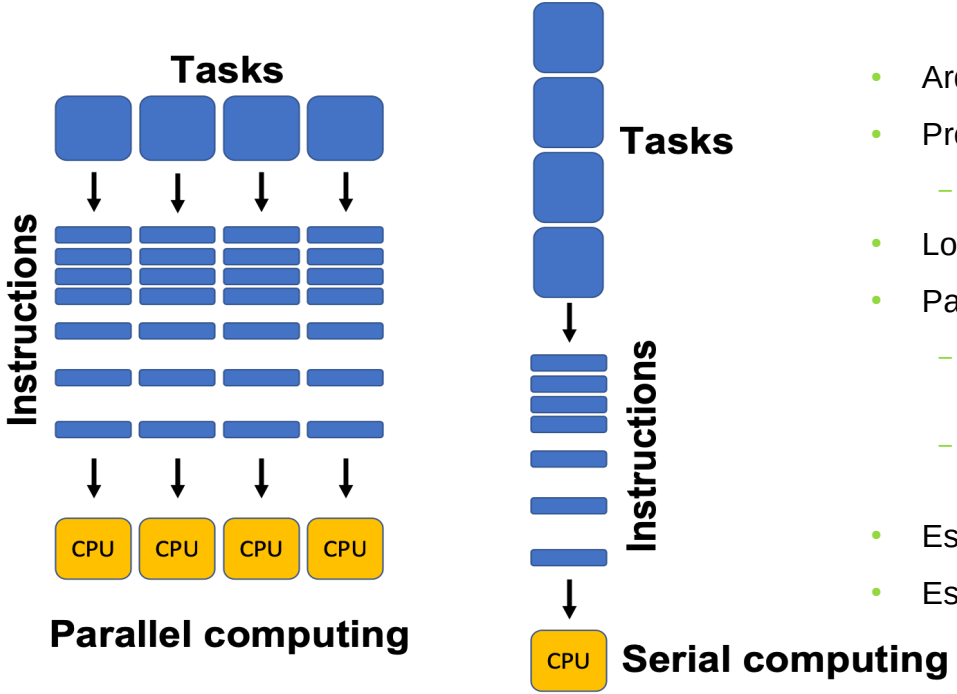
Introducción a la computación paralela

HPC Admin Tech 2023

Oporto | Portugal

Antonio Díaz Pozuelo
adiaz@iqfr.csic.es
IQFR – CSIC
Mayo 2023

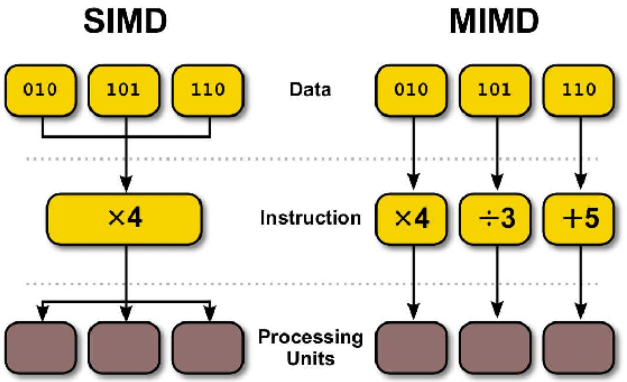
Computación serie y computación paralela



- Arquitectura y diseño: *x86_64, arm64, AVX*.
- Programación de versión serie:
  - Lenguaje de programación y compilador[2,3] (*GNU, Intel*).
- Localizar tareas/bucles paralelizables.
- Paralelizar dichas tareas/bucles:
  - Arquitectura (*SIMD*, *MIMD*)[1] y diseño: dependencia del dato, zonas críticas de memoria.
  - Lenguaje de programación, compilador[2,3] y librería de paralelización.
- Estudiar eficiencia de la paralelización (*SpeedUp).*
- Estudiar escalado de la paralelización (*HPC*)[2].

1.- Taxonomía de *Flynn*
2.- Fuera ámbito *Workshop*
3.- No se aplicarán optimizaciones

Programación de aplicaciones paralelas



Fortran
Standard Parallelism

OpenMP

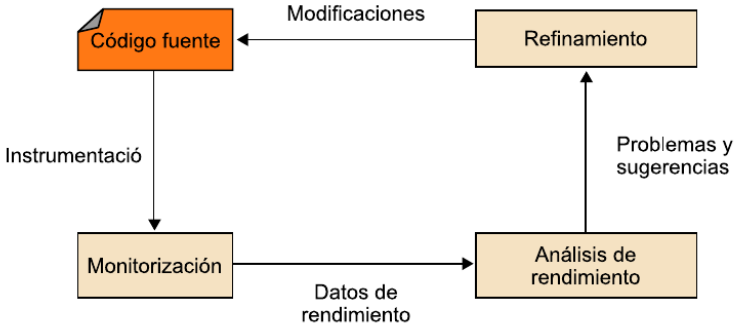
NVIDIA CUDA

Open MPI

OpenCoarrays

- *Single Instruction Multiple Data:*
  - Procesadores vectoriales.
  - *GPUs (SIMT).*
- *Multiple Instruction Multiple Data.*
  - Computación distribuida, *Middleware*.
- Memoria compartida: *SMP, UMA, NUMA.*
  - *Fortran SP, OpenMP, NVIDIA CUDA.*
- Memoria distribuida: *MPI, HPC.*
  - *OpenMPI, Fortran OpenCoarrays.*
- Hibridación:
  - *OpenMPI+OpenMP, OpenMPI+CUDA.*

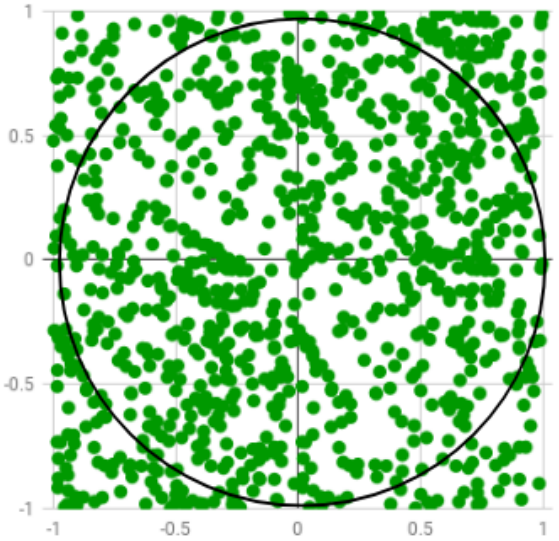
Eficiencia (*SpeedUp)* y escalado



- T: tiempo de ejecución.
- P: núcleos o cores.
- N: tamaño del problema.
- *Overhead*: comunicación, barreras acceso a memoria, sincronización hilos/cores, etc

- Límite eficiencia paralelismo: $T_{par} = {T_{seq}}/{P} \Rightarrow {T_{seq}}/{P} + T_{overhead}$
- *SpeedUp*: $S = {T_{seq}}/{T_{par}}$
- Eficiencia energética *CPU vs CPU*: $SpeedUp \geq EE_{cpu} \Leftrightarrow {T_{seq}}/{T_{par}} \geq {P_{par}}/{P_{seq}}$
- Escalado: mantener o mejorar la eficiencia al aumentar N y/o P.

Estimar el valor de PI (Monte Carlo)



$$\frac{Área_{círculo}}{Área_{cuadrado}} = \frac{Puntos_{círculo}}{Puntos_{cuadrado}} \Rightarrow \frac{\pi r^2}{4r^2} = \frac{Puntos_{círculo}}{Puntos_{cuadrado}} \Rightarrow \frac{\pi}{4} = \frac{Puntos_{círculo}}{Puntos_{cuadrado}} \Rightarrow \pi \approx 4\frac{Puntos_{círculo}}{Puntos_{cuadrado}}$$

- $P_{círculo}=0$, $P_{cuadrado}=N_{puntos}$
- Para N_{puntos}
  - Generar aleatoriamente, entre [-1,1], punto (x,y).
  - Calcular distancia al origen (0,0):
    - $d_{orig} = (x-0)^2 + (y-0)^2 \Rightarrow x^2 + y^2$
  - Si $d_{orig} <= 1 \rightarrow P_{círculo} = P_{círculo} + 1$
- Calcular $\pi = 4\frac{Puntos_{círculo}}{Puntos_{cuadrado}}$

pi_serial.py (P=1 : N=2^30)

i7 11700@2.5GHz, 16GB DDR4 3200MT/s

```python
import random

circle_points = 0
square_points = 1073741824

for i in range(square_points):
    rand_x = float(random.uniform(-1, 1))
    rand_y = float(random.uniform(-1, 1))

    origin_dist = rand_x * rand_x + rand_y * rand_y

    if origin_dist <= 1.0:
        circle_points += 1

pi = float(4 * circle_points) / float(square_points)
print("Pi ~=", pi)
```

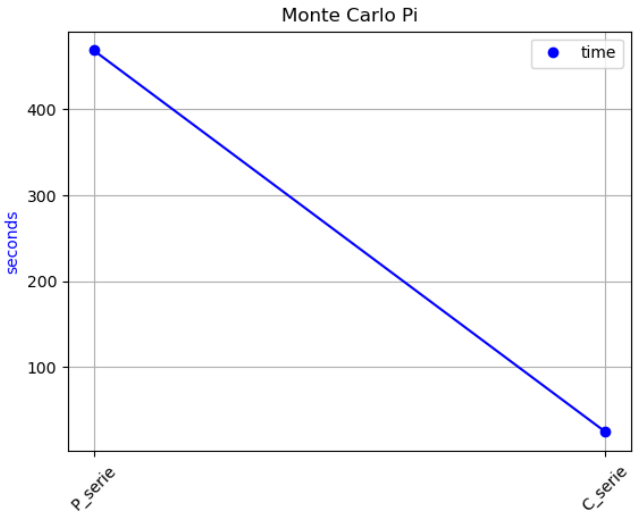
```
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ time python pi_serial.py
Pi ~= 3.1415679417550564

real    7m48,696s
user    7m48,655s
sys     0m0,036s
```

pi_serial.c (P=1 : N=2^30)

i7 11700@2.5GHz, 16GB DDR4 3200MT/s

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int i;
    int circle_points = 0;
    int square_points = 1073741824;
    float rand_x, rand_y, origin_dist, pi;

    srand(time(NULL));

    for (i = 0; i < square_points; i++)
    {
        rand_x = ((float)rand() / (RAND_MAX)) * 2 - 1;
        rand_y = ((float)rand() / (RAND_MAX)) * 2 - 1;

        origin_dist = rand_x * rand_x + rand_y * rand_y;

        if (origin_dist <= 1.0)
            circle_points++;
    }
    pi = (4 * (float)circle_points) / square_points;
    printf("Pi ~= %f\n", pi);
    return 0;
}
```

```
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ gcc -o pi_serial_c.exe pi_serial.c
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ time ./pi_serial_c.exe
Pi ~= 3.141565

real    0m24,719s
user    0m24,716s
sys     0m0,000s
```



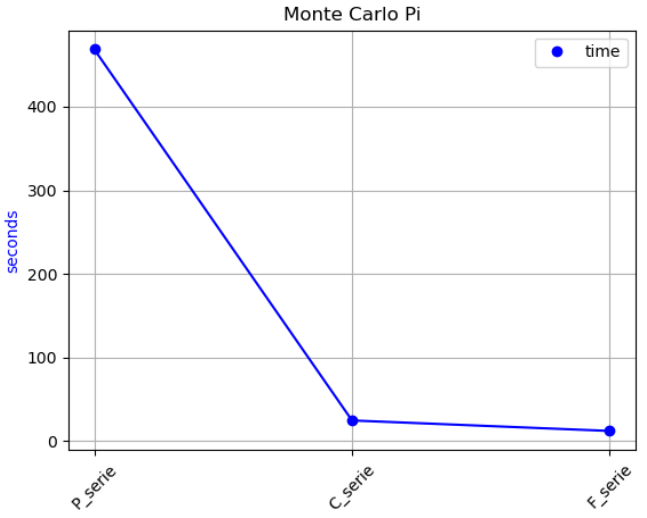
$$SpeedUp(Python/C) = \frac{468.7}{24.7} \approx 19$$

pi_serial.f90 (P=1 : N=2^30)

i7 11700@2.5GHz, 16GB DDR4 3200MT/s

```fortran
program pi_serial
   implicit none

   integer :: i
   integer :: circle_points = 0
   integer :: square_points = 1073741824
   real :: rand_x, rand_y, origin_dist, pi

   do i = 1, square_points
      call random_number(rand_x)
      rand_x = rand_x*2 - 1
      call random_number(rand_y)
      rand_y = rand_y*2 - 1

      origin_dist = rand_x*rand_x + rand_y*rand_y

      if (origin_dist <= 1.0) &
         circle_points = circle_points + 1
   end do
   pi = 4*real(circle_points)/square_points
   print *, "Pi ~=", pi

end program pi_serial
```

```
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ gfortran -o pi_serial_f.exe pi_serial.f90
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ time ./pi_serial_f.exe
 Pi ~=   3.14152384

real    0m12,181s
user    0m12,181s
sys     0m0,000s
```



$$SpeedUp(Python/C) = \frac{468.7}{24.7} \approx 19$$

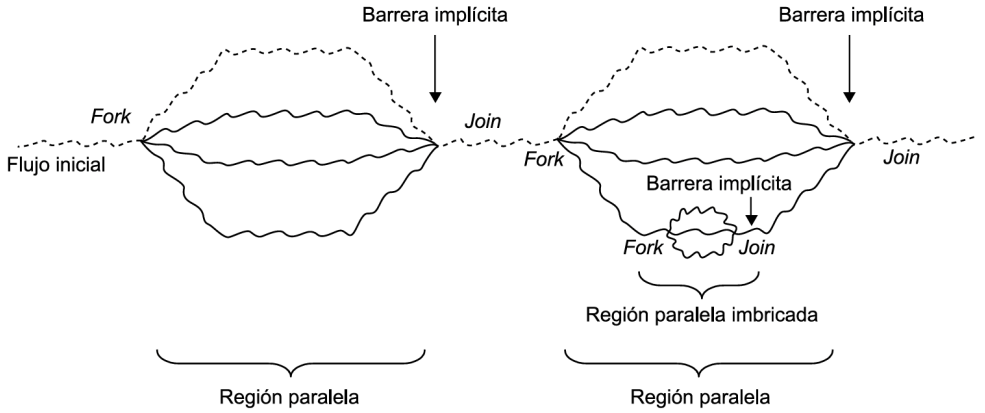
$$SpeedUp(Python/Fortran) = \frac{468.7}{12.2} \approx 38$$

$$SpeedUp(C/Fortran) = \frac{24.7}{12.2} \approx 2$$

Tareas/bucles a paralelizar

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int i;
    int circle_points = 0;
    int square_points = 1073741824;
    float rand_x, rand_y, origin_dist, pi;

    srand(time(NULL));

    for (i = 0; i < square_points; i++)
    {
        rand_x = ((float)rand() / (RAND_MAX)) * 2 - 1;
        rand_y = ((float)rand() / (RAND_MAX)) * 2 - 1;

        origin_dist = rand_x * rand_x + rand_y * rand_y;

        if (origin_dist <= 1.0)
            circle_points++;
    }
    pi = (4 * (float)circle_points) / square_points;
    printf("Pi ~= %f\n", pi);
    return 0;
}
```

- Arquitectura: *SIMD, SMP*.
- Dependencia del dato:
  - No dependencia del dato de origen
  - Cuidado ***¡RNG!***
- Zonas críticas de memoria: **circle_points**.
- Lenguaje de programación: C.
- Librería de paralelización: OpenMP.



https://www.openmp.org/spec-html/5.0/openmp.html

pi_openmp_a.c (P=2 : N=2^30)

i7 11700@2.5GHz, 16GB DDR4 3200MT/s

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int i;
    int circle_points = 0;
    int square_points = 1073741824;
    float rand_x, rand_y, origin_dist, pi;

    srand(time(NULL));

#pragma omp parallel for
    for (i = 0; i < square_points; i++)
    {
        rand_x = ((float)rand() / (RAND_MAX)) * 2 - 1;
        rand_y = ((float)rand() / (RAND_MAX)) * 2 - 1;

        origin_dist = rand_x * rand_x + rand_y * rand_y;

        if (origin_dist <= 1.0)
#pragma omp atomic
            circle_points++;
    }
    pi = (4 * (float)circle_points) / square_points;
    printf("Pi ~= %f\n", pi);
    return 0;
}
```

```
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ gcc -o pi_serial_c.exe pi_serial.c
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ time ./pi_serial_c.exe
Pi ~= 3.141550

real    0m24,737s
user    0m24,736s
sys     0m0,000s
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ gcc -fopenmp -o pi_openmp_c_a.exe pi_openmp_a.c
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ export OMP_NUM_THREADS=2 && time ./pi_openmp_c_a.exe
Pi ~= 3.141348

real    7m14,933s
user    7m35,609s
sys     6m53,205s
```

- No mantiene/mejora eficiencia
- *RNG(rand)* no soporta paralelismo

OpenMP

https://www.openmp.org/spec-html/5.0/openmp.html

pi_openmp_b.c (P=2 : N=2^30)

i7 11700@2.5GHz, 16GB DDR4 3200MT/s

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

int main()
{
    int i;
    int circle_points = 0;
    int square_points = 1073741824;
    float rand_x, rand_y, origin_dist, pi;
    unsigned int seed;

#pragma omp parallel
    seed = ((unsigned)time(NULL) & 0xFFFFFFF0) \
             | (omp_get_thread_num() + 1);
    srand(seed);

#pragma omp parallel for \
firstprivate(rand_x, rand_y, origin_dist, seed)
    for (i = 0; i < square_points; i++)
    {
        rand_x = ((float)rand_r(&seed) / (RAND_MAX)) * 2 - 1;
        rand_y = ((float)rand_r(&seed) / (RAND_MAX)) * 2 - 1;

        origin_dist = rand_x * rand_x + rand_y * rand_y;

        if (origin_dist <= 1.0)
#pragma omp atomic
            circle_points++;
    }
    pi = (4 * (float)circle_points) / square_points;
    printf("Pi ~= %f\n", pi);
    return 0;
}
```

```
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ gcc -o pi_serial_c.exe pi_serial.c
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ time ./pi_serial_c.exe
Pi ~= 3.141587

real    0m24,746s
user    0m24,741s
sys     0m0,005s
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ gcc -fopenmp -o pi_openmp_c_b.exe pi_openmp_b.c
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ export OMP_NUM_THREADS=2 && time ./pi_openmp_c_b.exe
Pi ~= 3.141595

real    0m12,158s
user    0m24,159s
sys     0m0,153s
```

- Mejora eficiencia: $SpeedUp = \frac{24.7}{12.2} \approx 2$
- *atomic*: sincronización de hilos/cores en acceso a zona crítica de memoria

OpenMP

https://www.openmp.org/spec-html/5.0/openmp.html

pi_openmp_c.c (P=2 : N=2^30)

i7 11700@2.5GHz, 16GB DDR4 3200MT/s

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

int main()
{
    int i;
    int circle_points = 0;
    int square_points = 1073741824;
    float rand_x, rand_y, origin_dist, pi;
    unsigned int seed;

#pragma omp parallel
    seed = ((unsigned)time(NULL) & 0xFFFFFFF0) \
             | (omp_get_thread_num() + 1);
    srand(seed);

#pragma omp parallel for \
firstprivate(rand_x, rand_y, origin_dist, seed) \
reduction(+:circle_points)
    for (i = 0; i < square_points; i++)
    {
        rand_x = ((float)rand_r(&seed) / (RAND_MAX)) * 2 - 1;
        rand_y = ((float)rand_r(&seed) / (RAND_MAX)) * 2 - 1;

        origin_dist = rand_x * rand_x + rand_y * rand_y;

        if (origin_dist <= 1.0)
            circle_points++;
    }
    pi = (4 * (float)circle_points) / square_points;
    printf("Pi ~= %f\n", pi);
    return 0;
}
```

```
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ gcc -o pi_serial_c.exe pi_serial.c
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ time ./pi_serial_c.exe
Pi ~= 3.141580

real    0m24,751s
user    0m24,750s
sys     0m0,000s
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ gcc -fopenmp -o pi_openmp_c_c.exe pi_openmp_c.c
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ export OMP_NUM_THREADS=2 && time ./pi_openmp_c_c.exe
Pi ~= 3.141595

real    0m5,360s
user    0m10,712s
sys     0m0,004s
```

- Mejora eficiencia: $SpeedUp = \frac{24.7}{12.2} \approx 4.6$
- *reduction*: no sincronización de hilos/cores en acceso a zona crítica de memoria

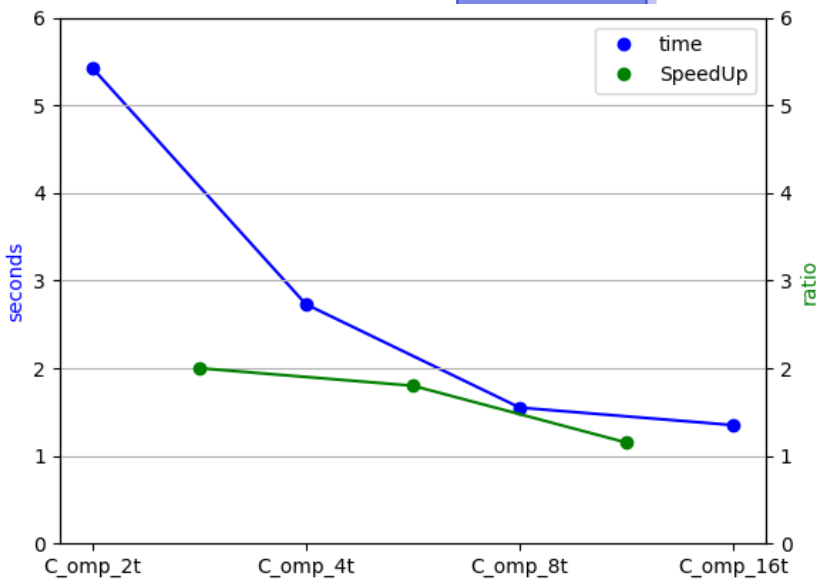
OpenMP

https://www.openmp.org/spec-html/5.0/openmp.html

Análisis: pi_openmp_c.c (P=2,4,8,16 : N=2^30)

i7 11700@2.5GHz, 16GB DDR4 3200MT/s

```
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ gcc -fopenmp -o pi_openmp_c_c.exe pi_openmp_c.c
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ export OMP_NUM_THREADS=2 && time ./pi_openmp_c_c.exe
Pi ~= 3.141595

real    0m5,431s
user    0m10,846s
sys     0m0,012s
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ export OMP_NUM_THREADS=4 && time ./pi_openmp_c_c.exe
Pi ~= 3.141595

real    0m2,734s
user    0m10,919s
sys     0m0,005s
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ export OMP_NUM_THREADS=8 && time ./pi_openmp_c_c.exe
Pi ~= 3.141592

real    0m1,557s
user    0m12,341s
sys     0m0,089s
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ export OMP_NUM_THREADS=16 && time ./pi_openmp_c_c.exe
Pi ~= 3.141592

real    0m1,354s
user    0m19,106s
sys     0m0,516s
```



$$S(2t/4t)=\frac{5.43}{2.73}\approx 2 \geq 2=\frac{4}{2}=EE(2t/4t)$$

$$S(4t/8t)=\frac{2.73}{1.55}\approx 1.8 \leq 2=\frac{8}{4}=EE(4t/8t)$$

$$S(8t/16t)=\frac{1.55}{1.35}\approx 1.15 < 2=\frac{16}{8}=EE(8t/16t)$$

- *SpeedUp* (decreciente) y eficiencia energética: escalado óptimo hasta 8 procs/cores

OpenMP

pi_fsp.f90 (P=*arch/comp dep* : N=2^30)

i7 11700@2.5GHz, 16GB DDR4 3200MT/s

```
program pi_fsp
   implicit none

   integer :: i
   integer, parameter :: square_points = 1073741824
   real :: pi
   integer, allocatable:: circle_points(:)
   real, allocatable :: rand_x(:), rand_y(:)

   allocate(circle_points(square_points))
   allocate(rand_x(square_points), rand_y(square_points))

   call random_number(rand_x)
   call random_number(rand_y)
   rand_x = (rand_x*2 - 1)*(rand_x*2 - 1)
   rand_y = (rand_y*2 - 1)*(rand_y*2 - 1)

   do concurrent(i = 1:square_points)
      if ((rand_x(i) + rand_y(i)) <= 1.0) &
         circle_points(i) = 1
   end do

   pi = 4*real(sum(circle_points))/square_points
   print *, "Pi ~=", pi

end program pi_fsp
```

```
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ gfortran -Ofast -o pi_serial_f.exe pi_serial.f90
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ time ./pi_serial_f.exe
 Pi ~=   3.14155102

real    0m11,485s
user    0m11,480s
sys     0m0,004s
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ gfortran -Ofast -o pi_serial_f.exe pi_serial.f90
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ time ./pi_fsp.exe
 Pi ~=   3.14155936

real    0m7,965s
user    0m5,656s
sys     0m2,309s
```

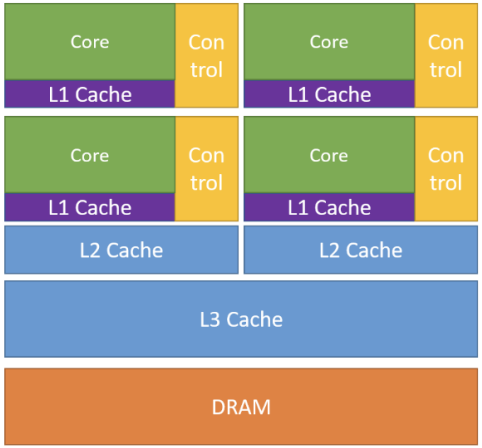
$$SpeedUp(seq/par) = \frac{11.49}{7.97} \approx 1.45$$

- No dependencia del dato de entrada ni zonas críticas de memoria
- Funciones/subrutinas/bucles a paralelizar tienen que ser puras
- Uso de funciones/subrutinas elementales
- Optimización en compilación requerida
- Eficiencia y escalado dependiente de la arquitectura y compilador

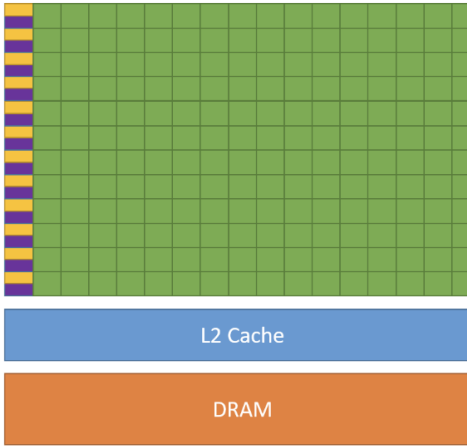
Fortran
Standard Parallelism

https://docs.nvidia.com/cuda

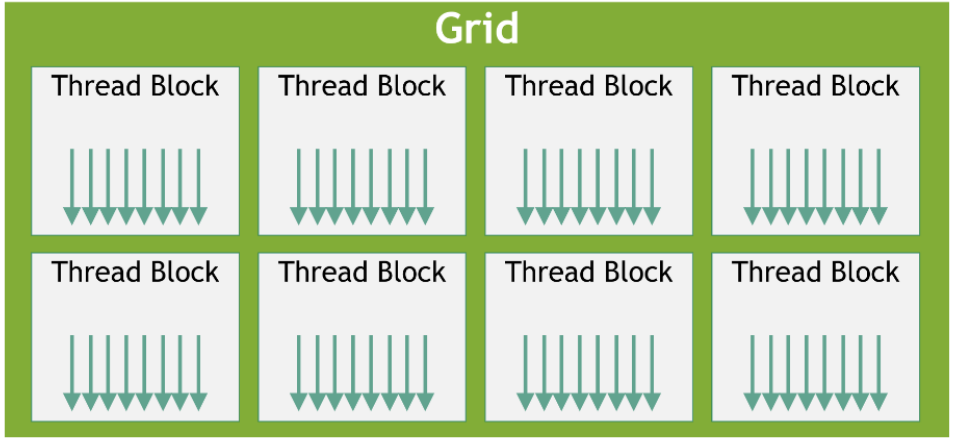
NVIDIA CUDA



CPU



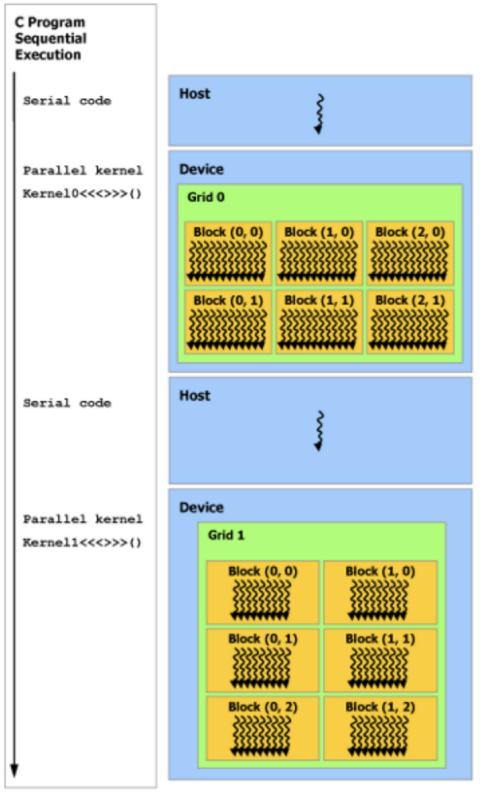
GPU



- Respecto a una *CPU*, una *GPU* posee más transistores para el procesamiento de datos
- Cada hilo se ejecuta en una unidad de procesamiento de la *GPU*
- Los hilos se agrupan en bloques, los cuales se ejecutan de forma paralela
- Los bloques se agrupan en un *Grid*

https://docs.nvidia.com/cuda

NVIDIA CUDA



- *SIMT: Single Instruction Multiple Thread*
- *CPU* → *host*, *GPU* → *device*
- *CPU-RAM* ↔ *GPU-RAM*: transferencia datos
- *GPU-RAM*: *shared, global, constant, texture*
- Modelo de programación:
  - *CPU* [Código serie]: preprocesamiento (*E/S*, inicializar memoria), *CPU-RAM* → *GPU-RAM* y lanzar *kernel GPU*
  - *GPU* [Código paralelo *kernel* (asíncrono)]: obtener identificador de hilo, todos los hilos realizarán las mismas operaciones sobre los datos transferidos a la *GPU-RAM*
  - *CPU* [Código serie]: *GPU-RAM* → *CPU-RAM* y posprocesamiento (*E/S*, liberar memoria)

https://docs.nvidia.com/hpc-sdk/compilers/cuda-fortran-prog-guide/

pi_cuda_main/mod_a.cuf (N=2^30) i7 11700@2.5GHz, 16GB DDR4 3200MT/s, GeForce RTX 3060 (12GB)

```
program pi_cuda
   use cudafor
   use pi_mod
   implicit none

   integer :: i, n_blocks
   integer, parameter :: n_threads = 128
   integer, parameter :: square_points = 1073741824
   real :: pi
   integer, allocatable:: circle_points(:)
   real, allocatable :: rand_x(:), rand_y(:)

   allocate (circle_points(square_points))
   allocate (dev_circle_points(square_points))
   allocate (rand_x(square_points), rand_y(square_points))
   allocate (dev_rand_x(square_points))

   call random_number(rand_x)
   call random_number(rand_y)
   rand_x = (rand_x*2 - 1)*(rand_x*2 - 1)
   rand_y = (rand_y*2 - 1)*(rand_y*2 - 1)
   rand_x = rand_x + rand_y

   dev_rand_x = rand_x

   n_blocks = square_points/n_threads
   if (mod(n_blocks, n_threads) > 0) n_blocks = n_blocks + 1
   call pi_kernel<<<n_blocks, n_threads>>>(square_points)

   circle_points = dev_circle_points

   pi = 4*real(sum(circle_points))/square_points
   print *, "Pi ~=", pi

end program pi_cuda
```

```
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ nvfortran -o pi_cuda_a.exe pi_cuda_mod_a.cuf pi_cuda_main_a.cuf
pi_cuda_mod_a.cuf:
pi_cuda_main_a.cuf:
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ time ./pi_cuda_a.exe
 Pi ~=    3.141579

real    0m16,930s
user    0m12,607s
sys     0m4,319s
```

```
module pi_mod
   implicit none
   real, allocatable, device :: dev_rand_x(:)
   integer, allocatable, device:: dev_circle_points(:)
contains
   attributes(global) subroutine pi_kernel(square_points)
      integer, value, intent(in) :: square_points
      integer :: i
      i = (blockidx%x - 1)*blockdim%x + threadidx%x

      if (i <= square_points) then
         if (dev_rand_x(i) <= 1.0) then
            dev_circle_points(i) = 1
         else
            dev_circle_points(i) = 0
         end if
      end if
   end subroutine pi_kernel

end module pi_mod
```

- Transferencias memoria *CPU* ↔ *GPU*
- ¡Cada vector ocupa 4 GB!
- *RNG* en *CPU*

NVIDIA CUDA

https://docs.nvidia.com/hpc-sdk/compilers/cuda-fortran-prog-guide/

pi_cuda_main/mod_b.cuf (N=2^30) i7 11700@2.5GHz, 16GB DDR4 3200MT/s, GeForce RTX 3060 (12GB)

```
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ nvfortran -cudalib=curand -o pi_cuda_b.exe pi_cuda_mod_b.cuf pi_cuda_main_b.cuf
pi_cuda_mod_b.cuf:
pi_cuda_main_b.cuf:
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ time ./pi_cuda_b.exe
 Pi ~=    3.141623

real    0m2,925s
user    0m2,020s
sys     0m0,902s
```

- Reducción transferencias memoria *CPU* ↔ *GPU*
- *RNG* dentro del *kernel* de la *GPU* (*curand*)
- Uso de la hora del sistema para la semilla del *RNG*


```
program pi_cuda
   use cudafor
   use pi_mod
   implicit none

   integer :: i, n_blocks, time
   integer, parameter :: n_threads = 128
   integer, parameter :: square_points = 1073741824
   real :: pi
   integer, allocatable:: circle_points(:)
   integer date_time(8)
   character*10 b(3)
   call date_and_time(b(1), b(2), b(3), date_time)
   read(b(2),*) time

   allocate (circle_points(square_points))
   allocate (dev_circle_points(square_points))

   n_blocks = square_points/n_threads
   if (mod(n_blocks, n_threads) > 0) n_blocks = n_blocks + 1

   call pi_kernel<<<n_blocks, n_threads>>>(square_points, time)
   circle_points = dev_circle_points

   pi = 4*real(sum(circle_points))/square_points
   print *, "Pi ~=", pi

end program pi_cuda
```

```
module pi_mod
   use curand_device
   implicit none
   integer, allocatable, device:: dev_circle_points(:)
contains
   attributes(global) subroutine pi_kernel(square_points, time)
      integer, value, intent(in) :: square_points
      integer, value, intent(in) :: time
      type(curandStateXORWOW) :: h
      integer(8) :: seed, seq, offset
      integer :: i
      real(8) :: rand_x, rand_y, origin_dist

      i = (blockidx%x - 1)*blockdim%x + threadidx%x
      seed = i * 1024 + time
      seq = 0
      offset = 0
      call curand_init(seed, seq, offset, h)

      if (i <= square_points) then
         rand_x = curand_uniform_double(h) *2 - 1
         rand_y = curand_uniform_double(h) *2 - 1

         origin_dist = rand_x*rand_x + rand_y*rand_y

         if (origin_dist <= 1.0) then
            dev_circle_points(i) = 1
         else
            dev_circle_points(i) = 0
         end if
      end if
   end subroutine pi_kernel

end module pi_mod
```

https://docs.nvidia.com/hpc-sdk/compilers/cuda-fortran-prog-guide/

pi_cuda_main/mod_c.cuf (N=2^30) i7 11700@2.5GHz, 16GB DDR4 3200MT/s, GeForce RTX 3060 (12GB)

```
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ nvfortran -cudalib=curand -o pi_cuda_c.exe pi_cuda_mod_c.cuf pi_cuda_main_c.cuf
pi_cuda_mod_c.cuf:
pi_cuda_main_c.cuf:
(minimal) adpozuelo@tyrion:~/ownCloud/Development/mc_pi$ time ./pi_cuda_c.exe
 Pi ~=    3.141734

real    0m0,230s
user    0m0,150s
sys     0m0,075s
```

- Mínima transferencia memoria *CPU ↔ GPU*
- Uso de operaciones atómicas en *kernel*
- *RNG* en *GPU* (*curand*)

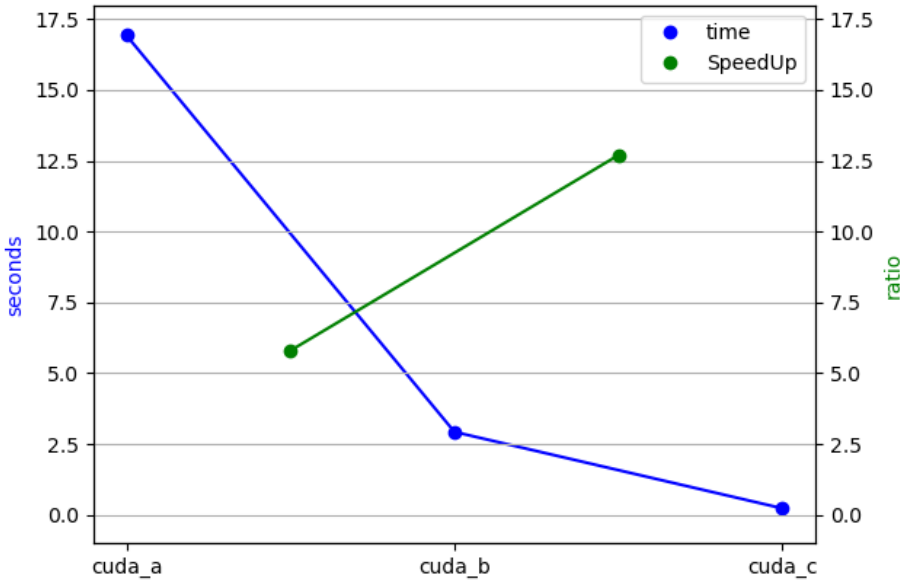

```
program pi_cuda
   use cudafor
   use pi_mod
   implicit none

   integer :: i, n_blocks, time, circle_points
   integer, parameter :: n_threads = 128
   integer, parameter :: square_points = 1073741824
   real :: pi
   integer date_time(8)
   character*10 b(3)
   call date_and_time(b(1), b(2), b(3), date_time)
   read(b(2),*) time

   dev_circle_points = 0

   n_blocks = square_points/n_threads
   if (mod(n_blocks, n_threads) > 0) n_blocks = n_blocks + 1

   call pi_kernel<<<n_blocks, n_threads>>>(square_points, time)
   circle_points = dev_circle_points

   pi = 4*real(circle_points)/square_points
   print *, "Pi ~=", pi

end program pi_cuda
```

```
module pi_mod
   use curand_device
   implicit none
   integer, device:: dev_circle_points
contains
   attributes(global) subroutine pi_kernel(square_points, time)
      integer, value, intent(in) :: square_points
      integer, value, intent(in) :: time
      type(curandStateXORWOW) :: h
      integer(8) :: seed, seq, offset
      integer :: i, istat
      real(8) :: rand_x, rand_y, origin_dist

      i = (blockidx%x - 1)*blockdim%x + threadidx%x
      seed = i * 1024 + time
      seq = 0
      offset = 0
      call curand_init(seed, seq, offset, h)

      if (i <= square_points) then
         rand_x = curand_uniform_double(h) *2 - 1
         rand_y = curand_uniform_double(h) *2 - 1

         origin_dist = rand_x*rand_x + rand_y*rand_y

         if (origin_dist <= 1.0) &
            istat = atomicadd(dev_circle_points, 1)
      end if
   end subroutine pi_kernel

end module pi_mod
```

https://docs.nvidia.com/hpc-sdk/compilers/cuda-fortran-prog-guide/

Análisis: pi_cuda_main/mod.cuf (N=2^30)

i7 11700@2.5GHz, 16GB DDR4 3200MT/s, GeForce RTX 3060 (12GB)

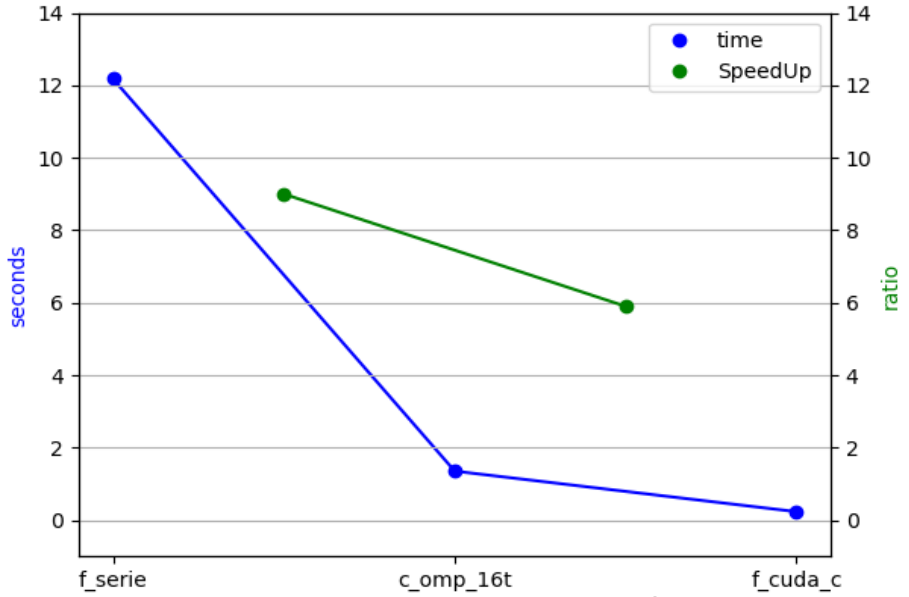


- *SpeedUp* (creciente)*:*

$$S(a/b)=\frac{16.93}{2.93}\approx 5.98 \qquad S(b/c)=\frac{2.93}{0.23}\approx 12.7 \qquad S(a/c)=\frac{16.93}{0.23}\approx 73.6$$

- Cálculo de eficiencia energética mediante consumo/tiempo y no mediante *SpeedUP*: EE_{CPU} *vs* $EE_{CPU} \neq EE_{CPU}$ *vs* EE_{GPU}
- Consideraciones importantes:
  - Minimizar transferencias memoria *CPU* ↔ *GPU*
  - Uso de operaciones atómicas en *kernel*
  - *RNG* válidos para paralelización

Análisis códigos más rápidos y conclusiones



$$S(p_{serie}/f_{cuda-c}) = \frac{468.7}{0.23} \approx 2038$$

- *SpeedUp* (decreciente):

$$S(f_{serie}/c_{omp\,16t}) = \frac{12.18}{1.35} \approx 9 \qquad S(c_{omp\,16t}/f_{cuda-c}) = \frac{1.35}{0.23} \approx 5.9$$

- La computación paralela permite mitigar y/o abordar:
  - Límites físicos procesadores (nm, ºC)
  - Límites consumo energético
  - Problemas *NP* (*Nondeterministic Polynomial*)
  - Procesamiento de grandes cantidades de datos (*Big Data*)
- Sin la computación paralela no sería posible resolver problemas complejos, en tiempos razonables

Futuro ¡Muchas gracias por vuestra atención!

- *GPUs*: coste bajo/medio y rendimiento muy alto
- *SMP*: aumenta relación cores/procesador: coste medio y rendimiento alto
- *Infiniband/Similares*: coste y rendimiento altos
- Soluciones híbridas: (*SMP/MPI*)+*GPU* [**aceleradores**]
- Lenguajes de programación convertidos en un negocio empresarial
  - Reinventar la rueda una y otra vez
  - Efecto “caja negra” o “darle al botón”
  - Afecta a la formación/educación
- Descarga del *Workshop*:
  - https://github.com/adpozuelo/Workshop_ICP