t> &v,
                  uint32_t &suma,
                  size_t beginIndex,
                  size_t endIndex) {
    suma = 0;
    for (size_t i = beginIndex; i < endIndex; ++i){
        suma += v[i];
    }
}
```

```
t main(int argc, char** argv){

    uint32_t totalElementos = 100000000;

    uint32_t sumaSerial = 0;
    uint32_t suma1 = 0;
    uint32_t suma2 = 0;
    uint32_t sumaThreads = 0;

    std::vector<uint32_t> v;

    for(size_t i=0; i < totalElementos; i++){
        v.push_back(i);
    }

    //=====SERIAL=====
    sumaParcial( std::ref(v), std::ref(sumaSerial), 0, v.size() );

    //=====THREADS=====
    //((1) Separación del trabajo
    std::thread t1(sumaParcial, std::ref(v), std::ref(suma1), 0, v.size() / 2);
    std::thread t2(sumaParcial, std::ref(v), std::ref(suma2), v.size() / 2, v.size());

    t1.join();
    t2.join();

    //((2) Reducción (Consolidación de resultados parciales)
    sumaThreads = suma1 + suma2;

    std::cout << "=====Serial====" << std::endl;
    std::cout << "sumaSerial: " << sumaSerial << std::endl;

    std::cout << "=====Threads====" << std::endl;
    std::cout << "suma1: " << suma1 << std::endl;
    std::cout << "suma2: " << suma2 << std::endl;
    std::cout << "suma1 + suma2: " << sumaThreads << std::endl;

    return(EXIT_SUCCESS);
}
```

# Threads C++ (standard $\geq$ 2011)

```
class CsumaParcial{
public:
    void operator()(const std::vector<uint32_t> &v,
                     uint32_t& sum,
                     size_t beginIndex, size_t endIndex){

        sum = 0;
        for (size_t i = beginIndex; i < endIndex; ++i) {
            sum += v[i];
        }
    }
};
```

Creación de threads (2)

Con functors

```
=====THREADS=====
//(1) Separación del trabajo

CsumaParcial sumador1 = CsumaParcial();
CsumaParcial sumador2 = CsumaParcial();

std::thread t1(std::ref(sumador1), std::ref(v), std::ref(sumaParcial1), 0, v.size() / 2);
std::thread t2(std::ref(sumador2), std::ref(v), std::ref(sumaParcial2), v.size() / 2, v.size());
t1.join();
t2.join();

//(2) Reducción (Consolidación de resultados parciales)
//sumaThreads = suma1.suma() + suma2.suma();
sumaThreads = sumaParcial1 + sumaParcial2;
```

# Threads C++ (standard $\geq$ 2011)

```
int main(int argc, char** argv){  
  
    uint32_t totalElementos = 100000000;  
  
    uint32_t sumaSerial = 0;  
    uint32_t sumaParcial1 = 0;  
    uint32_t sumaParcial2 = 0;  
    uint32_t sumaThreads = 0;  
  
    std::vector<uint32_t> v;  
  
    for(size_t i=0; i < totalElementos; i++){  
        v.push_back(i);  
    }  
  
    //=====SERIAL=====  
    CsumaParcial CsumaSerial = CsumaParcial();  
    CsumaSerial( std::ref(v), std::ref(sumaSerial), 0, v.size() );  
  
    //=====THREADS=====  
    // (1) Separación del trabajo  
  
    CsumaParcial sumador1 = CsumaParcial();  
    CsumaParcial sumador2 = CsumaParcial();  
  
    std::thread t1(std::ref(sumador1), std::ref(v), std::ref(sumaParcial1), 0, v.size() / 2);  
    std::thread t2(std::ref(sumador2), std::ref(v), std::ref(sumaParcial2), v.size() / 2, v.size());  
    t1.join();  
    t2.join();  
  
    // (2) Reducción (Consolidación de resultados parciales)  
    //sumaThreads = suma1.suma() + suma2.suma();  
    sumaThreads = sumaParcial1 + sumaParcial2;  
  
    std::cout << "====Serial====" << std::endl;  
    std::cout << "sumaSerial: " << sumaSerial << std::endl;  
  
    std::cout << "====Threads====" << std::endl;  
    std::cout << "suma1: " << sumaParcial1 << std::endl;  
    std::cout << "suma2: " << sumaParcial2 << std::endl;  
    std::cout << "suma1 + suma2: " << sumaThreads << std::endl;  
  
    return(EXIT_SUCCESS);  
}
```

```
class CsumaParcial{  
public:  
    void operator()(const std::vector<uint32_t> &v,  
                    uint32_t& sum,  
                    size_t beginIndex, size_t endIndex){  
  
        sum = 0;  
        for (size_t i = beginIndex; i < endIndex; ++i) {  
            sum += v[i];  
        }  
    }  
};
```

```
int main(int argc, char** argv){  
    uint32_t totalElementos = 100000000;  
  
    uint32_t sumaSerial = 0;  
    uint32_t sumaParcial1 = 0;  
    uint32_t sumaParcial2 = 0;  
    uint32_t sumaThreads = 0;  
  
    std::vector<uint32_t> v;  
  
    for(size_t i=0; i < totalElementos; i++){  
        v.push_back(i);  
    }  
}
```

# Threads C++ (standard $\geq$ 2011)

## Creación de threads (3)

Con Lambda Functions

```
//=====SERIAL=====  
for(auto& num : v){  
    sumaSerial += num;  
}
```

```
=====THREADS=====  
//(1) Separación del trabajo  
auto sumaThread = [](std::vector<uint32_t> &v, uint32_t& suma , size_t left, size_t right) {  
    suma = 0;  
    for (size_t i = left; i < right; ++i){  
        suma += v[i];  
    }  
};  
  
std::thread t1( sumaThread, std::ref(v), std::ref(sumaParcial1), 0, v.size()/2 );  
std::thread t2( sumaThread, std::ref(v), std::ref(sumaParcial2), v.size()/2, v.size() );  
  
t1.join();  
t2.join();  
  
//(2) Reducción (Consolidación de resultados parciales)  
sumaThreads = sumaParcial1 + sumaParcial2;
```

# Tasks C++ (standard $\geq$ 2011)

```
#include <future>

auto sumaParcial = [](std::vector<uint32_t> &v, size_t left, size_t right) {
    uint32_t suma = 0;
    for (size_t i = left; i < right; ++i){
        suma += v[i];
    }

    return suma;
};

//=====THREADS=====
//(1) Separación del trabajo
auto t1 = std::async(std::launch::async, sumaParcial, std::ref(v), 0, v.size() / 2);
auto t2 = std::async(std::launch::async, sumaParcial, std::ref(v), v.size() / 2, v.size());

sumaParcial1 = t1.get();
sumaParcial2 = t2.get();

//(2) Reducción (Consolidación de resultados parciales)
//sumaThreads = suma1.suma() + suma2.suma();
sumaThreads = sumaParcial1 + sumaParcial2;
```

# Threads C++ (standard ≥ 2011)

Dos ejecuciones distintas...

```
./example
Total threads = 4
main thread
main thread: id = 0x7fff8cee2380
main thread: child id = 0x70000e0c0000
thread child: start
thread child: id = 0x70000e0c0000
thread child: msg=hello!!

./example
Total threads = 4
main thread
main thread: id = 0x7fff8cee2380
thread child: start

thread child: id = 0x700008507000
thread child: msg=main thread: child id = hello!!0x700008507000
```

# Threads C++ (standard ≥ 2011)

# Dos ejecuciones distintas...

```
void threadMain(const std::string& sym, const uint32_t cMax){
    for(size_t i = 0; i < cMax; ++i) {
        std::cout << sym;
    }
}

int main(int argc, char** argv){
    std::thread t01(threadMain, std::string("."), 600);
    std::thread t02(threadMain, std::string("*"), 600);

    t01.join();
    t02.join();

    return(EXIT_SUCCESS);
}
```

Código:  
example04

# Race Condition

Situación donde el resultado depende del orden relativo de ejecución de dos o más procesos o hilos.

```
int main(int argc, char** argv){  
  
    std::cout << "MAIN: balance=" << balance << std::endl;  
  
    std::thread t00(depositar, 0, 1000000);  
    std::thread t01(depositar, 1, 1000000);  
  
    t00.join();  
    t01.join();  
  
    std::cout << "MAIN: balance=" << balance << std::endl;  
  
    return(EXIT_SUCCESS);  
}
```

```
static volatile int balance = 0;  
  
void depositar(const uint32_t thID, const uint32_t totalDinero){  
  
    for(size_t i = 0; i < totalDinero; i++) {  
        balance = balance + 1;  
    }  
  
    std::cout << "Thread:" << thID << " FIN." << std::endl;  
}
```

```
gabriel@homeserver:~/UV/02-pthreads/Ejercicios/example05$ ./example  
MAIN: balance=0  
Thread:0 FIN.  
Thread:1 FIN.  
MAIN: balance=1044228  
gabriel@homeserver:~/UV/02-pthreads/Ejercicios/example05$ ./example  
MAIN: balance=0  
Thread:0 FIN.  
Thread:1 FIN.  
MAIN: balance=1029762
```

Código:  
example05

# Race Condition

```
#include <mutex>
```

```
std::mutex mux; //Debe ser global

thread_loop{
    mux.lock();
    //
    // Sección crítica
    //
    mux.unlock();
}
```

```
std::mutex mux;

void depositar_safe(const uint32_t thID, const uint32_t totalDinero){

    for(size_t i = 0; i < totalDinero; i++) {
        mux.lock();
        balance = balance + 1;
        mux.unlock();
    }

    std::cout << "Thread:" << thID << " FIN." << std::endl;
}
```

```
std::mutex g_DMutex;

void threadMain(const std::string& msg){

    g_DMutex.lock();
    std::cout << "thread child: start" <<
        std::endl;
    g_DMutex.unlock();

    g_DMutex.lock();
    std::cout << "thread child: id = " <<
        std::this_thread::get_id() <<
        std::endl;
    g_DMutex.unlock();

    g_DMutex.lock();
    std::cout << "thread child: msg=" <<
        msg <<
        std::endl;
    g_DMutex.unlock();
}
```