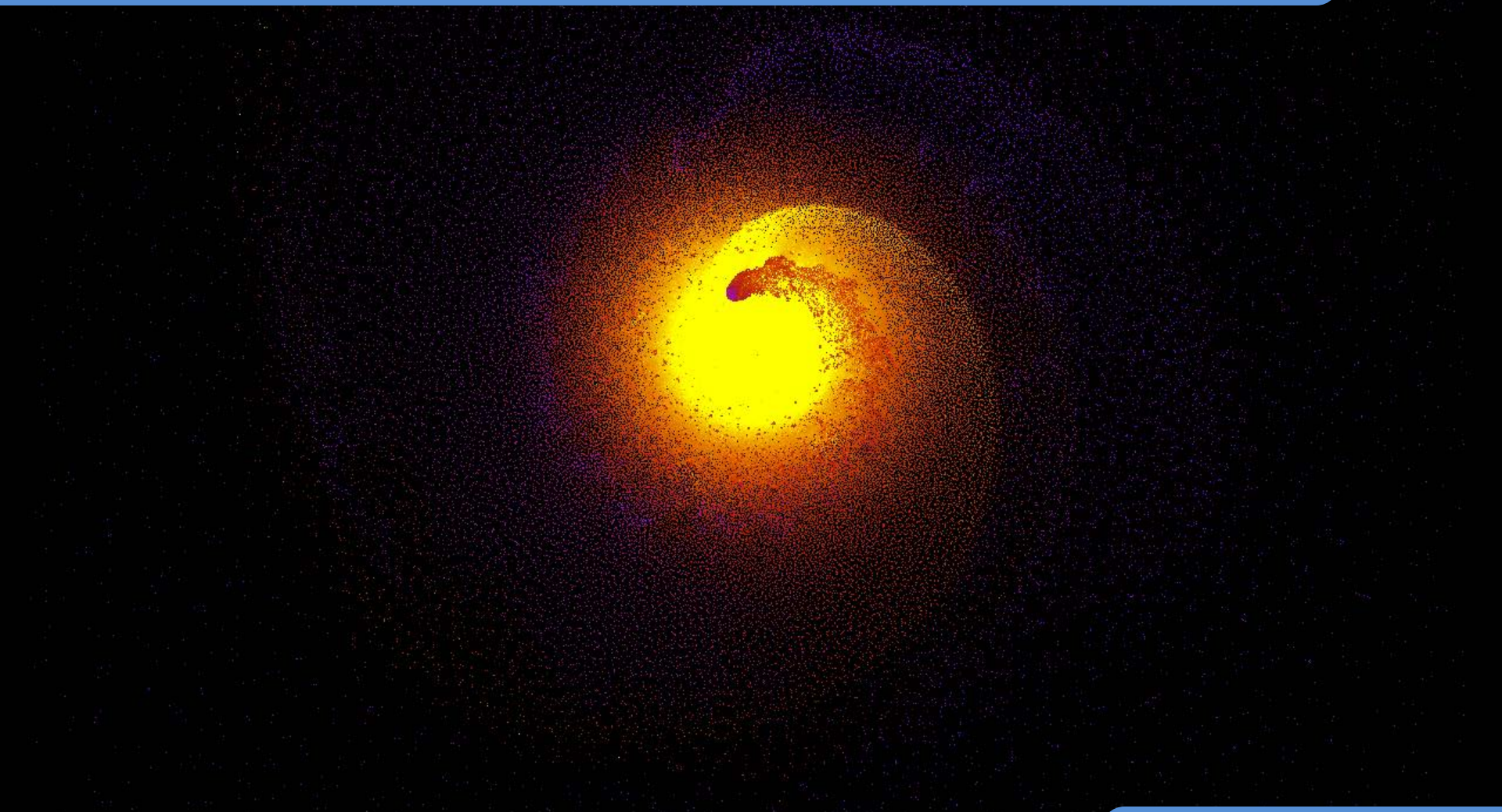
Introduction to Parallel Computing



Dr. Rubén M. Cabezón
Dr. Aurélien Cavelan

sciCORE Center for scientific computing

University of Basel

OpenMP

Outline

Introduction

OpenMP

- ❑ Scheduling
- ❑ Variables scope
- ❑ Reduction
- ❑ Atomic
- ❑ Collapse
- ❑ Orphaned directives

Python

- ❑ Multiprocessing
- ❑ Numba

OpenMP and GPU offloading

MPI (just a smidge)

Launching in clusters (SLURM)

09:00 – 10:30: Lecture
10:30 – 10:45: Coffee break
10:45 – 12:00: Lecture

12:00 – 13:30: Lunch

13:30 – 15:00: Lecture
15:00 – 15:15: Coffee break
15:15 – 17:00: Lecture

First thing to do:

Download and extract the files for the course:

From the browser: just go to https://bit.ly/2KzMEJ4
From the terminal: wget --trust-server-names https://bit.ly/2KzMEJ4

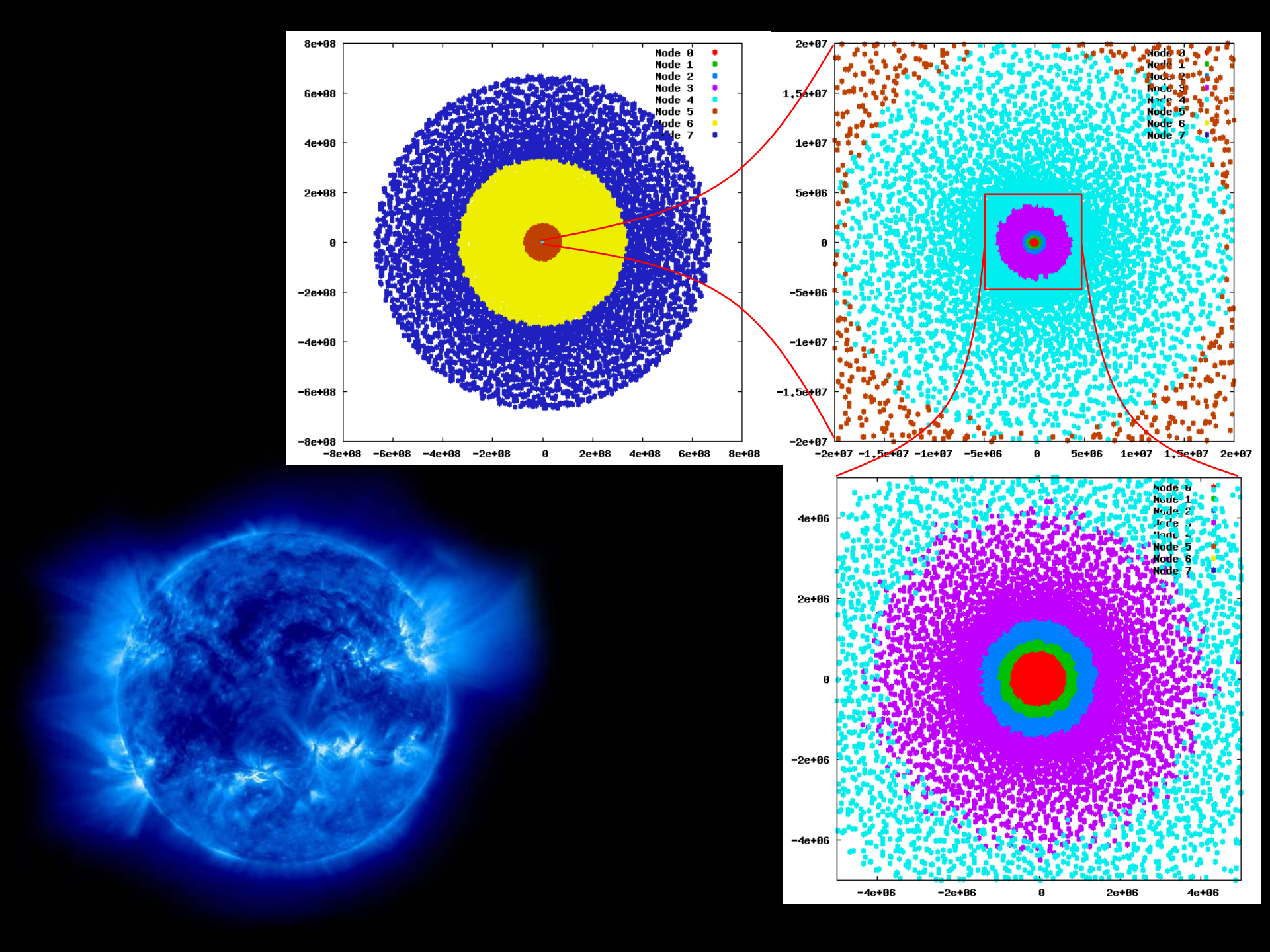
tar xvf openmp_course.tar.gz

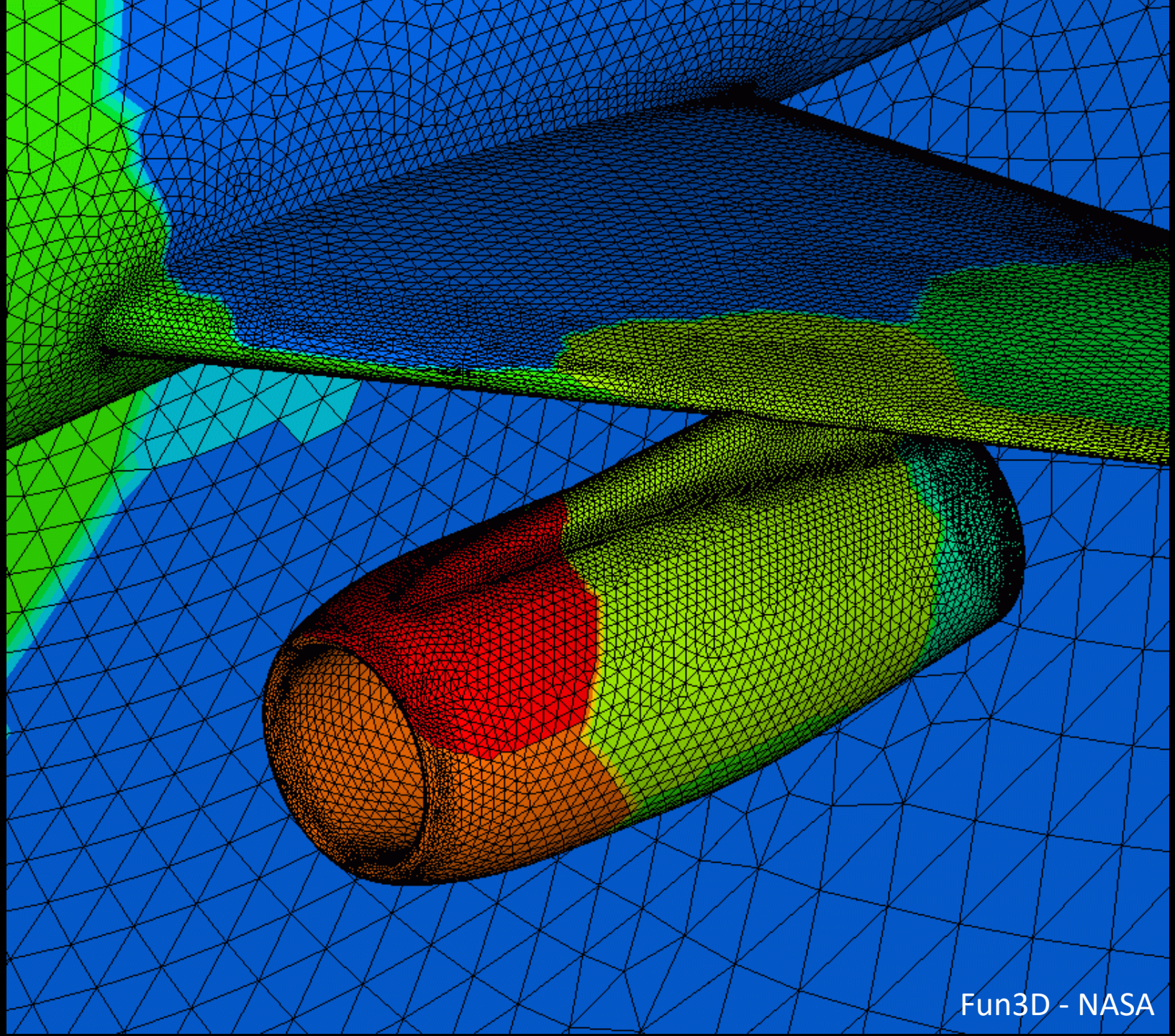
This will create a directory named openmp_course/ with some files we will need

You can find there the slides of the course (exercises are redacted)

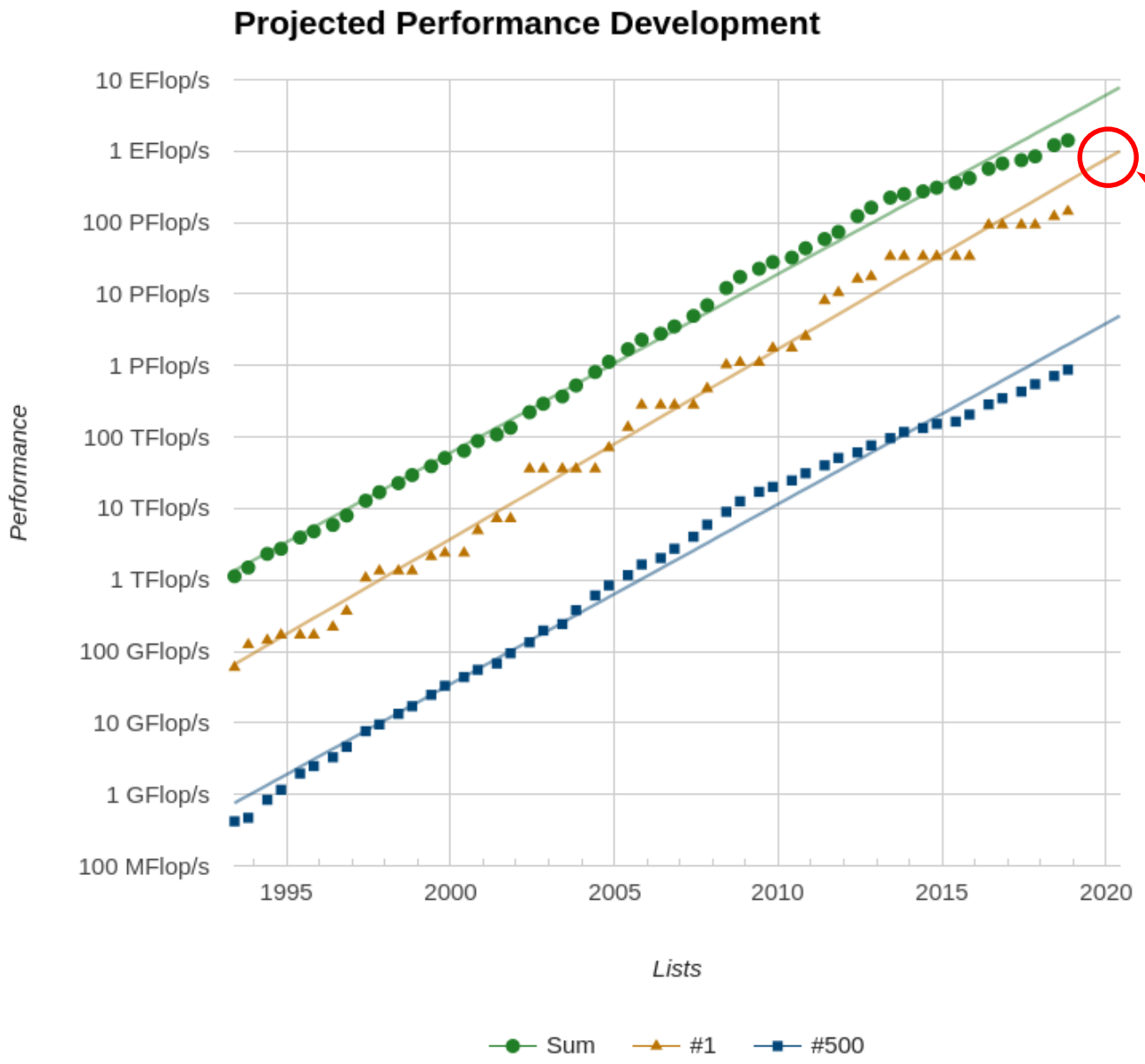
Multiple executions of the same code, applied to different sections of the input data.

Split the problem in subdomains that are partially solved independently.


Node 0
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
Node 7
8e+08
6e+08
4e+08
2e+08
0
-2e+08
-4e+08
-6e+08
-8e+08
2e+07
1.5e+07
1e+07
5e+06
-5e+06
-1e+07
-1.5e+07
-2e+07
4e+06
2e+06
-2e+06
-4e+06

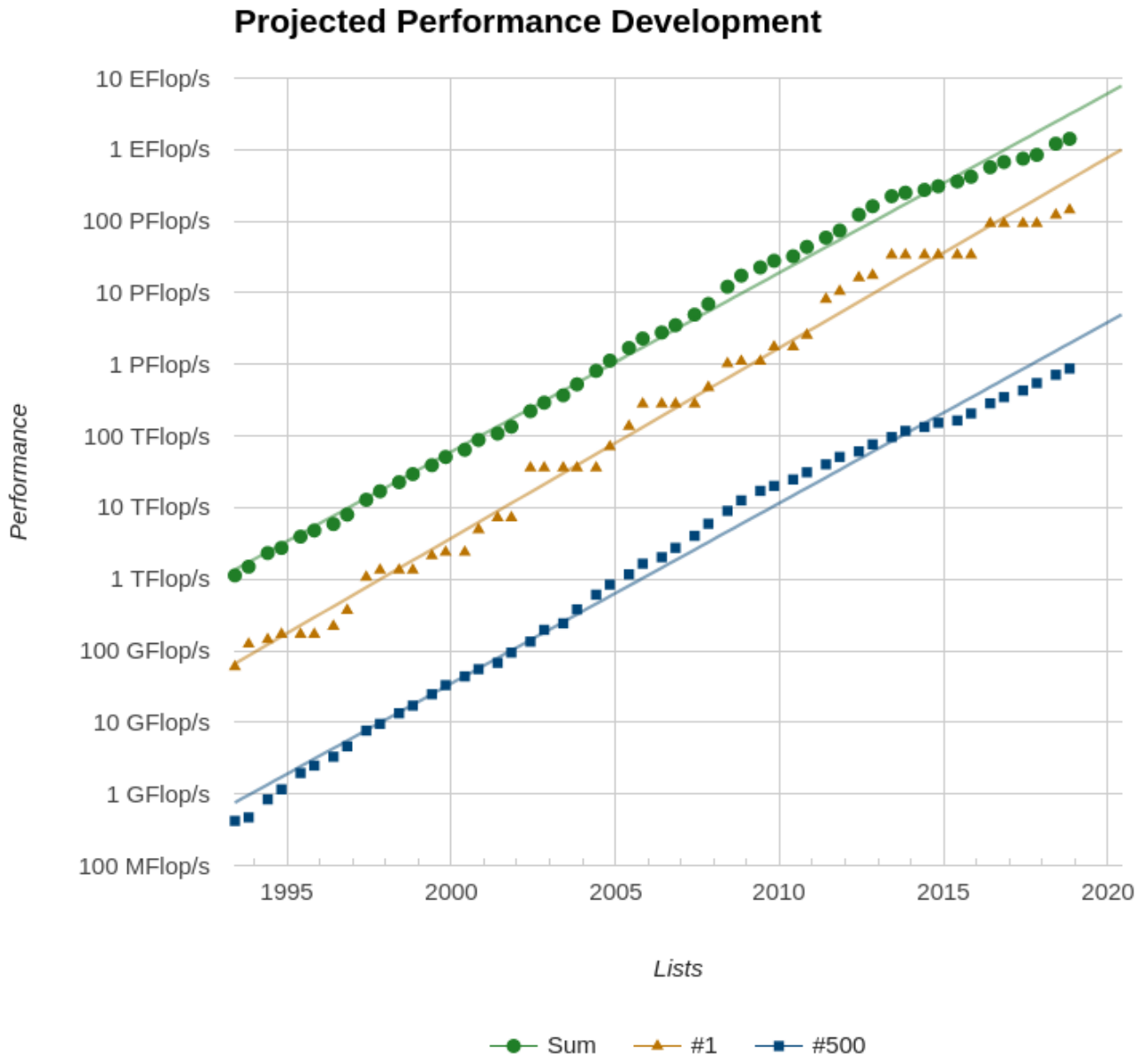

Fun3D - NASA

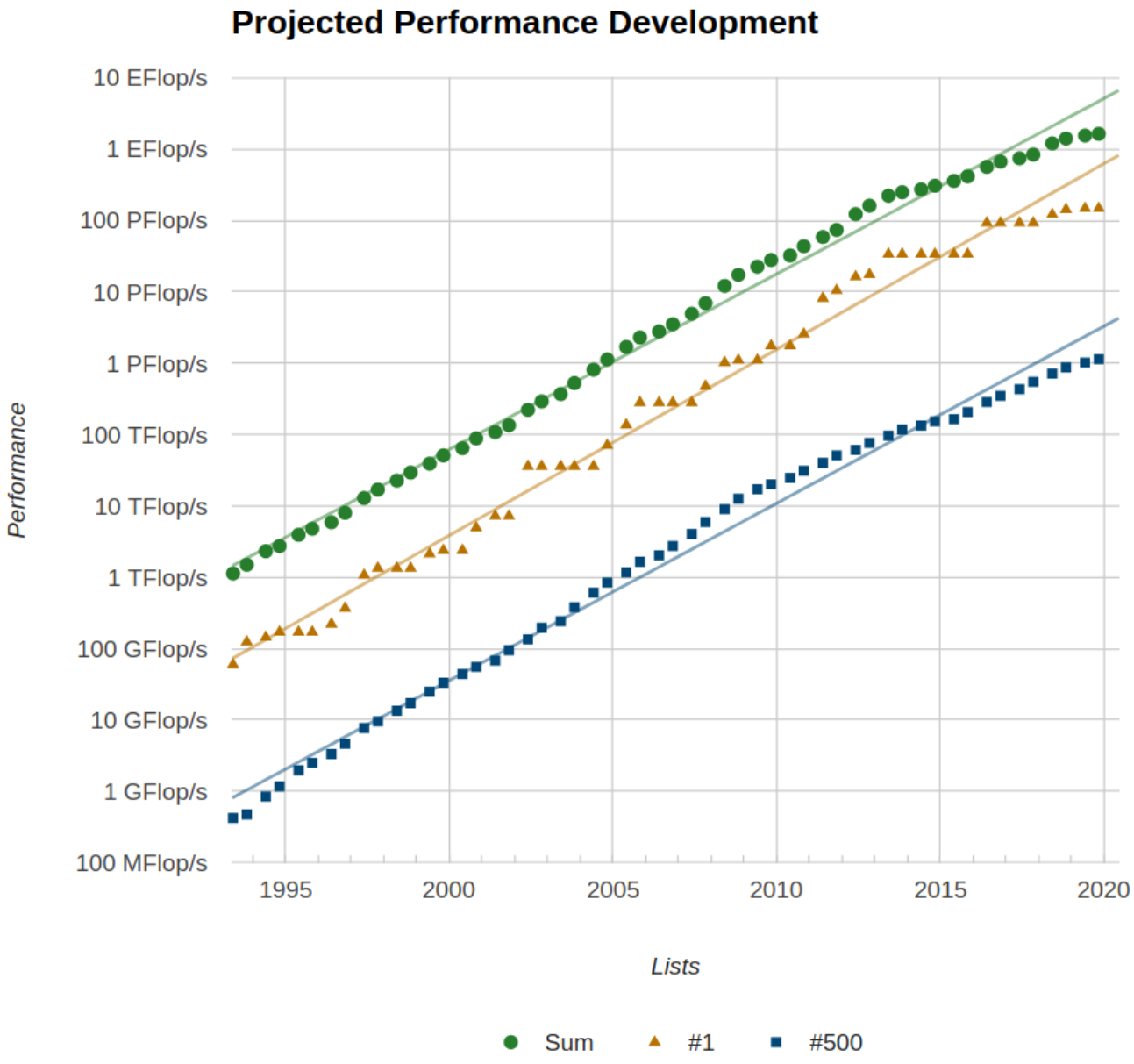
Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	2,414,592	148,600.0	200,794.9	10,096
2	DOE/NNSA/LLNL United States	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM / NVIDIA / Mellanox	1,572,480	94,640.0	125,712.0	7,438
3	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
4	National Super Computer Center in Guangzhou China	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 NUDT	4,981,760	61,444.5	100,678.7	18,482
5	Texas Advanced Computing Center/Univ. of Texas United States	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR Dell EMC	448,448	23,516.4	38,745.9	
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 Cray/HPE	387,872	21,230.0	27,154.3	2,384
7	DOE/NNSA/LANL/SNL United States	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray/HPE	979,072	20,158.7	41,461.2	7,578
8	National Institute of Advanced	**AI Bridging Cloud Infrastructure**	391,680	19,880.0	32,576.6	1,649

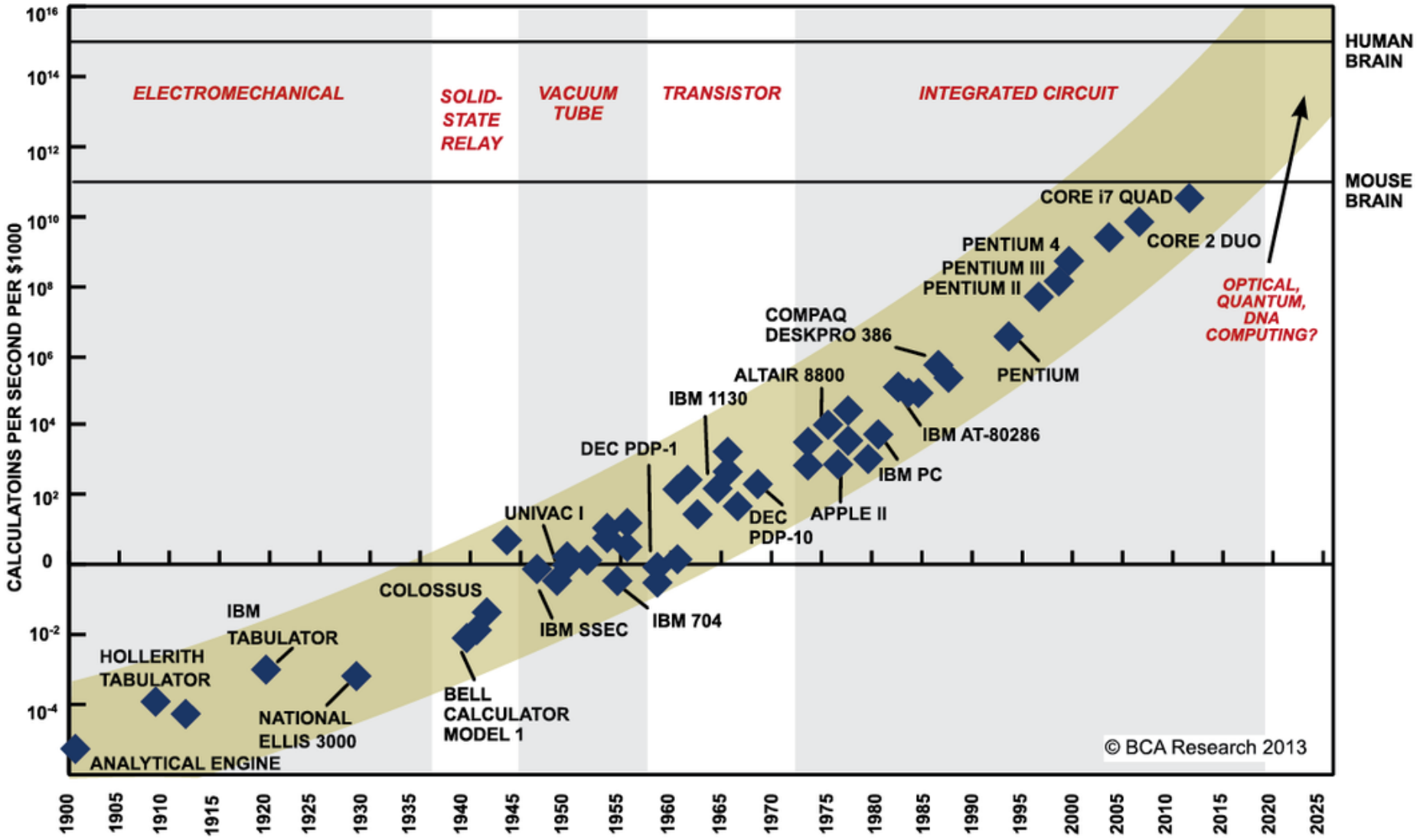


Exascale will be reached close to 2020

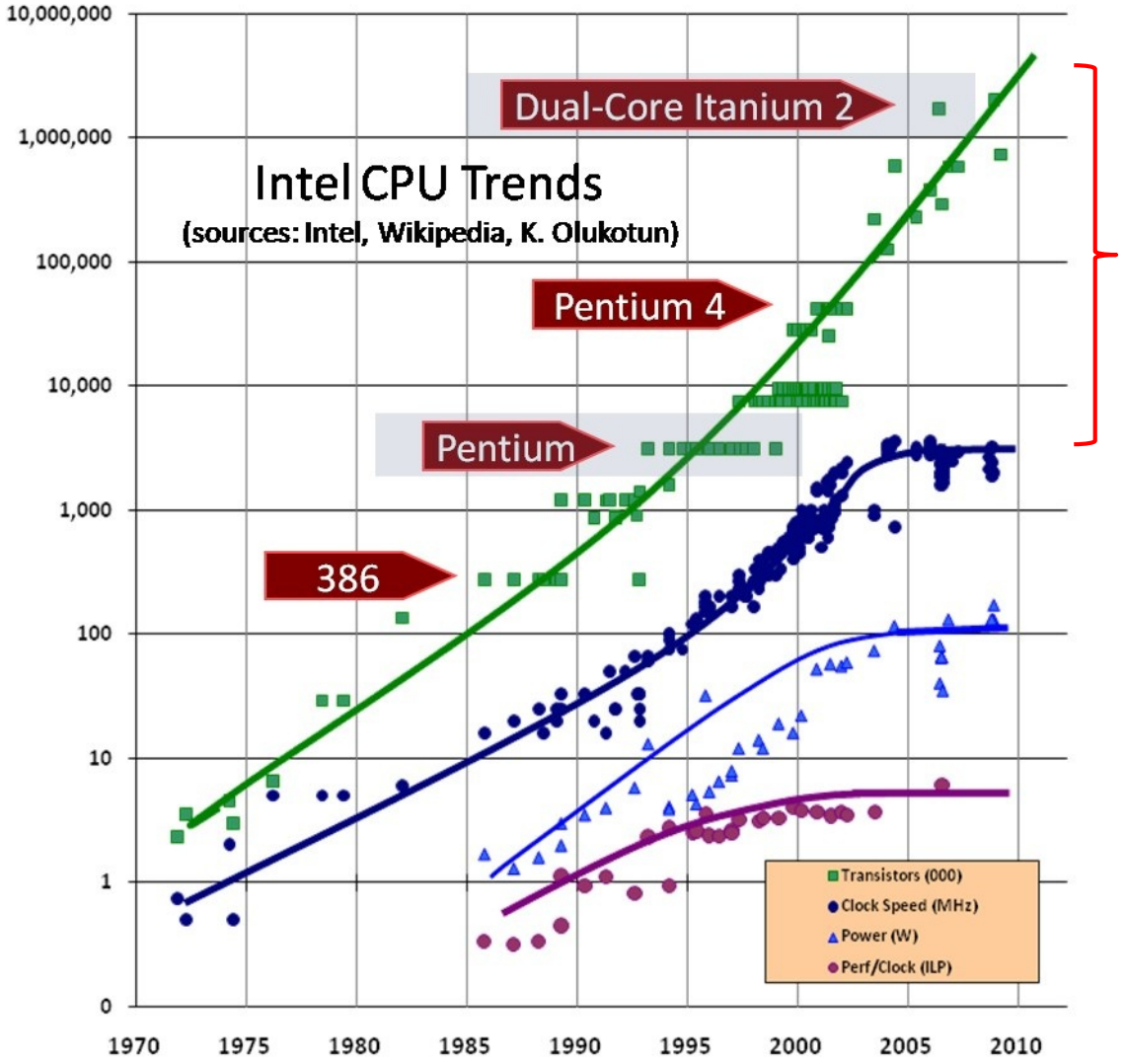
- Aerospace, Airframes, Jet Turbines
- Astrophysics
- Biological and medical systems
- Climate and weather
- Combustion
- Materials science
- Fusion energy
- National security
- Nuclear engineering


Projected Performance Development
10 EFlop/s
1 EFlop/s
100 PFlop/s
10 PFlop/s
1 PFlop/s
100 TFlop/s
10 TFlop/s
1 TFlop/s
100 GFlop/s
10 GFlop/s
1 GFlop/s
100 MFlop/s
Performance
1995
2000
2005
2010
2015
2020
Lists
Sum
#1
#500

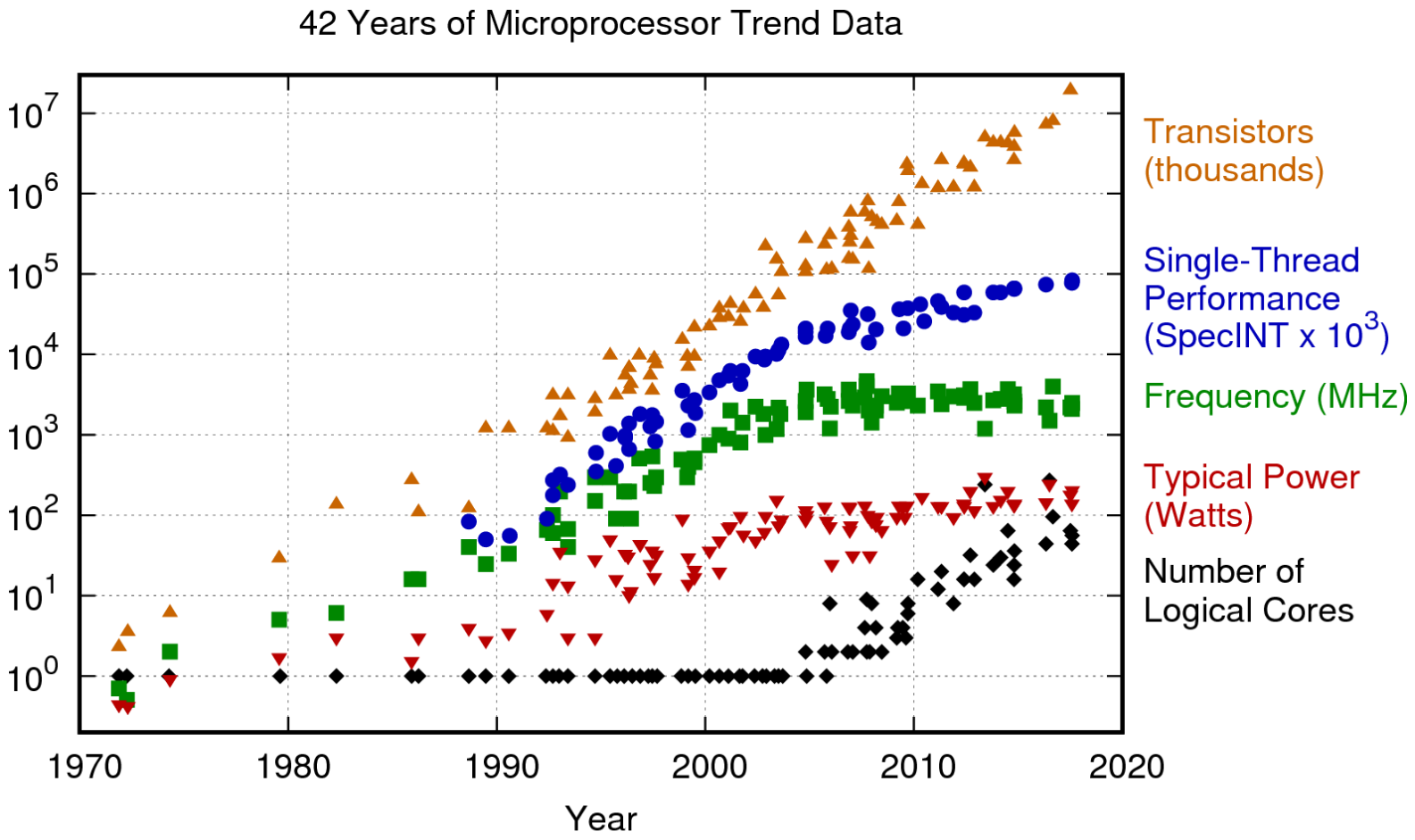

Projected Performance Development
10 EFlop/s
1 EFlop/s
100 PFlop/s
10 PFlop/s
1 PFlop/s
100 TFlop/s
10 TFlop/s
1 TFlop/s
100 GFlop/s
10 GFlop/s
1 GFlop/s
100 MFlop/s
Performance
1995
2000
2005
2010
2015
2020
Lists
Sum
#1
#500



SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, *THE VIKING PRESS*, 2006. DATAPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

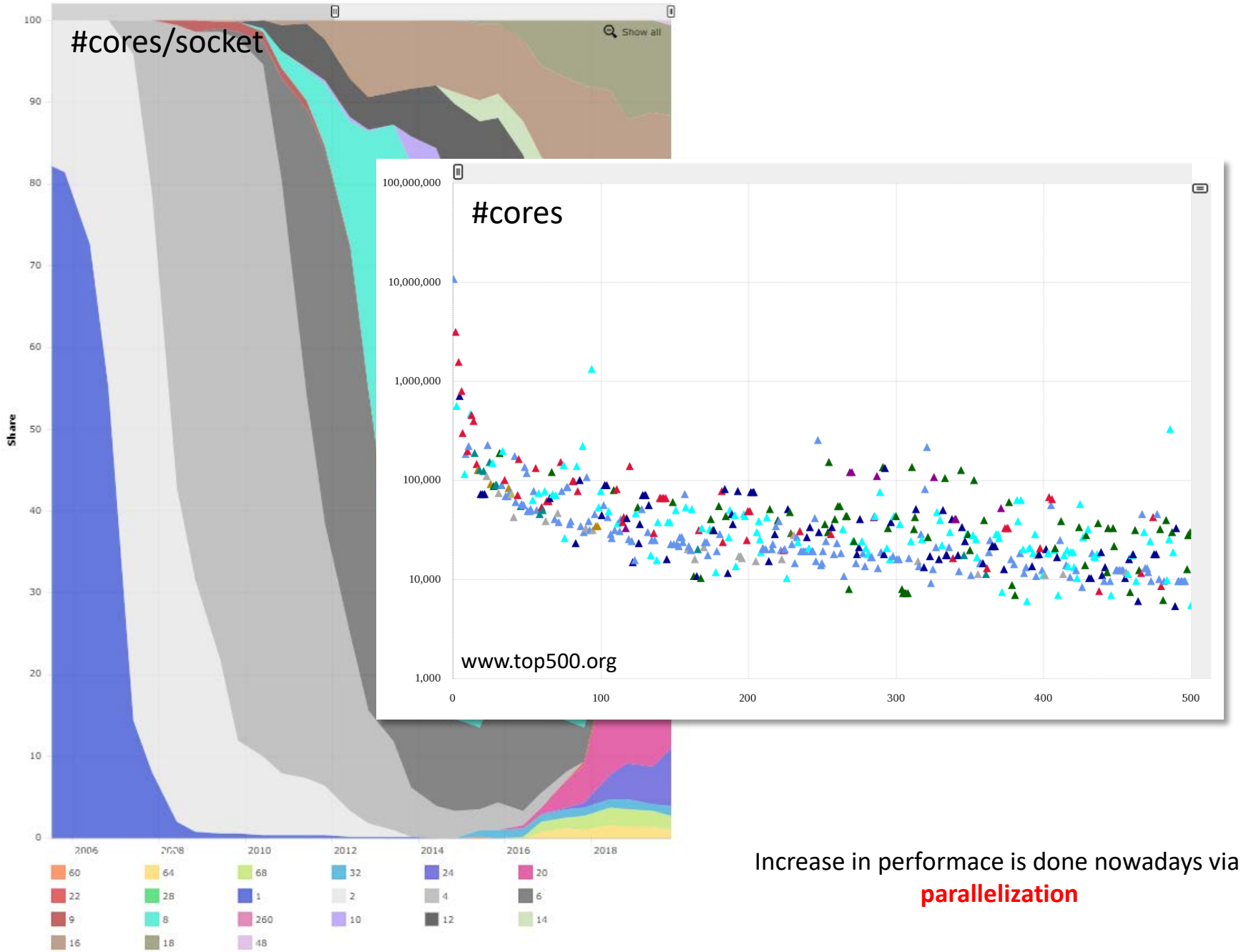


More transistors do not mean faster anymore!



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

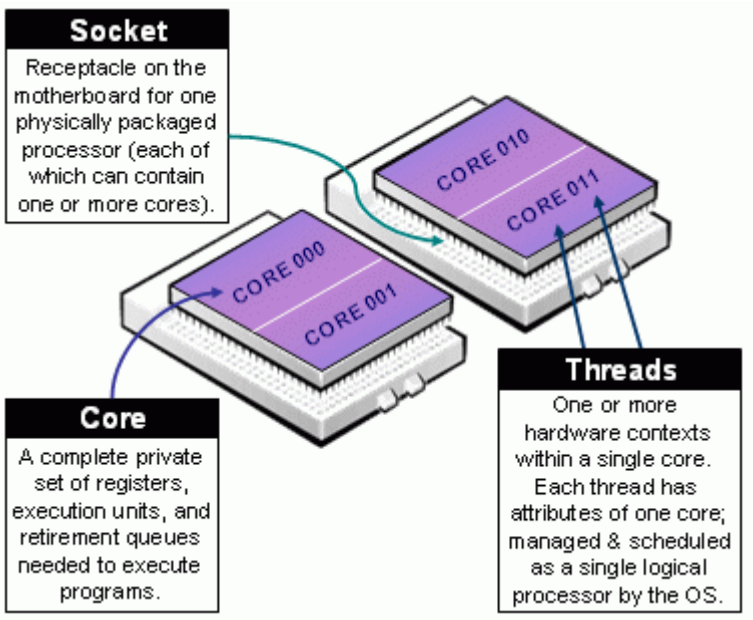
Increase in performace is done nowadays via **parallelization**



Increase in performace is done nowadays via **parallelization**

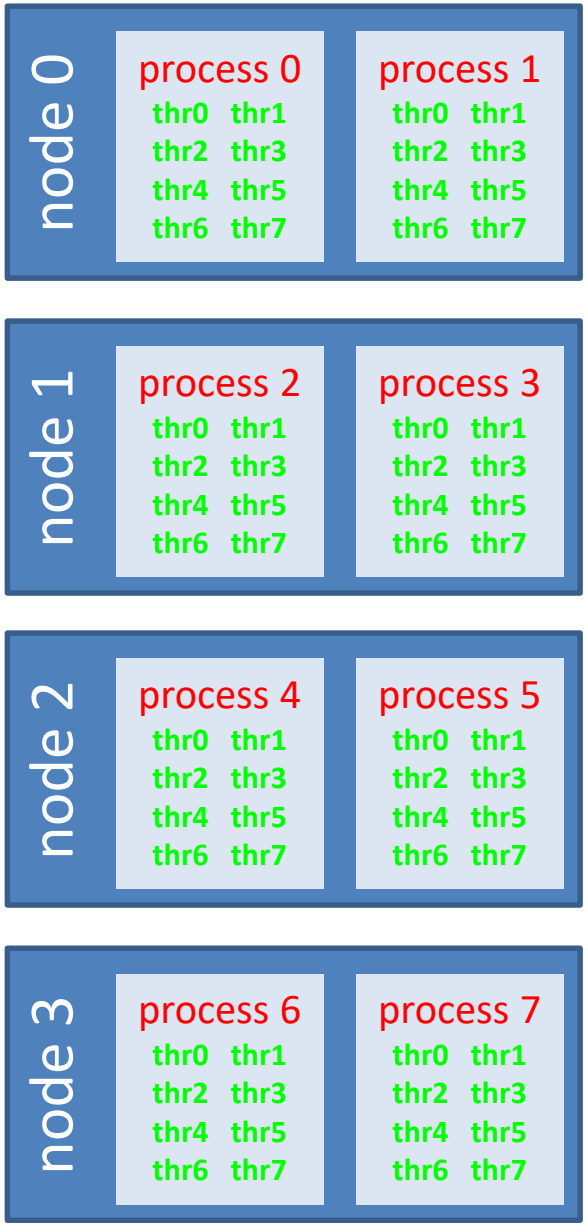
Usually a hybrid MPI+X approach is taken

(X=OpenMP, OpenMP offloading, OpenACC, CUDA)



Hyperthreading is usually deactivated in HPC facilities by default.
Ask the sysadmin!

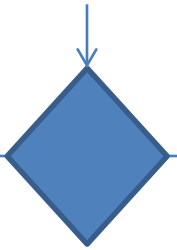
logical cores = # physical cores * # threads/core



In general, the problem can be reduced to a logical repetitive structure:

```
do i = 1, n

<calculations>

end do
```

Are these <calculations> completely **independent** from each other?



Yes

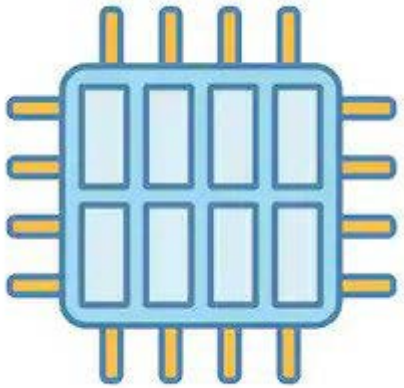
No

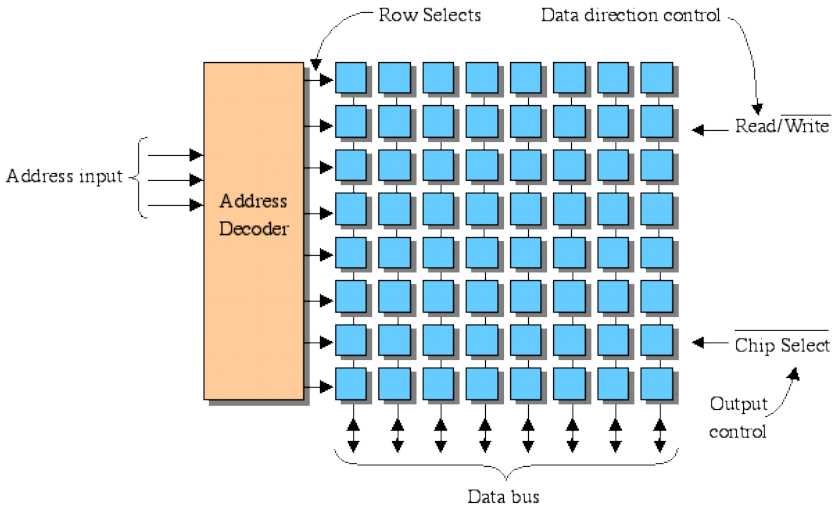
- What's the degree of dependency?
- Can I rewrite/factorize to eliminate the dependency?
- Do variables share contents? Is it necessary?

OpenMP

Open **M**ulti-**P**rocessing

Works in shared memory systems
Can use as many cores as there are in the compute node
Uses threads to split the work
Threads have both private and shared variables
Uses preprocessor directives in commented lines

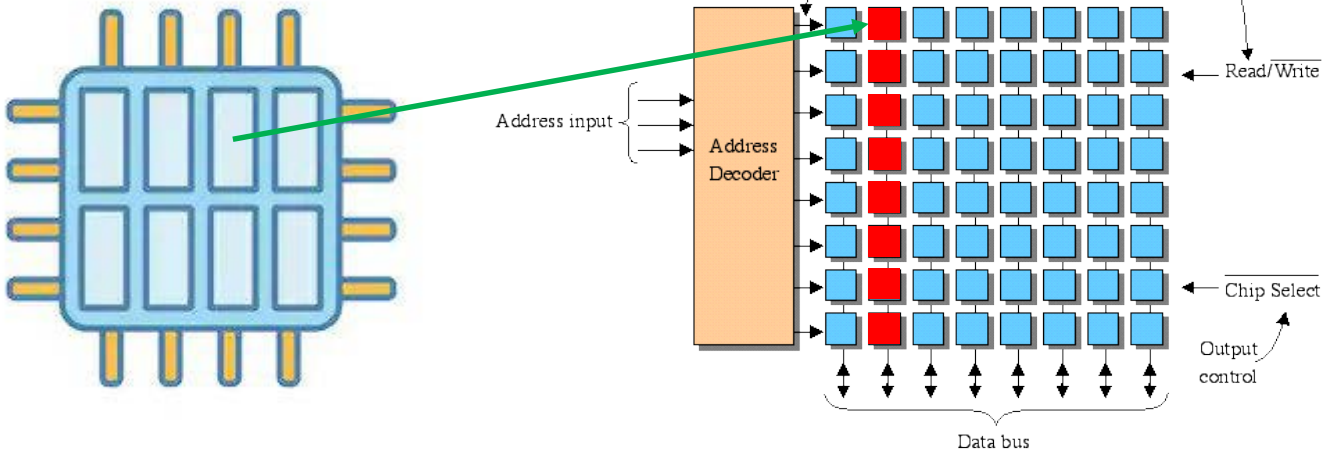




OpenMP

Open **M**ulti-**P**rocessing

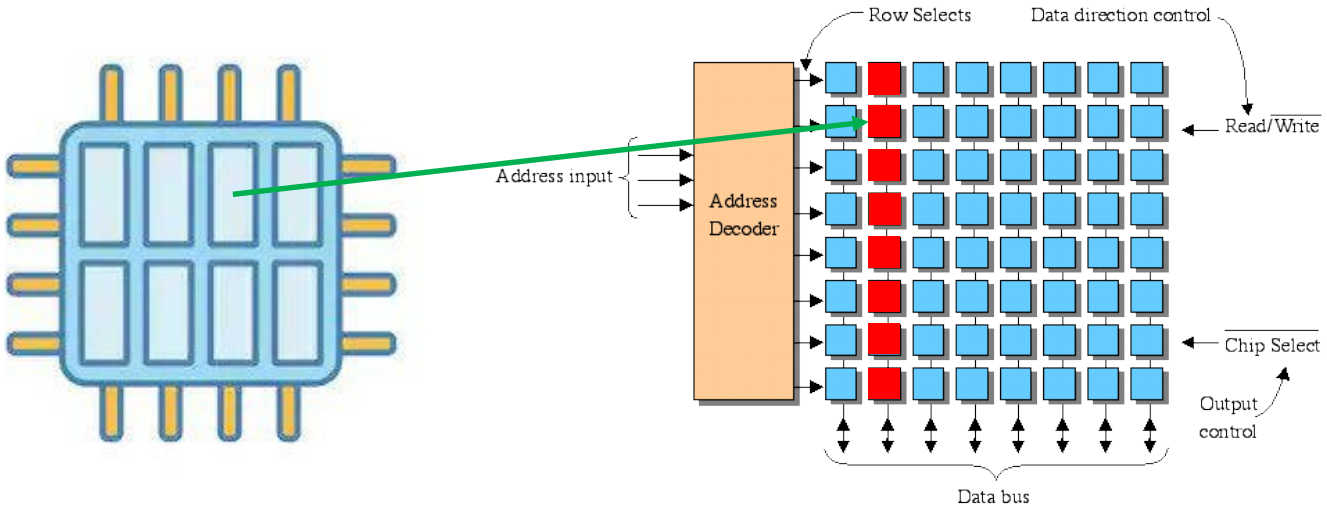
Works in shared memory systems
Can use as many cores as there are in the compute node
Uses threads to split the work
Threads have both private and shared variables
Uses preprocessor directives in commented lines



OpenMP

Open **M**ulti-**P**rocessing

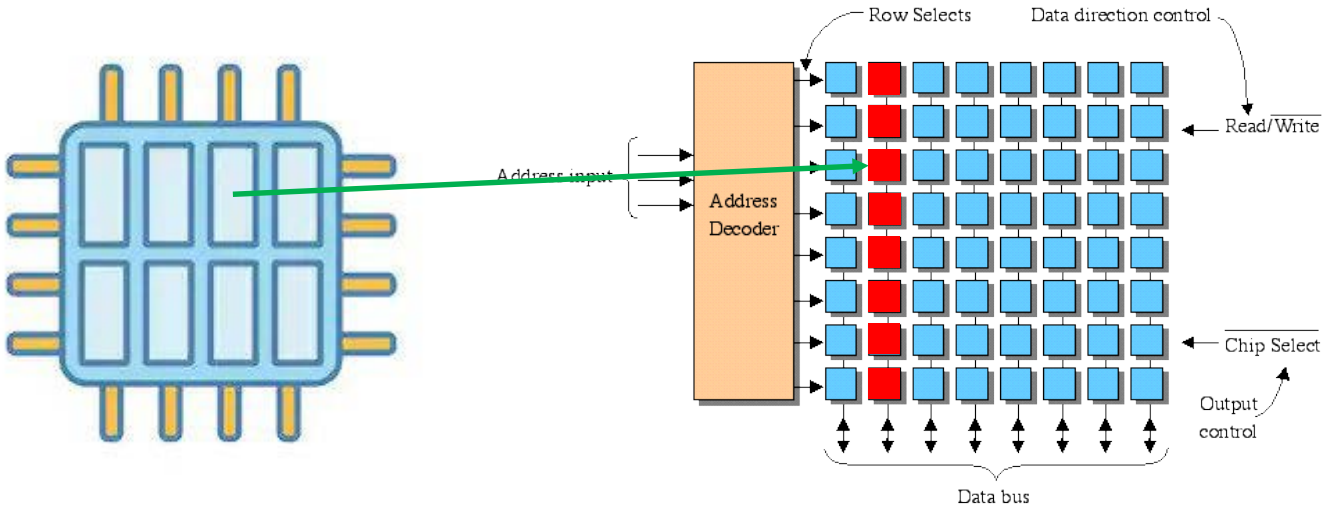
Works in shared memory systems
Can use as many cores as there are in the compute node
Uses threads to split the work
Threads have both private and shared variables
Uses preprocessor directives in commented lines



OpenMP

Open **M**ulti-**P**rocessing

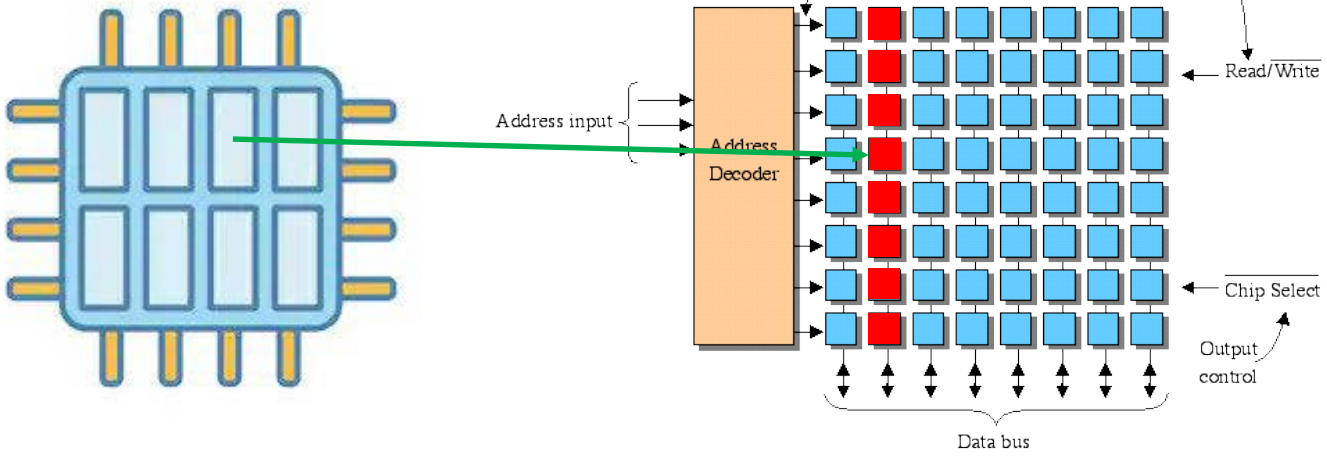
Works in shared memory systems
Can use as many cores as there are in the compute node
Uses threads to split the work
Threads have both private and shared variables
Uses preprocessor directives in commented lines



OpenMP

Open **M**ulti-**P**rocessing

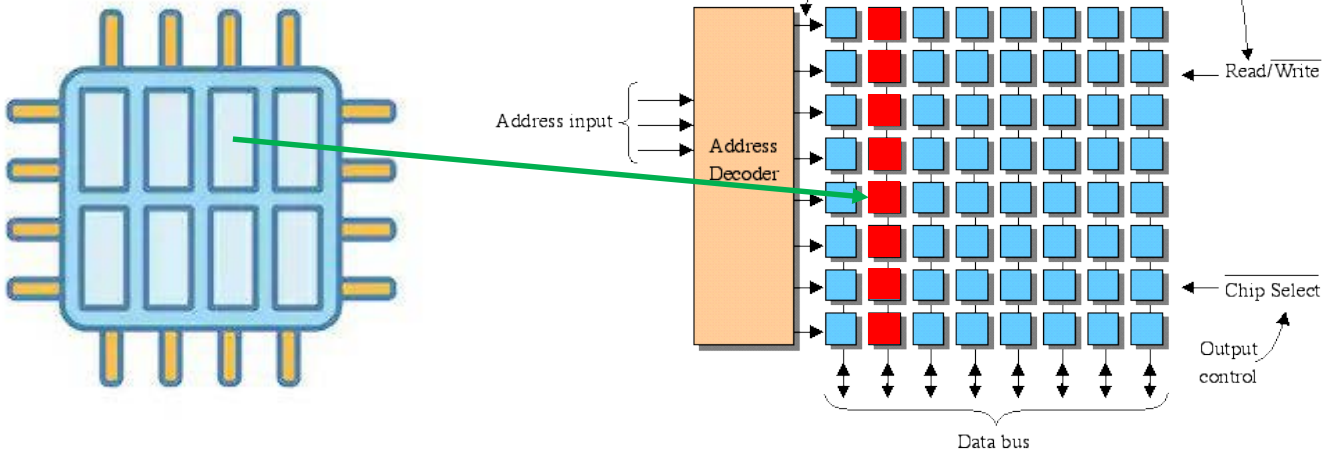
Works in shared memory systems
Can use as many cores as there are in the compute node
Uses threads to split the work
Threads have both private and shared variables
Uses preprocessor directives in commented lines



OpenMP

Open **M**ulti-**P**rocessing

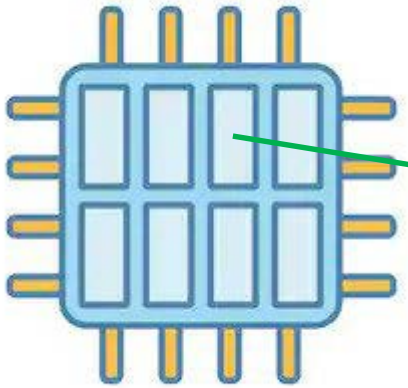
Works in shared memory systems
Can use as many cores as there are in the compute node
Uses threads to split the work
Threads have both private and shared variables
Uses preprocessor directives in commented lines

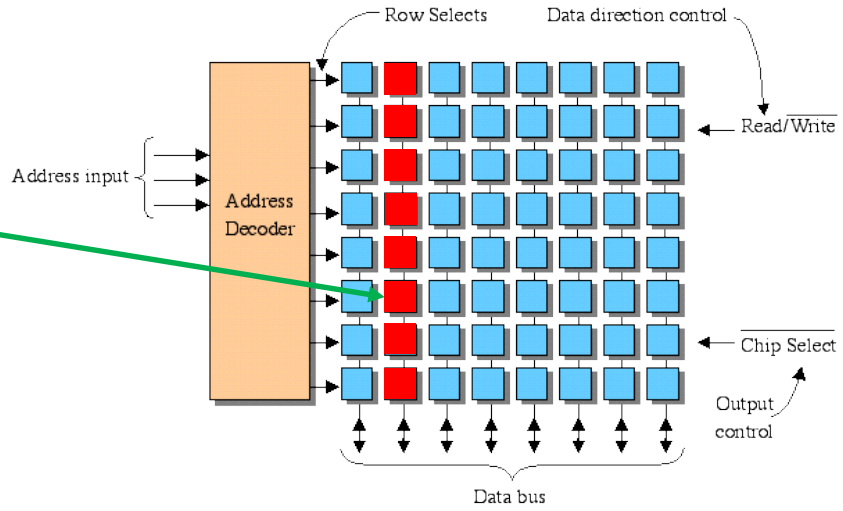


OpenMP

Open **M**ulti-**P**rocessing

Works in shared memory systems
Can use as many cores as there are in the compute node
Uses threads to split the work
Threads have both private and shared variables
Uses preprocessor directives in commented lines

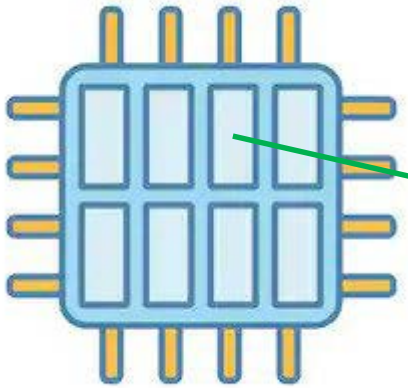


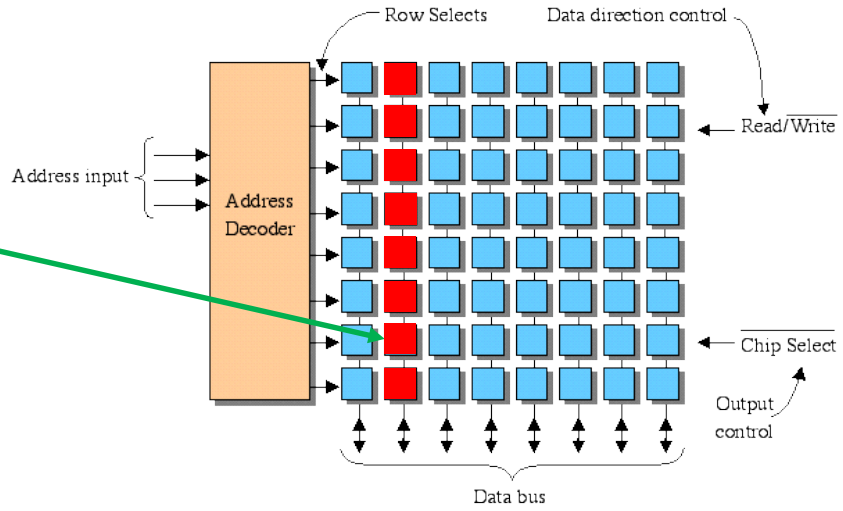


OpenMP

Open **M**ulti-**P**rocessing

Works in shared memory systems
Can use as many cores as there are in the compute node
Uses threads to split the work
Threads have both private and shared variables
Uses preprocessor directives in commented lines

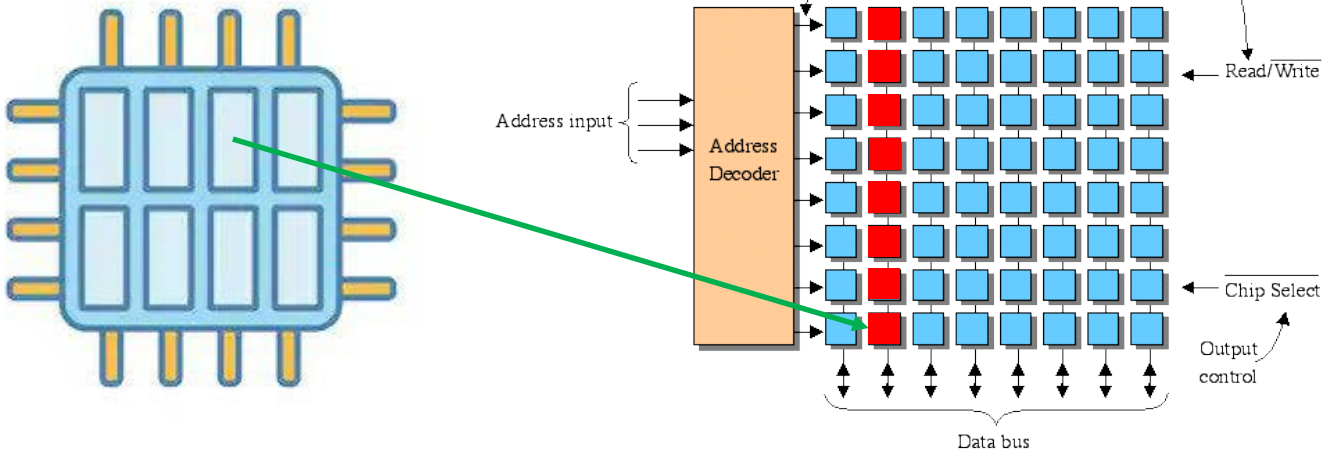




OpenMP

Open **M**ulti-**P**rocessing

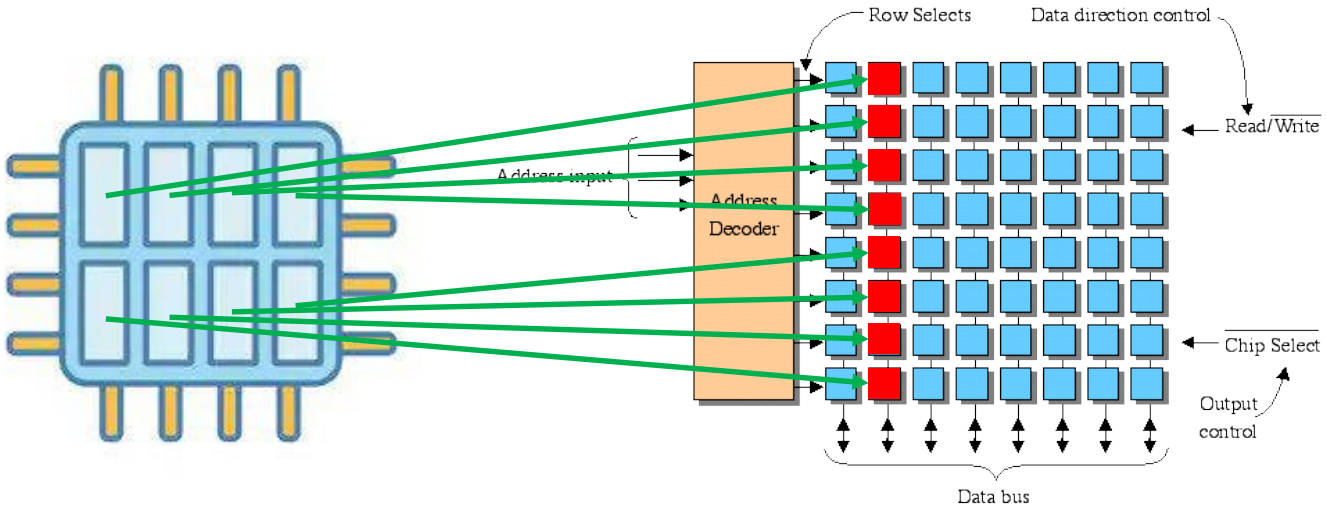
Works in shared memory systems
Can use as many cores as there are in the compute node
Uses threads to split the work
Threads have both private and shared variables
Uses preprocessor directives in commented lines



OpenMP

Open **M**ulti-**P**rocessing

Works in shared memory systems
Can use as many cores as there are in the compute node
Uses threads to split the work
Threads have both private and shared variables
Uses preprocessor directives in commented lines



Some resources

- Main references:
  http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
  https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-111802-web.pdf
  http://www.compunity.org/

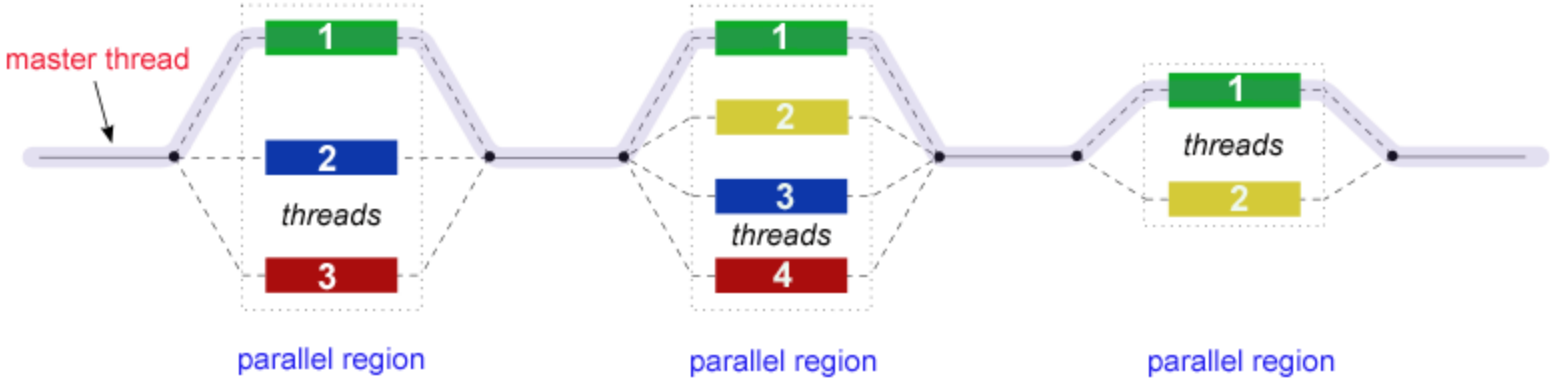
- Examples & tutorials:
  http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.Examples.pdf
  http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf
  https://computing.llnl.gov/tutorials/openMP/
  https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG
  https://numba.pydata.org/numba-doc/latest/user/5minguide.html
  https://mpi4py.readthedocs.io/en/stable/tutorial.html

- Books:
  The OpenMP Common Core: Making OpenMP Simple Again (Mattson, et al. 2019)

- In-person Courses:
  Unibas Master and PhD course, https://hpc.dmi.unibas.ch/HPC/Teaching.html
  In particular: Foundations of Distributed Systems (45402-01/HS): F. Ciorba, H. Schuldt, C. Tschundin
  High-Performance Computing (17164-01/FS): F. Ciorba

  EPFL course, https://moodle.epfl.ch/enrol/index.php?id=13817
  Parallel and High-Performance Computing (MATH-454)

Fork - join Fork - join Fork - join



FORTRAN

```
use omp_lib


!$omp parallel
...
!$omp do
do ...
enddo
!$omp end do
...
!$omp end parallel
```

C/C++

```
#include <omp.h>


#pragma omp parallel
{
  ...
  #pragma omp for
  for(...){
  ...
  }
}
```

- Serial and parallel program share the same source code.
- Serial compiler simply overlooks parallel directives (they are comments).
- Modifying parallel directives cannot break the serial code.
- Modifying the serial code outside of parallel regions cannot break the parallelization.
- Easy to maintain.
- Compact code.

```
!$omp parallel private(id)
id=omp_get_thread_num()
print *,'I am thread: ',id
!$omp end parallel
```

Runtime library rutine.

- omp_get_num_threads()
- omp_set_num_threads()
- omp_get_wtime()
- …

```
I am thread 0
I am thread 1
I am thread 2
I am thread 3
```

```
I am thread 0
I am thread 3
I am thread 2
I am thread 1
```

```
I am thread 2
I am thread 1
I am thread 0
I am thread 3
```

```
I am thread 0
I am thread 1
I am thread 2
I am thread 3
```

We cannot ensure that the threads will execute and finish in order!

```
do i = 1, n

   f(i) = sin(dble(i))

end do
```

f(i) =

sin(1)
sin(2)
sin(3)
sin(4)
sin(5)
sin(6)
sin(7)
sin(8)

```
!$omp parallel
!$omp do
do i = 1, n

   f(i) = sin(dble(i))

end do
!$omp end do
!$omp end parallel
```

f(i) =

sin(1)
sin(2)
sin(3)
sin(4)
sin(5)
sin(6)
sin(7)
sin(8)

```
do i = 1, n

   f(i) = sin(dble(i))

end do
```

```
!$omp parallel
!$omp do
do i = 1, n

   f(i) = sin(dble(i))

end do
!$omp end do
!$omp end parallel
```

f(i) =

sin(1)
sin(2)
sin(3)
sin(4)
sin(5)
sin(6)
sin(7)
sin(8)

Check env. variable, f.e.:
$OMP_NUM_THREADS=4

f(i) =

sin(1)
sin(2)
sin(3)
sin(4)
sin(5)
sin(6)
sin(7)
sin(8)

OpenMP

```
do i = 1, n

   f(i) = sin(dble(i))

end do
```

```
!$omp parallel
!$omp do
do i = 1, n

   f(i) = sin(dble(i))

end do
!$omp end do
!$omp end parallel
```

f(i) =

sin(1)
sin(2)
sin(3)
sin(4)
sin(5)
sin(6)
sin(7)
sin(8)

Check env. variable, f.e.:
$OMP_NUM_THREADS=4

f(i) =

sin(1)
sin(2)
sin(3)
sin(4)
sin(5)
sin(6)
sin(7)
sin(8)

Creates 4 threads and distributes the work evenly

OpenMP

```
do i = 1, n

   f(i) = sin(dble(i))

end do
```

```
!$omp parallel
!$omp do
do i = 1, n

   f(i) = sin(dble(i))

end do
!$omp end do
!$omp end parallel
```

f(i) =

sin(1)
sin(2)
sin(3)
sin(4)
sin(5)
sin(6)
sin(7)
sin(8)

Check env. variable, f.e.:
$OMP_NUM_THREADS=4

f(i) =

sin(1)	thread 0
sin(2)	thread 0
sin(3)	thread 1
sin(4)	thread 1
sin(5)	thread 2
sin(6)	thread 2
sin(7)	thread 3
sin(8)	thread 3

x4 Speed-up!

Creates 4 threads and distributes the work evenly

OpenMP

```
!$omp parallel
!$omp do
do i = 1, n

   f(i) = sin(dble(i))

end do
!$omp end do
!$omp end parallel
```

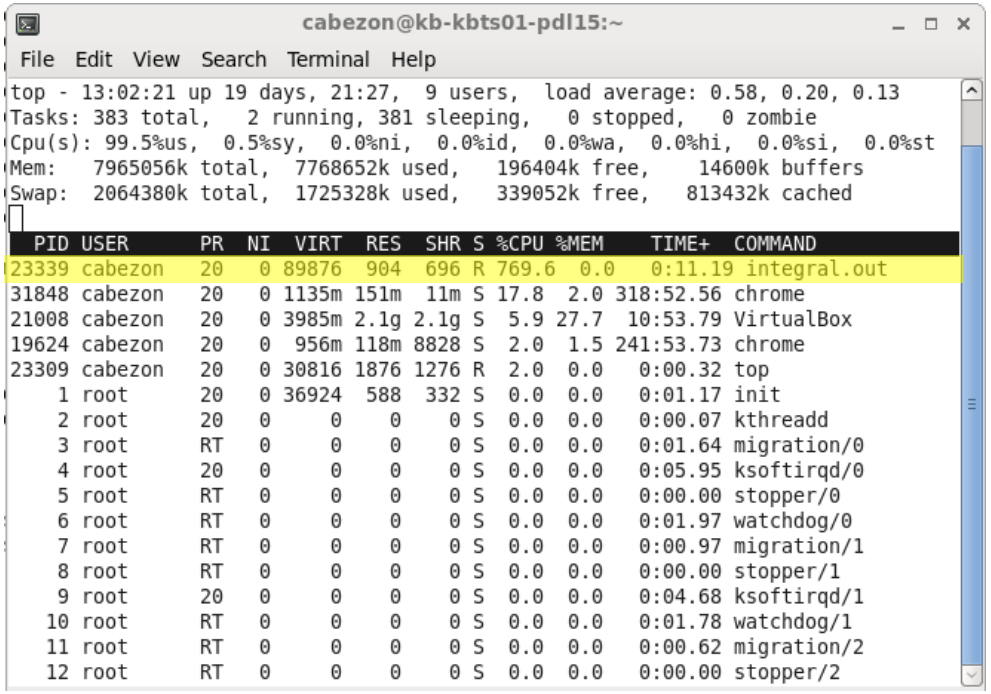
Establish parallel section

Parallel loop evenly divided among the threads

The number of threads is given by the environment variable: $OMP_NUM_THREADS
If not defined, OpenMP assumes OMP_NUM_THREADS = # cores in the computer.
Can be changed with: export OMP_NUM_THREADS = <integer>
(consider adding it to .bashrc)

These are comments!
They are understood by the compiler only when compiled with –openmp option (or similar). Otherwise they are ignored.

An OpenMP calculation will appear as using more than 100% of CPU



OpenMP

```
!$omp parallel
!$omp do schedule(static)
do i = 1, n

   f(i) = sin(dble(i))

end do
!$omp end do
!$omp end parallel
```

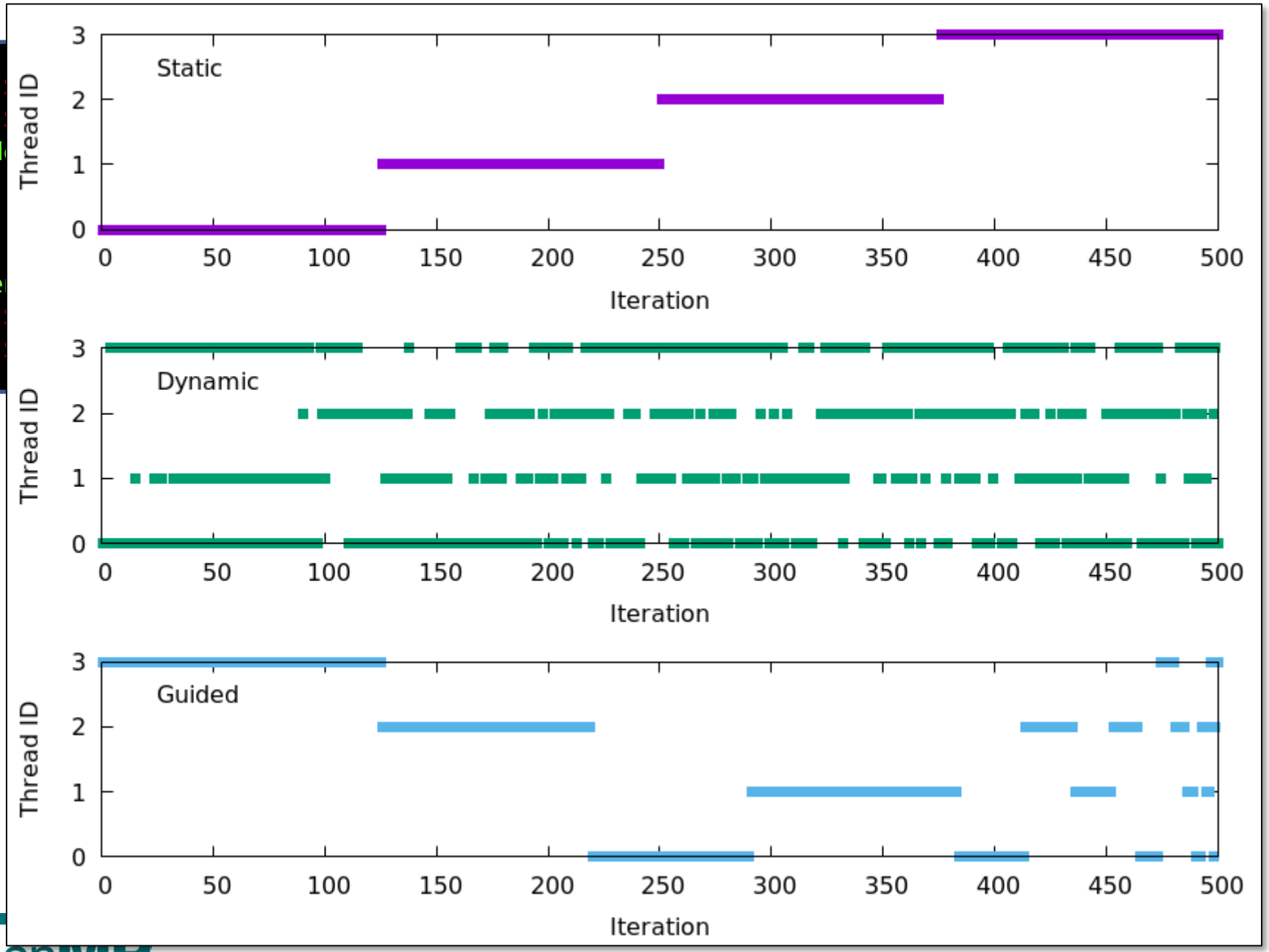
Parallel loop evenly divided among the threads

$$\text{chunk} = \frac{\text{loop count}}{\text{\# threads}}$$

static schedule

Other options are available: `!$omp do schedule(<kind> [,chunk_size])`

Kind	Description
static	Divides the loop in equal-sized chunks (or as equal as possible).
dynamic	When a thread finishes, it retrieves the next chunk from the internal queue. Be aware of the extra overhead! (defaul chunk size = 1).
guided	Similar to dynamic, but starts off large and decreases to better handle imbalace. (defaul chunk size = same as static).
auto	Decision regarding scheduling is delegated to the compiler.
runtime	Uses OMP_SCHDEULE env. Variable to select the schdeluding type.


Static
Thread ID
Iteration
Dynamic
Thread ID
Iteration
Guided
Thread ID
Iteration
0
1
2
3
0
50
100
150
200
250
300
350
400
450
500

```
factorial(1) = 1



do i = 2, 6

   factorial(i) = i * factorial(i-1)

end do
```

factorial(i) =

1
2
6
24
120
720
5040

```
factorial(1) = 1

!$omp parallel
!$omp do schedule(static)
do i = 2, 6

   factorial(i) = i * factorial(i-1)

end do
!$omp end do
!$omp end parallel
```

There is something wrong here!

We cannot ensure this!

factorial(i) =

1
2
6
24
120
720
5040

factorial(i) =

1	
2	thread 0
6	thread 0
24	thread 1
120	thread 1
720	thread 2
5040	thread 2

```
factorial(1) = 1

!$omp parallel
!$omp do schedule(static)
do i = 2, 6

   factorial(i) = i * factorial(i-1)

end do
!$omp end do
!$omp end parallel
```

factorial(i) =

1
2
6
24
120
720
5040

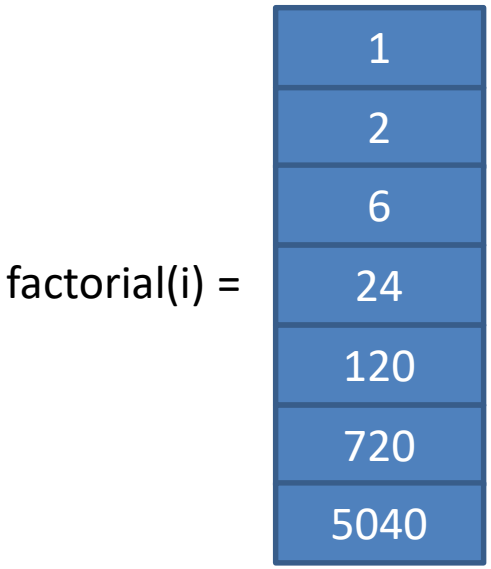
Threads access data in a disordered way

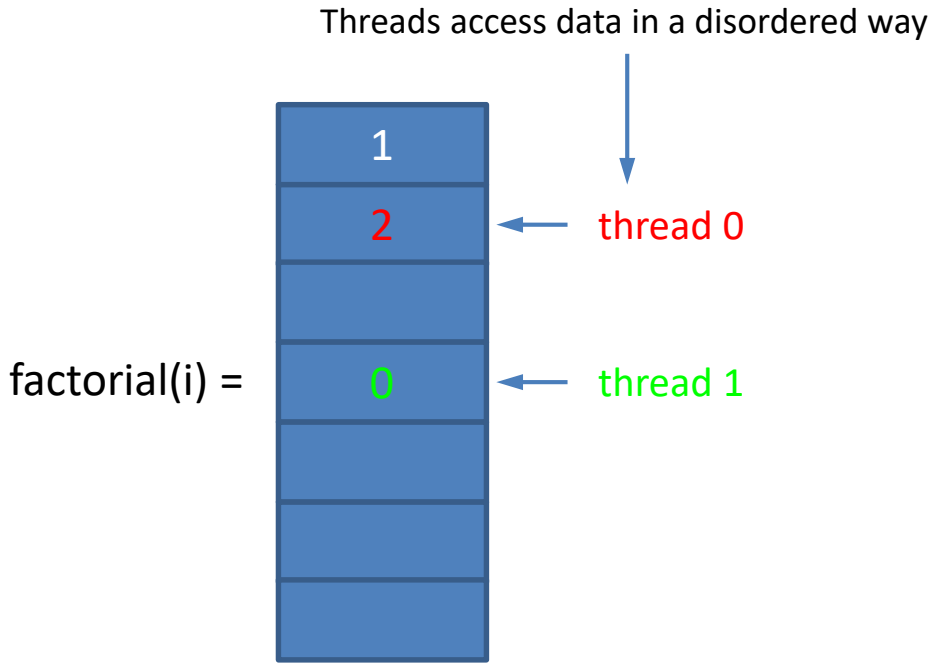
factorial(i) =

1
2

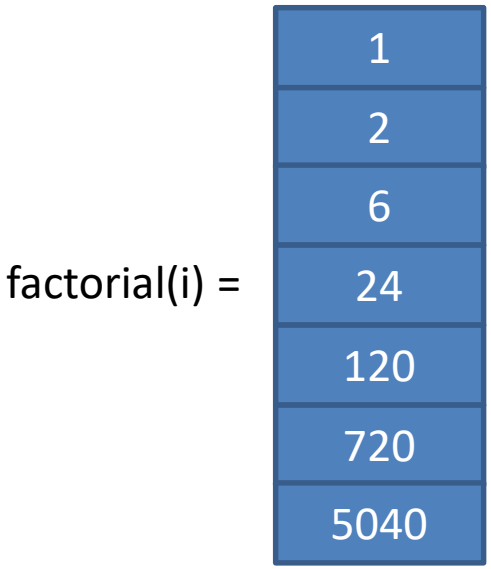
thread 0

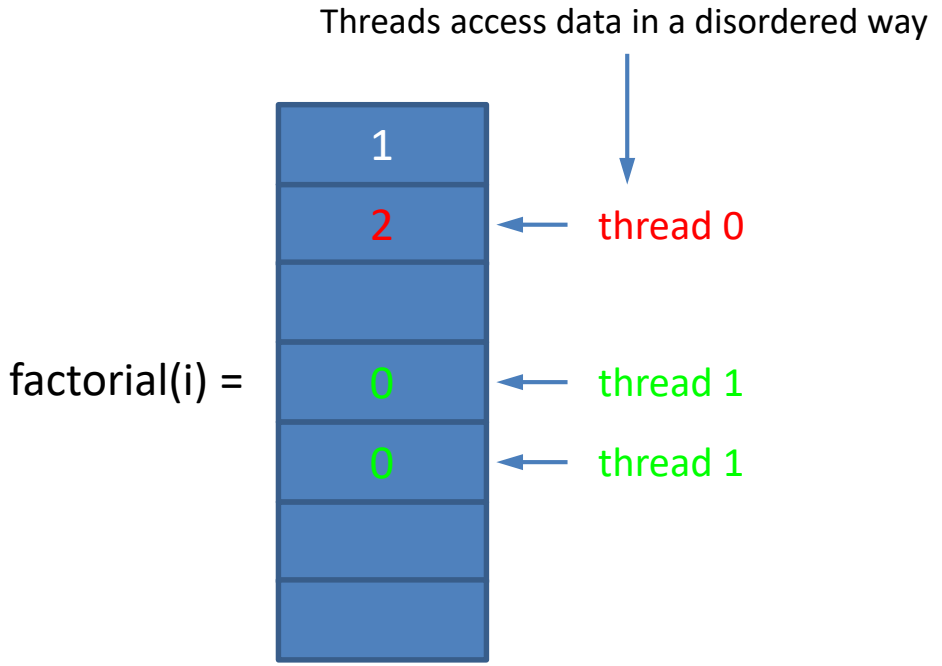
```
factorial(1) = 1

!$omp parallel
!$omp do schedule(static)
do i = 2, 6

   factorial(i) = i * factorial(i-1)

end do
!$omp end do
!$omp end parallel
```





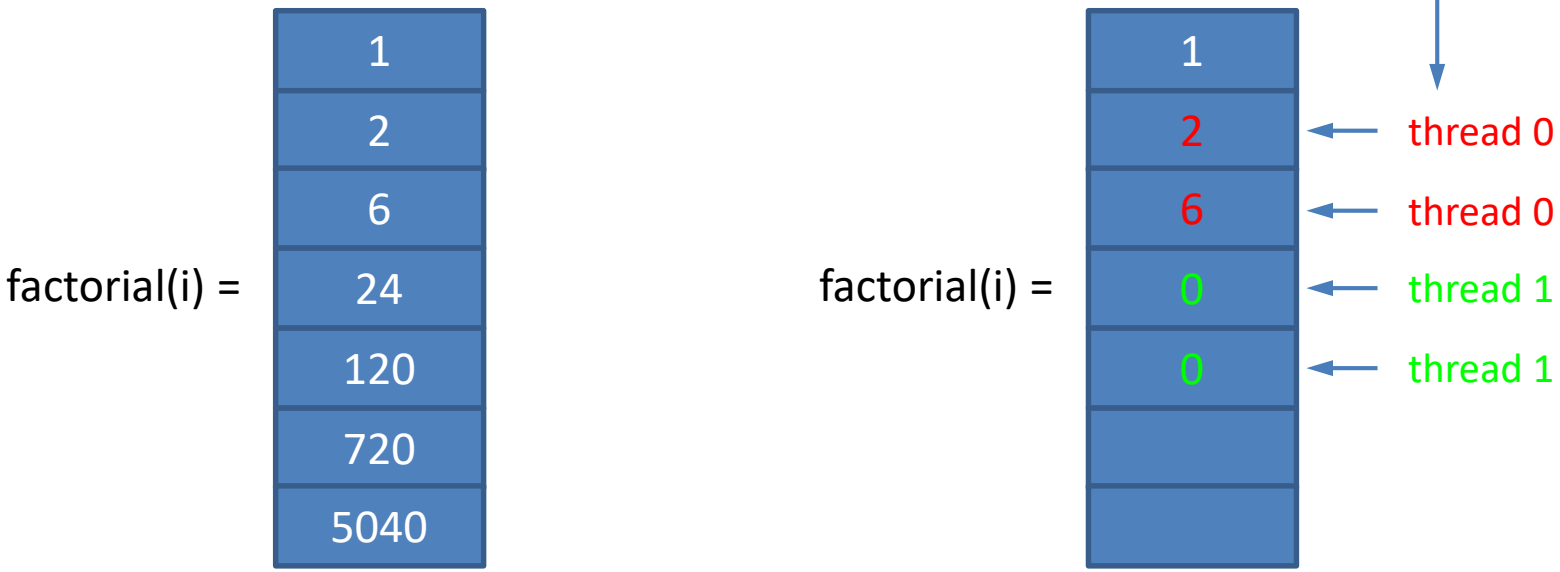
```
factorial(1) = 1

!$omp parallel
!$omp do schedule(static)
do i = 2, 6

   factorial(i) = i * factorial(i-1)

end do
!$omp end do
!$omp end parallel
```





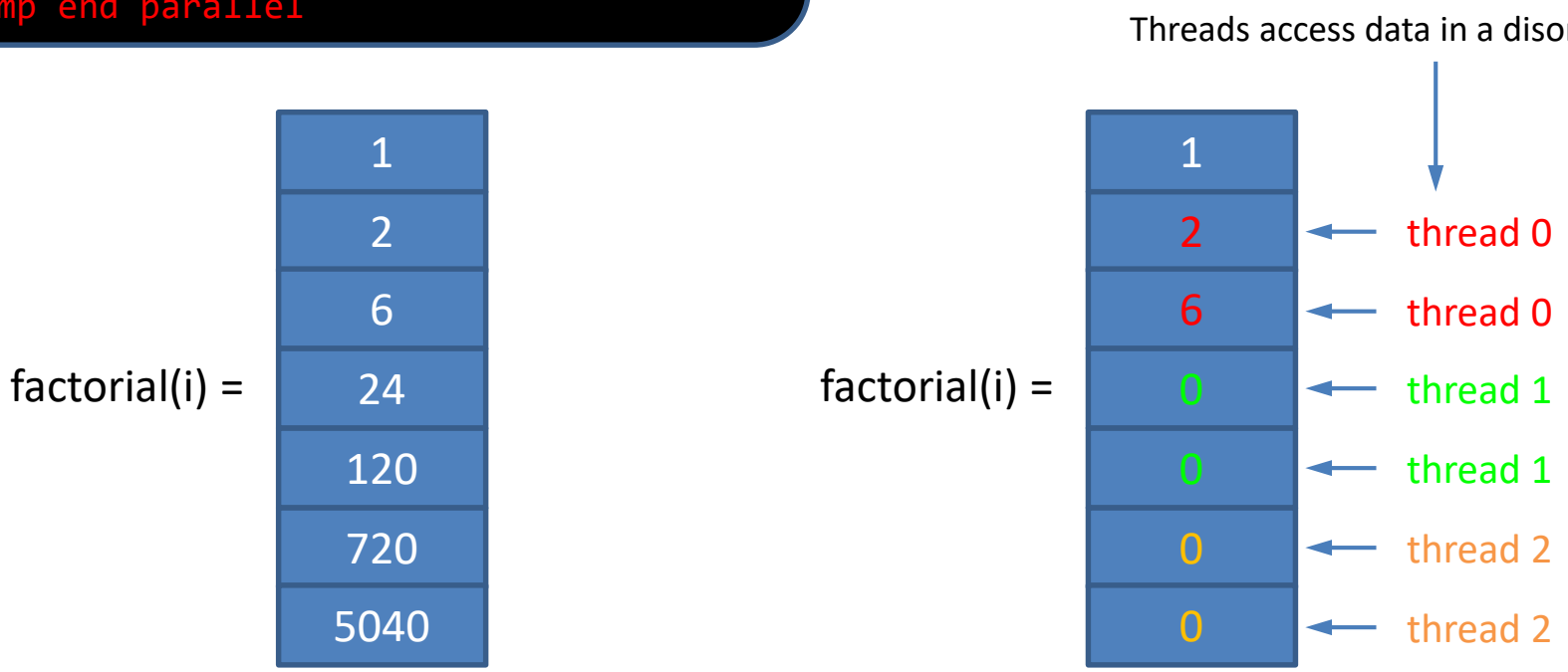
```
factorial(1) = 1

!$omp parallel
!$omp do schedule(static)
do i = 2, 6

   factorial(i) = i * factorial(i-1)

end do
!$omp end do
!$omp end parallel
```

Threads access data in a disordered way



```
factorial(1) = 1

!$omp parallel
!$omp do schedule(static)
do i = 2, 6

   factorial(i) = i * factorial(i-1)

end do
!$omp end do
!$omp end parallel
```

Threads access data in a disordered way



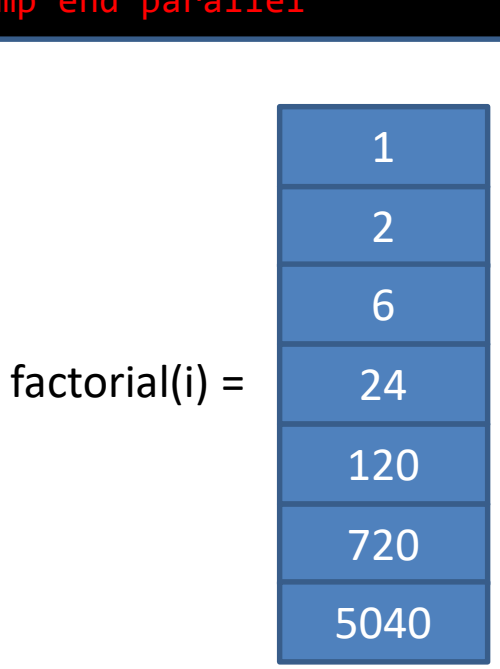
OpenMP

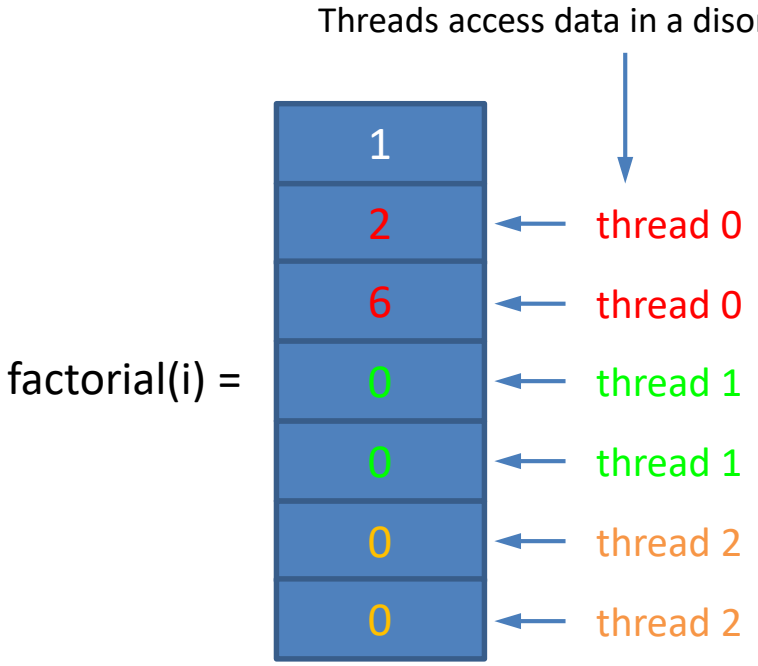
```
factorial(1) = 1

!$omp parallel
!$omp do schedule(static)
do i = 2, 6

   factorial(i) = i * factorial(i-1)

end do
!$omp end do
!$omp end parallel
```

This is called **Race Condition**

It can even be worse if factorial was not properly initialized!

Difficult to detect because for a given case, system or run, the threads may win the race in an order that happens to make the program run correctly.





```
!$omp parallel
!$omp do schedule(static)
do i = 1, 6

   x = i
   y(i) = x * 2

end do
!$omp end do
!$omp end parallel
```

Another example of **Race Condition**

y(i) =

2
4
6
8
10
12

y(i) =

x	y(i)
1	

← thread 0

```
!$omp parallel
!$omp do schedule(static)
do i = 1, 6

   x = i
   y(i) = x * 2

end do
!$omp end do
!$omp end parallel
```

Another example of **Race Condition**

y(i) =

2
4
6
8
10
12

y(i) =

x	y(i)	
1	2	← thread 0

```
!$omp parallel
!$omp do schedule(static)
do i = 1, 6

   x = i
   y(i) = x * 2

end do
!$omp end do
!$omp end parallel
```

Another example of **Race Condition**

y(i) =

2
4
6
8
10
12

y(i) =

x	y(i)	
1	2	← thread 0
2		← thread 0

```
!$omp parallel
!$omp do schedule(static)
do i = 1, 6

   x = i
   y(i) = x * 2

end do
!$omp end do
!$omp end parallel
```

Another example of **Race Condition**

y(i) =

2
4
6
8
10
12

y(i) =

x	y(i)	
1	2	← thread 0
2		← thread 0
5		← thread 2

```
!$omp parallel
!$omp do schedule(static)
do i = 1, 6

   x = i
   y(i) = x * 2

end do
!$omp end do
!$omp end parallel
```

Another example of **Race Condition**

y(i) =

2
4
6
8
10
12

y(i) =

x	y(i)	
1	2	← thread 0
2	10	← thread 0
5		← thread 2

```
!$omp parallel
!$omp do schedule(static)
do i = 1, 6

   x = i
   y(i) = x * 2

end do
!$omp end do
!$omp end parallel
```

Another example of **Race Condition**

y(i) =

2
4
6
8
10
12

y(i) =

x	y(i)	
1	2	← thread 0
2	10	← thread 0
3		← thread 1
5		← thread 2

```
!$omp parallel
!$omp do schedule(static)
do i = 1, 6

   x = i
   y(i) = x * 2

end do
!$omp end do
!$omp end parallel
```

Another example of **Race Condition**

y(i) =

2
4
6
8
10
12

y(i) =

x	y(i)	
1	2	← thread 0
2	10	← thread 0
3		← thread 1
5	6	← thread 2

```
!$omp parallel
!$omp do schedule(static)
do i = 1, 6

   x = i
   y(i) = x * 2

end do
!$omp end do
!$omp end parallel
```

Another example of **Race Condition**

y(i) =

2
4
6
8
10
12

y(i) =

x	y(i)	
1	2	← thread 0
2	10	← thread 0
3	6	← thread 1
5	6	← thread 2

```
!$omp parallel
!$omp do schedule(static)
do i = 1, 6

   x = i
   y(i) = x * 2

end do
!$omp end do
!$omp end parallel
```

Another example of **Race Condition**

y(i) =

2
4
6
8
10
12

y(i) =

x	y(i)	
1	2	← thread 0
2	10	← thread 0
3	6	← thread 1
4		← thread 1
5	6	← thread 2

```
!$omp parallel
!$omp do schedule(static)
do i = 1, 6

   x = i
   y(i) = x * 2

end do
!$omp end do
!$omp end parallel
```

Another example of **Race Condition**

y(i) =

2
4
6
8
10
12

y(i) =

x	y(i)	
1	2	← thread 0
2	10	← thread 0
3	6	← thread 1
4		← thread 1
5	6	← thread 2
6		← thread 2

```
!$omp parallel
!$omp do schedule(static)
do i = 1, 6

   x = i
   y(i) = x * 2

end do
!$omp end do
!$omp end parallel
```

Another example of **Race Condition**

To solve this we have the **private** clause.

y(i) =

2
4
6
8
10
12

y(i) =

x	y(i)	
1	2	← thread 0
2	10	← thread 0
3	6	← thread 1
4	12	← thread 1
5	6	← thread 2
6	12	← thread 2

```
!$omp parallel private (x)
!$omp do schedule(static)
do i = 1, 6

   x = i
   y(i) = x * 2

end do
!$omp end do
!$omp end parallel
```

Another example of **Race Condition**

To solve this we have the **private** clause.

With this, each thread has its private copy of x and the problem is solved.

This comes at the cost of an overhead in memory.

y(i) =

2
4
6
8
10
12

y(i) =

x_0	x_1	x_2	y(i)	
1			2	← thread 0
2			4	← thread 0
	3		6	← thread 1
	4		8	← thread 1
		5	10	← thread 2
		6	12	← thread 2

OpenMP

By default everything is shared inside the OMP section, but loop indexes, which are private.
Nevertheless, I recommend to explicitely declare private everything that it has to be private.

Can you declare all private variables of the following code?

NOTE: jumpx and w1d are outputs of the corresponding subroutine calls.

```
!$omp parallel private (???)
!$omp do schedule(static)
do i = ini, end
   ii = 1+dim*(i-1)
   do k = 1, nvi(i)
      j = neighbors(i,k)
      call apply_PBC(i,k,0,jumpx)
      d1 = a(ii) – a(jj) – jumpx
      d05 = sqrt(d1)
      v1 = d05/h(i)
      call Wkernel(v1,w1d)
      dter = pk(i)*w1d
      sumwh(i) = sumwh(i) + xmass(j) * dter
      if (equ) then
         dlw1d = log10(w1d)/indice(i)
      end if
   end do
   do it = 1, nut
      do ie = 1, ne
         f(ie, it) = kfactor * alpha( ie, it, i)
         ftot(ie, it, i) = f(ie, it) / dens(i)
      end do
    end do
end do
!$omp end do
!$omp end parallel
```

Can you declare all private variables of the following code?

NOTE: jumpx and w1d are outputs of the corresponding subroutine calls.

```
!$omp parallel private (REDACTED)
!$omp do schedule(static)
do i = ini, end
   ii = 1+dim*(i-1)
   do k = 1, nvi(i)
      j = neighbors(i,k)
      call apply_PBC(i,k,0,jumpx)
      d1 = a(ii) - a(jj) - jumpx
      d05 = sqrt(d1)
      v1 = d05/h(i)
      call Wkernel(v1,w1d)
      dter = pk(i)*w1d
      sumwh(i) = sumwh(i) + xmass(j) * dter
      if (equ) then
         dlw1d = log10(w1d)/indice(i)
      end if
   end do
   do it = 1, nut
      do ie = 1, ne
         f(ie, it) = kfactor * alpha( ie, it, i)
         ftot(ie, it, i) = f(ie, it) / dens(i)
      end do
    end do
end do
!$omp end do
!$omp end parallel
```

As a rule of thumb private terms (either variables or arrays) are always:

- on the left of assignments
- outputs of subroutines

Otherwise, it is very likely that they have to be shared.

Checking this requires practice!

If you declare private a variable that should not, you are still messing it up! This variable will very likely be used with a wrong value on it, depending on what was stored in memory.

A piece of advise: use default(none)

OpenMP has a set of default rules about data sharing:

1. Variables declared outside the parallel region are shared.
2. Loop indices inside the parallel region are private.
3. Local variables declared within the parallel region are private.

You don't need to remember this.

Just declare everthing!

With more practice you can skip declaring shared variables.

```
!$omp parallel default(none) private(REDACTED) &
!$omp        & shared(REDACTED)
!$omp do schedule(static)
do i = ini, end
   ii = 1+dim*(i-1)
   do k = 1, nvi(i)
      j = neighbors(i,k)
      call apply_PBC(i,k,0,jumpx)
      d1 = a(ii) - a(jj) - jumpx
      d05 = sqrt(d1)
      v1 = d05/h(i)
      call Wkernel(v1,w1d)
      dter = pk(i)*w1d
      sumwh(i) = sumwh(i) + xmass(j) * dter
      if (equ) then
         dlw1d = log10(w1d)/indice(i)
      end if
   end do
   do it = 1, nut
      do ie = 1, ne
         f(ie, it) = kfactor * alpha( ie, it, i)
         ftot(ie, it, i) = f(ie, it) / dens(i)
      end do
    end do
end do
!$omp end do
!$omp end parallel
```

A note regarding private variables: They are not initialized!

```
result = 20

!$omp parallel private(result)
result = result + 10
print *, ‘Thread ‘,omp_get_thread_num(), result
!$omp end parallel

print *, result
```

To control the initialization of private variables we have **firstprivate**

```
Thread 3 10
Thread 0 10
Thread 1 71
Thread 2 10

20
```

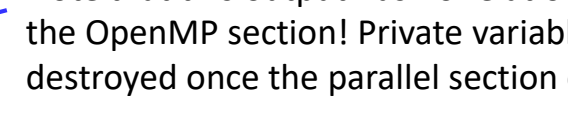
If you are lucky, result may access a section of the memory that is empty. Otherwise, any value can be used to initialize result.

A note regarding private variables: They are not initialized!

```
result = 20

!$omp parallel firstprivate(result)
result = result + 10
print *, ‘Thread ‘,omp_get_thread_num(), result
!$omp end parallel

print *, result
```

To control the initialization of private variables we have **firstprivate**

```
Thread 3 30
Thread 0 30
Thread 1 30
Thread 2 30

20
```



Note that this output has no relation with the OpenMP section! Private variables are destroyed once the parallel section ends.

Consider now this code. Is it correct?

```
sum = 0

!$omp parallel
!$omp do
do i = 1, 6

   sum = sum + i

end do
!$omp end do
!$omp end parallel
```

The variable sum should be private to give the correct answer, but it should be also shared to be accessed by all threads!

Consider now this code. Is it correct?

```
sum = 0

!$omp parallel
!$omp do reduction(+: sum)
do i = 1, 6

   sum = sum + i

end do
!$omp end do
!$omp end parallel
```

The variable sum should be private to give the correct answer, but it should be also shared to be accessed by all threads!

We can solve this with reduction

This allows each thread to have a private copy of sum where they store their partial calculation and then, when the threads exit, it groups all partial values from all threads using the defined operation (+ in this case) in one global variable.

Reduction variables have to meet the following requirements:

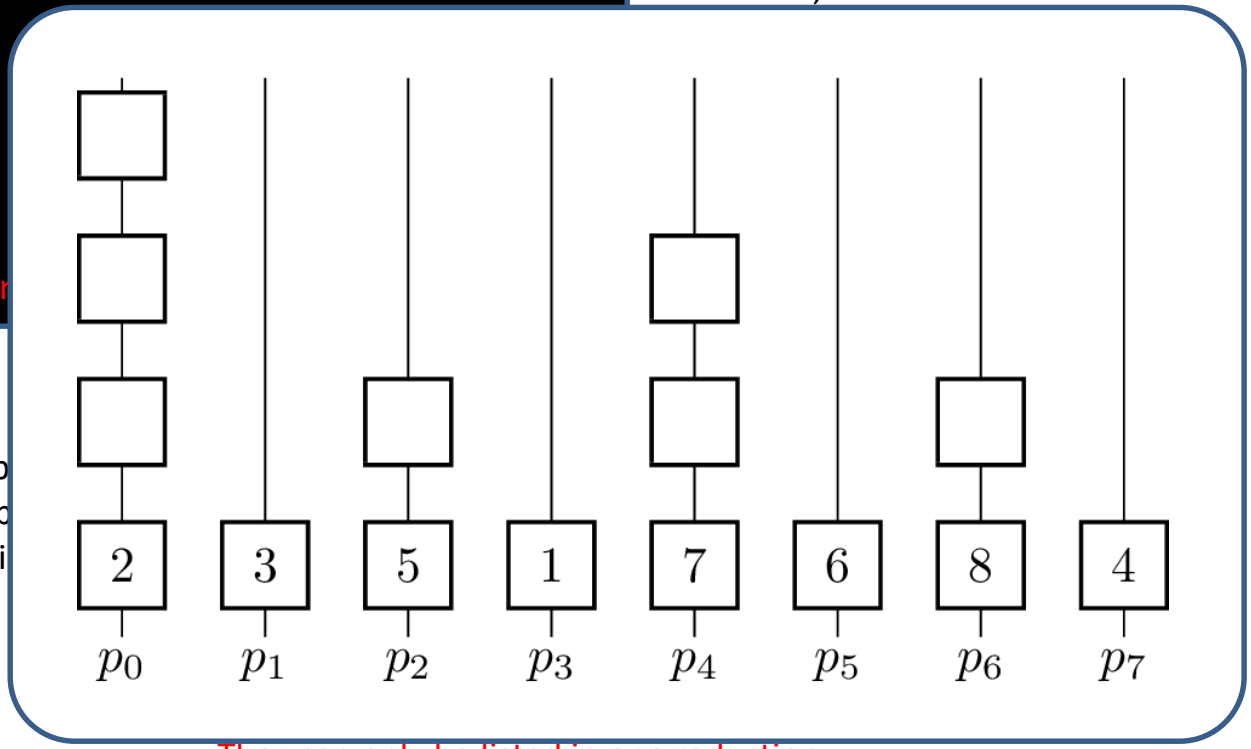
- They can only be listed in one reduction
- Cannot be declared constant
- Cannot be declared private in the parallel contruct

Consider now this code. Is it correct?

```
sum = 0

!$omp parallel
!$omp do reduction(+: sum)
do i = 1, 6

   sum = sum

end do
!$omp end do
!$omp end par
```

The variable sum should be private to give the correct answer, but it should be also shared to be accessed by

This allows each
they store their p
exit, it groups all p
operation (+ in thi



Source: wikipedia

- They can only be listed in one reduction
- Cannot be declared constant
- Cannot be declared private in the parallel contruct

Connecting to the cloud cluster:

Open a new local terminal (Linux, Mac) or MobaXterm (Windows)

Connect to the cluster:

ssh –Y <username>@<cluster_name>

You should have your username via private message in the chat.

The password is: <password>

Download the tar ball with the exercises again, now in the cluster:

wget --trust-server-names https://bit.ly/2KzMEJ4

tar xvf openmp_course.tar.gz

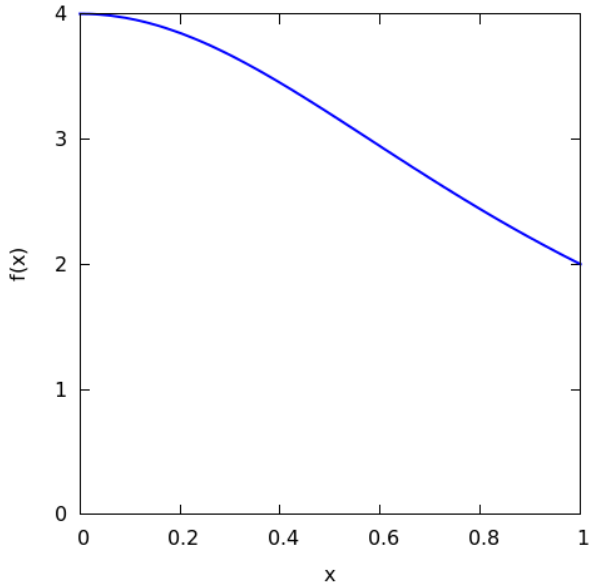
Enter in openmp_course/:

cd openmp_course

Exercise: Numerical integration

We know that:

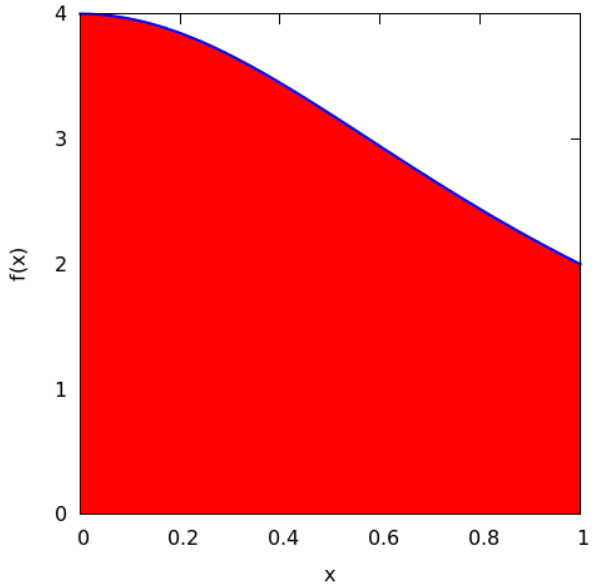
$$\int_0^1 \frac{4}{(1+x^2)} dx = \pi$$



Exercise: Numerical integration

We know that:

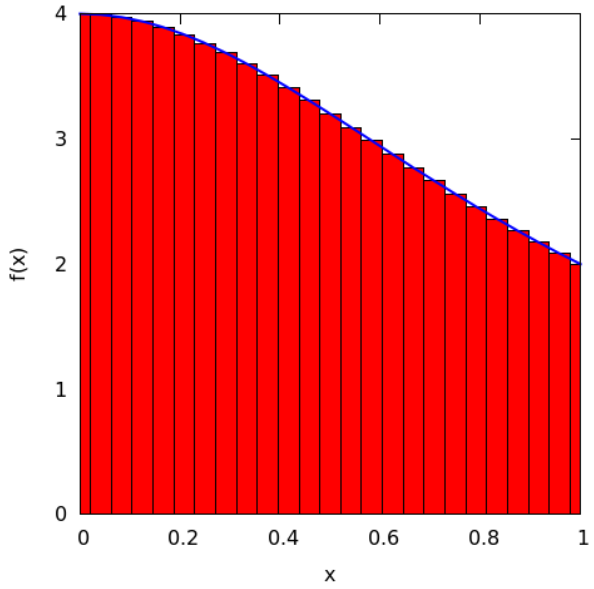
$$\int_0^1 \frac{4}{(1+x^2)} dx = \pi$$



Exercise: Numerical integration

We know that:

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



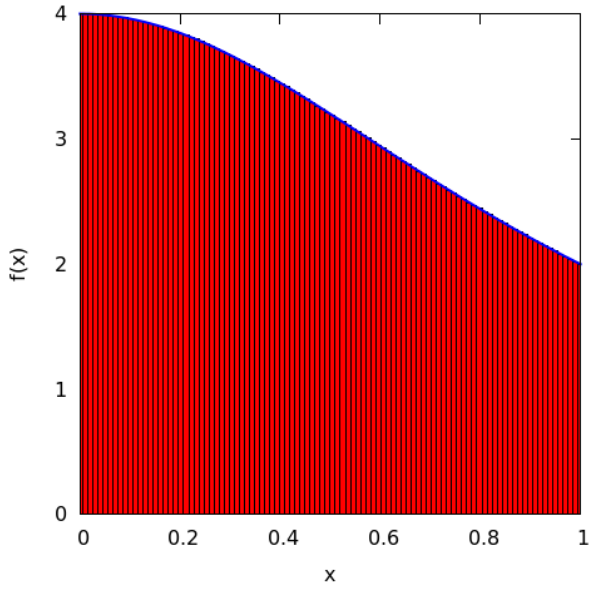
Using the composite formula we can approximate the integral as:

$$\int_a^b f(x)dx \approx \left(\sum_{i=0}^{n} f(x_i)\right)\frac{b-a}{n}$$

Exercise: Numerical integration

We know that:

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



Using the composite formula we can approximate the integral as:

$$\int_a^b f(x)dx \approx \left(\sum_{i=0}^{n} f(x_i)\right)\frac{b-a}{n}$$

Exercise: Numerical integration

Parallelize the following program with openMP

```
steps = 1000000000
dx = 1./dble(steps)
sum = 0.
do i = 1, steps
   x = (dble(i)-0.5)*dx
   sum = sum + 4./(1. + x*x)
end do
sum = sum * dx
```

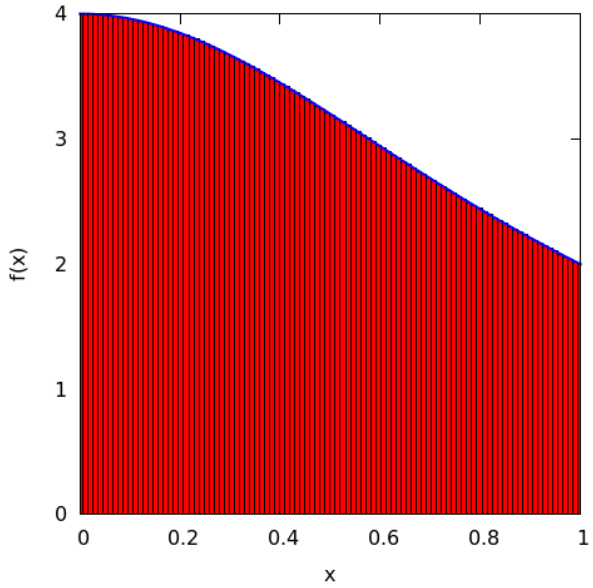
Compile with: **gfortran –fopenmp pi.f90**

Fix the number of threads: **export OMP_NUM_THREADS=4**

Execute with: **time ./a.out**

We know that:

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



Using the composite formula we can approximate the integral as:

$$\int_a^b f(x)dx \approx \left(\sum_{i=0}^{n} f(x_i)\right)\frac{b-a}{n}$$

Exercise: Numerical integration

Parallelize the following program with openMP

```
steps = 1000000000
dx = 1./dble(steps)
sum = 0.
do i = 1, steps
   x = (dble(i)-0.5)*dx
   sum = sum + 4./(1. + x*x)
end do
sum = sum * dx
```

We know that:

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

Compile with: **gfortran –fopenmp**

Fix the number of threads: **exp**

Execute with: **time ./a.o**

Using the composite formula we can approximate the integral as:

$$\int_a^b f(x)dx \approx \left(\sum_{i=0}^{n} f(x_i)\right)\frac{b-a}{n}$$

What happens if we have nested loops?

```
do i = 1, ni
   do j = 1, nj
      do k = 1, nk
         <calculations>
      end do
   end do
end do
```

What happens if we have nested loops?

Not much… we simply parallelize the outer loop.

```
!$omp parallel private(i,j,k)
!$omp do
do i = 1, ni
   do j = 1, nj
      do k = 1, nk
         <calculations>
      end do
   end do
end do
!$omp end do
!$omp end parallel
```

But this depends on the number of iterations for each loop.

What happens if we have nested loops?

Not much… we simply parallelize the outer loop.

```
!$omp parallel private(i,j,k)
!$omp do
do i = 1, 4
   do j = 1, 20
      do k = 1, 10000
         <calculations>
      end do
   end do
end do
!$omp end do
!$omp end parallel
```

But this depends on the number of iterations for each loop.

Having such small count of iterations in the outer loop is not efficient. We might even have less iterations than threads!

What happens if we have nested loops?

Not much… we simply parallelize the outer loop.

```
!$omp parallel private(i,j,k)
!$omp do
do k = 1, 10000
   do j = 1, 20
      do i = 1, 4
         <calculations>
      end do
   end do
end do
!$omp end do
!$omp end parallel
```

But this depends on the number of iterations for each loop.

Having such small count of iterations in the outer loop is not efficient. We might even have less iterations than threads!

Re-arranging the nested loops will solve it!

But this is not always possible, or maybe all loops have relatively small counts!

What happens if we have nested loops?

Not much… we simply parallelize the outer loop.

```
!$omp parallel private(i,j,k)
!$omp do
do i = 1, 10
   do j = 1, 10
      do k = 1, 10
         <calculations>
      end do
   end do
end do
!$omp end do
!$omp end parallel
```

But this depends on the number of iterations for each loop.

Having such small count of iterations in the outer loop is not efficient. We might even have less iterations than threads!

Re-arranging the nested loops will solve it!

But this is not always possible, or maybe all loops have relatively small counts!

What happens if we have nested loops?

Not much… we simply parallelize the outer loop.

```
!$omp parallel private(i,j,k)
!$omp do collapse(3)
do i = 1, 10
   do j = 1, 10
      do k = 1, 10
         <calculations>
      end do
   end do
end do
!$omp end do
!$omp end parallel
```

To solve this we have the clause **collapse(n)**

It specifies how many nested loops will be collapsed in a single loop.

But this depends on the number of iterations for each loop.

Having such small count of iterations in the outer loop is not efficient. We might even have less iterations than threads!

Re-arranging the nested loops will solve it!

But this is not always possible, or maybe all loops have relatively small counts!

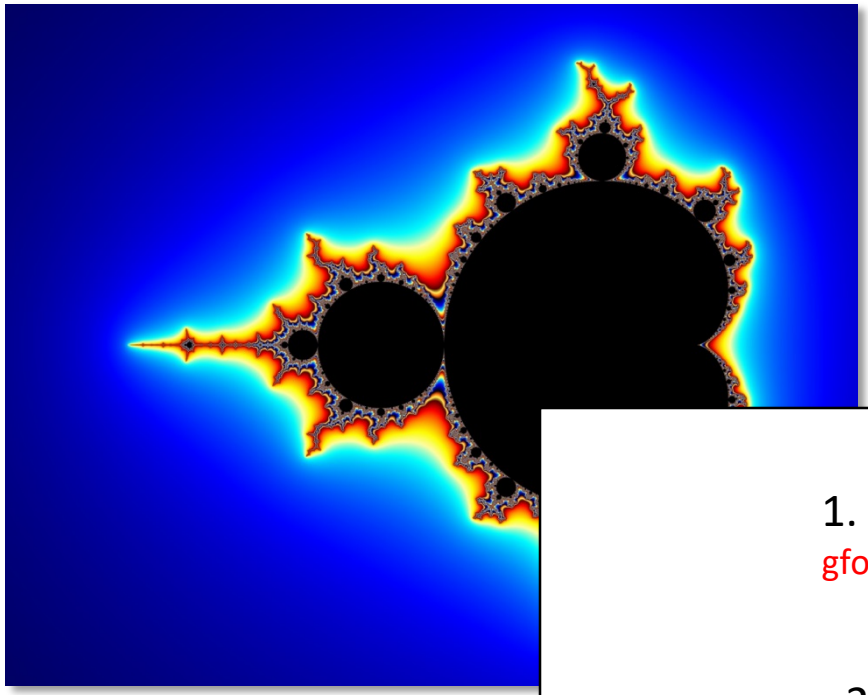
```
!$omp parallel private(ii,i,j,k)
!$omp do
do ii = 1, 1000
   i = mod(ii/100, 10)+1
   j = mod(ii/10, 10)+1
   k = mod(ii, 10)+1
   <calculations>
end do
!$omp end do
!$omp end parallel
```

What about subroutines? We can use orphaned directives

```
!$omp parallel private(i,j,k)

call calculate_manythings()

!$omp end parallel
```

```
subroutine calculate_manythings()

!$omp do schedule(static)
      do i=1,n
           <calculations>
      enddo
!$omp end do

return
end subroutine
```

Mandelbrot set area calculation



Mathematically it can be calculated exactly as:

$$A = \pi\left(1 - \sum_{n=1}^{\infty} n\, b_n^2\right)$$

But this series converges VERY slowly. It needs 10^{118} terms to get the first two digits, and 10^{1181} to get the third!

The area obtained by pixel counting is:

1. Compile and execute the serial code

gfortran mandelbrot.f90 –o mandelbrot_serial
time ./mandelbrot_serial

2. Parallelize the code with OpenMP

3. Compile and execute the parallel code

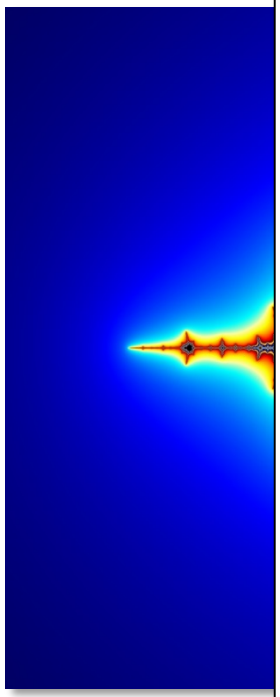
gfortran –fopenmp mandelbrot.f90 –o mandelbrot_parallel
time ./mandelbrot_parallel

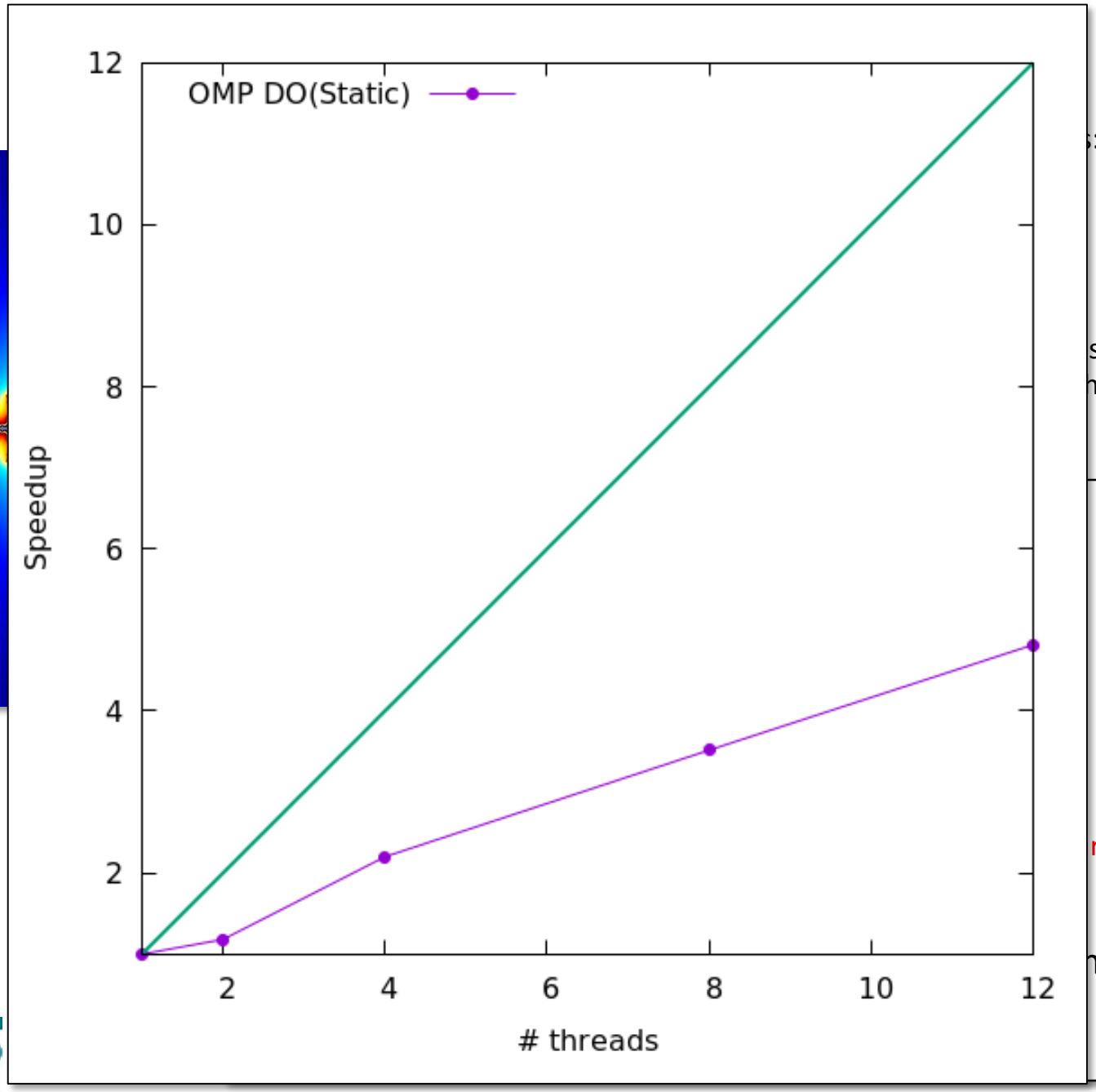
4. Change the number of threads (2-16) and plot the scaling

export OMP_NUM_THREADS = <# of threads>

OpenMP

Mandelbrot





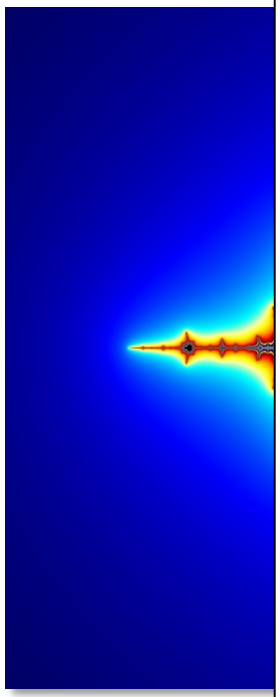
s 10^{118} terms to
hird!

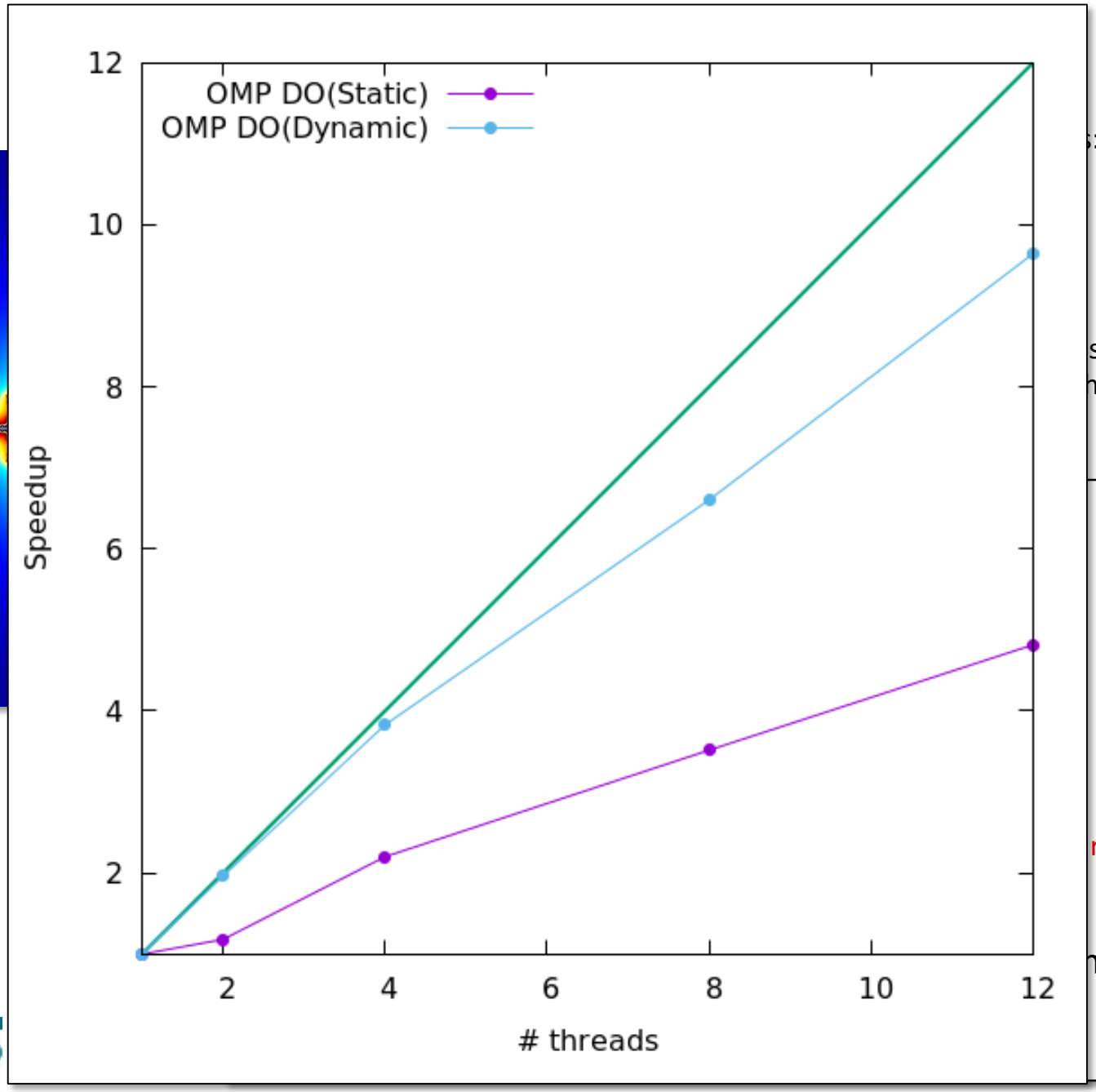
rallel

he scaling

OpenMP

Mandelbrot





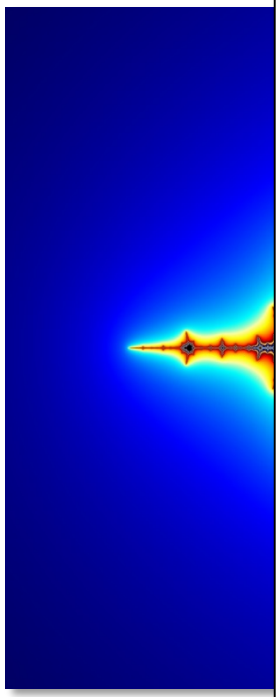
s 10^{118} terms to
hird!

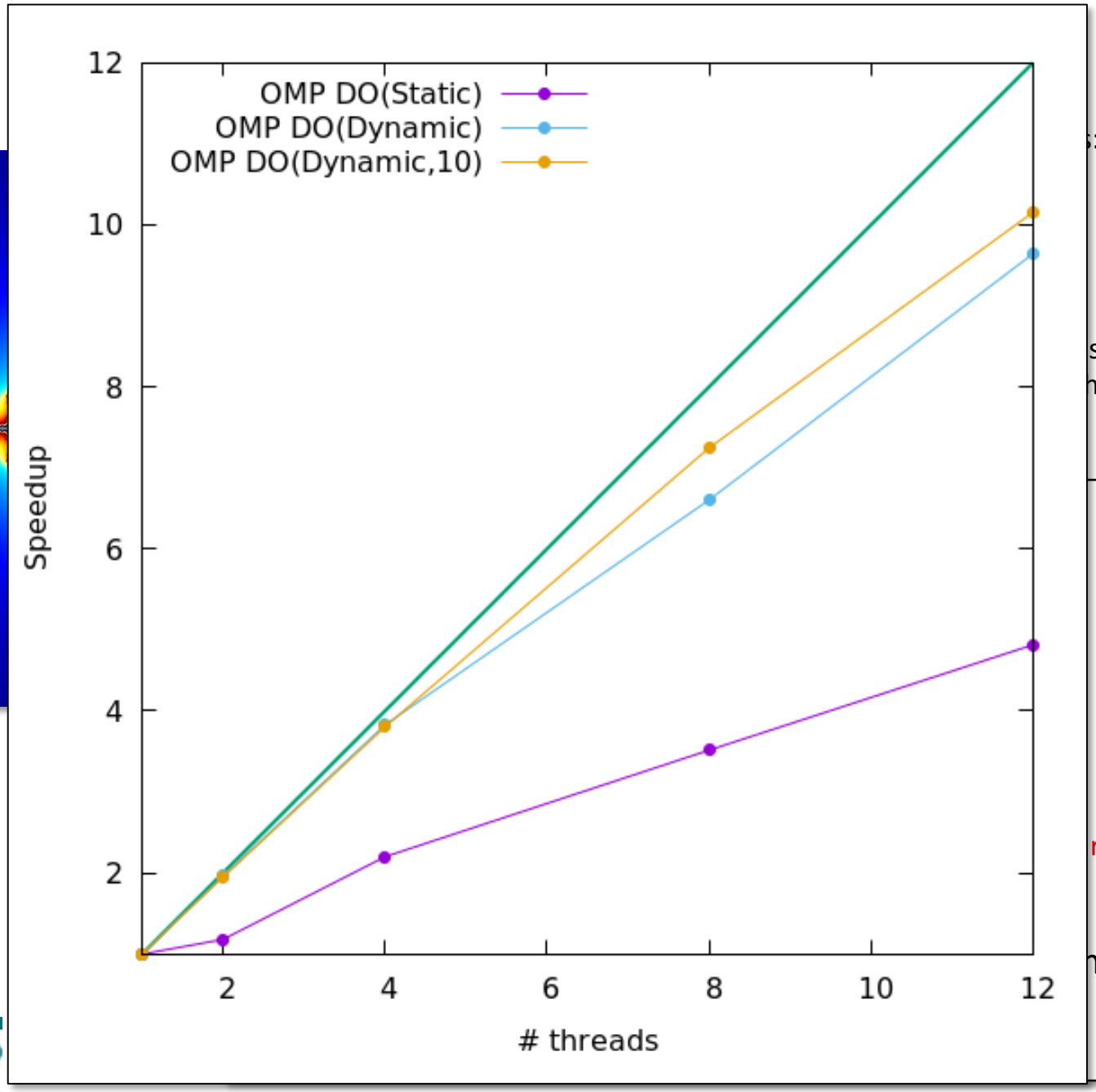
rallel

he scaling

OpenMP

Mandelbrot





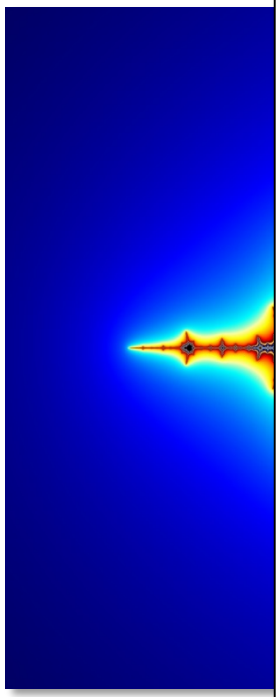
s 10^{118} terms to
hird!

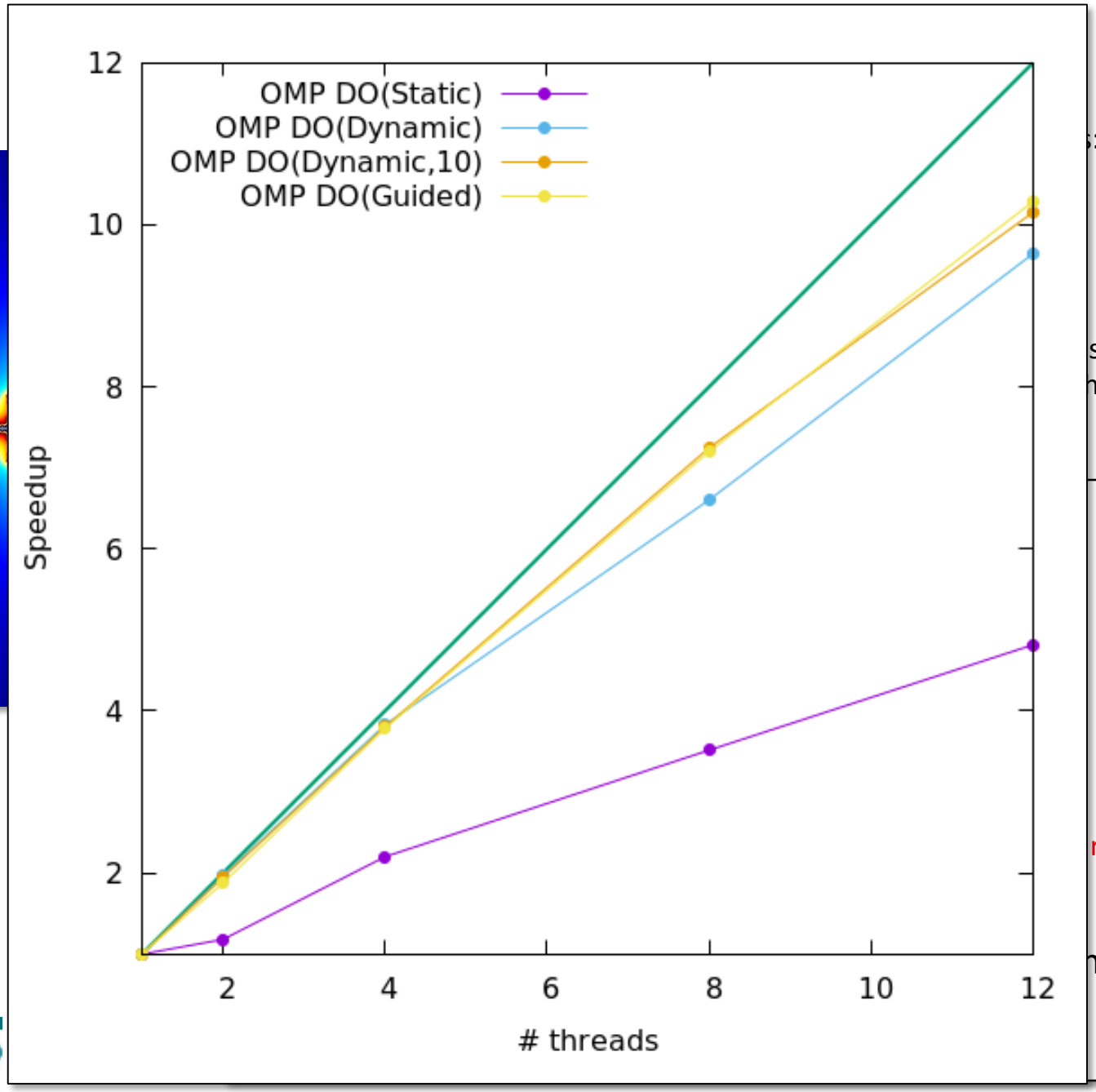
rallel

he scaling

OpenMP

Mandelbrot





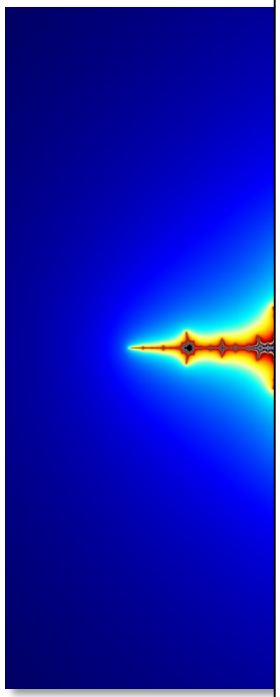
s 10^{118} terms to
hird!

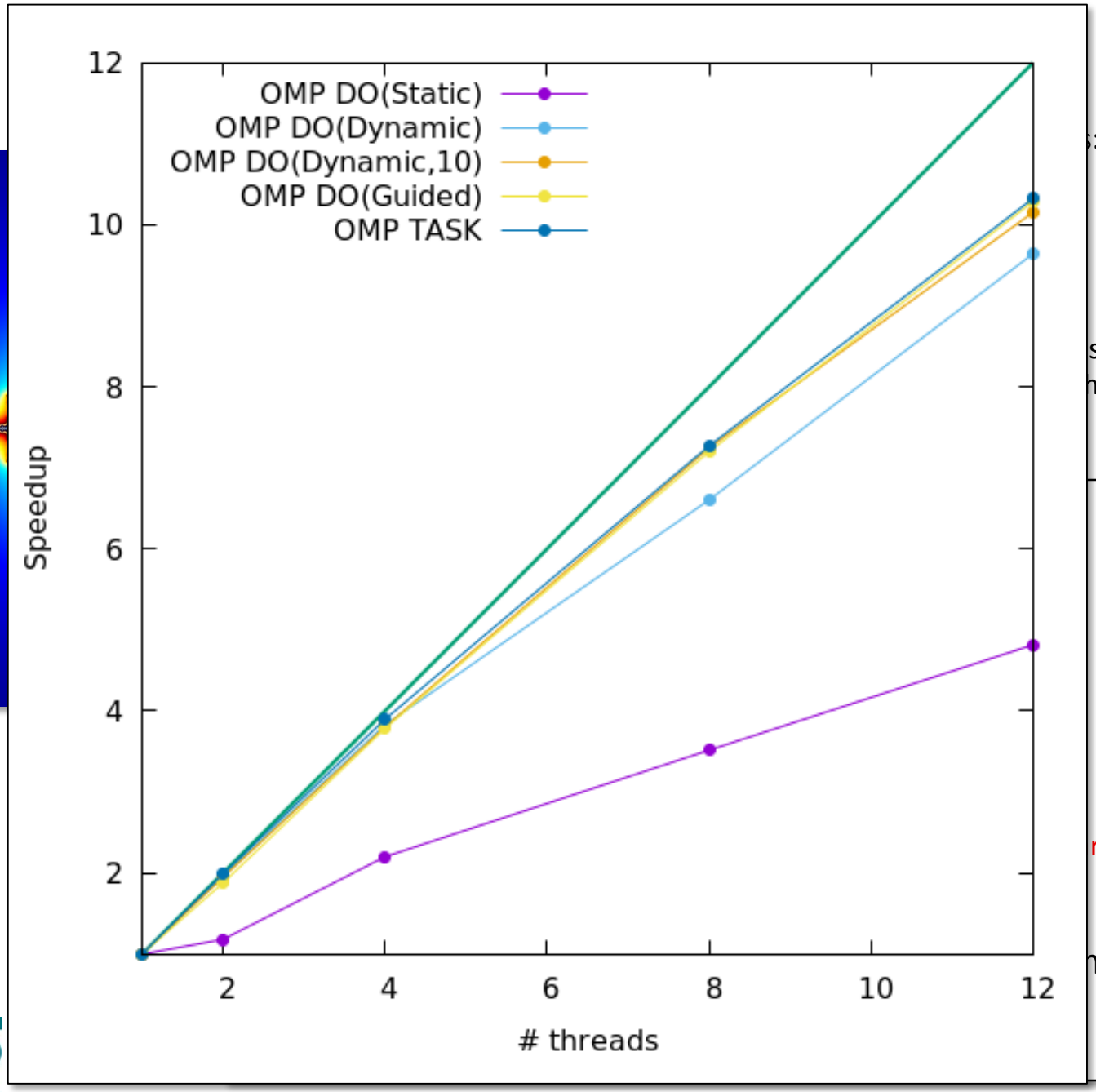
rallel

he scaling

OpenMP

Mandelbrot



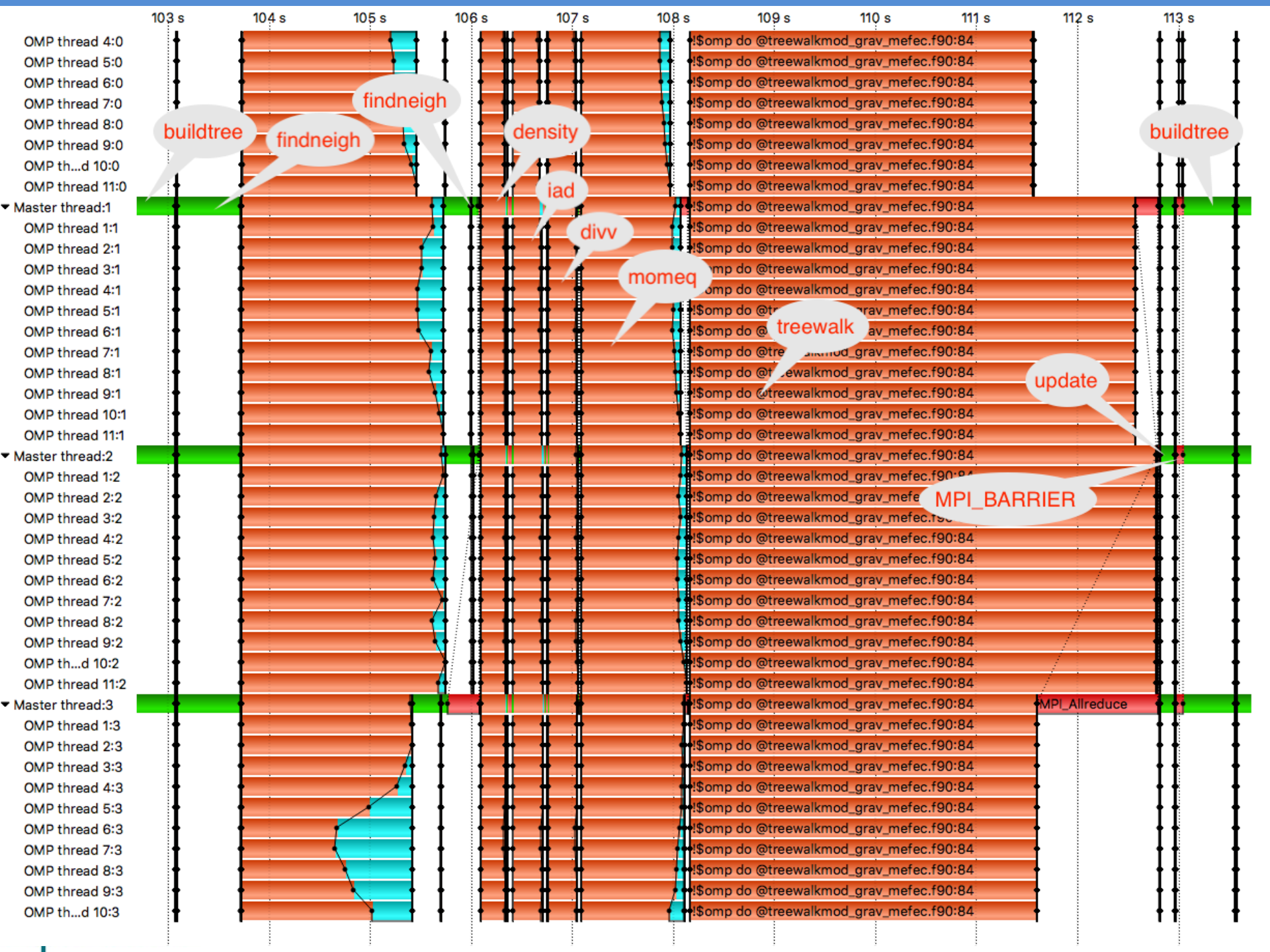


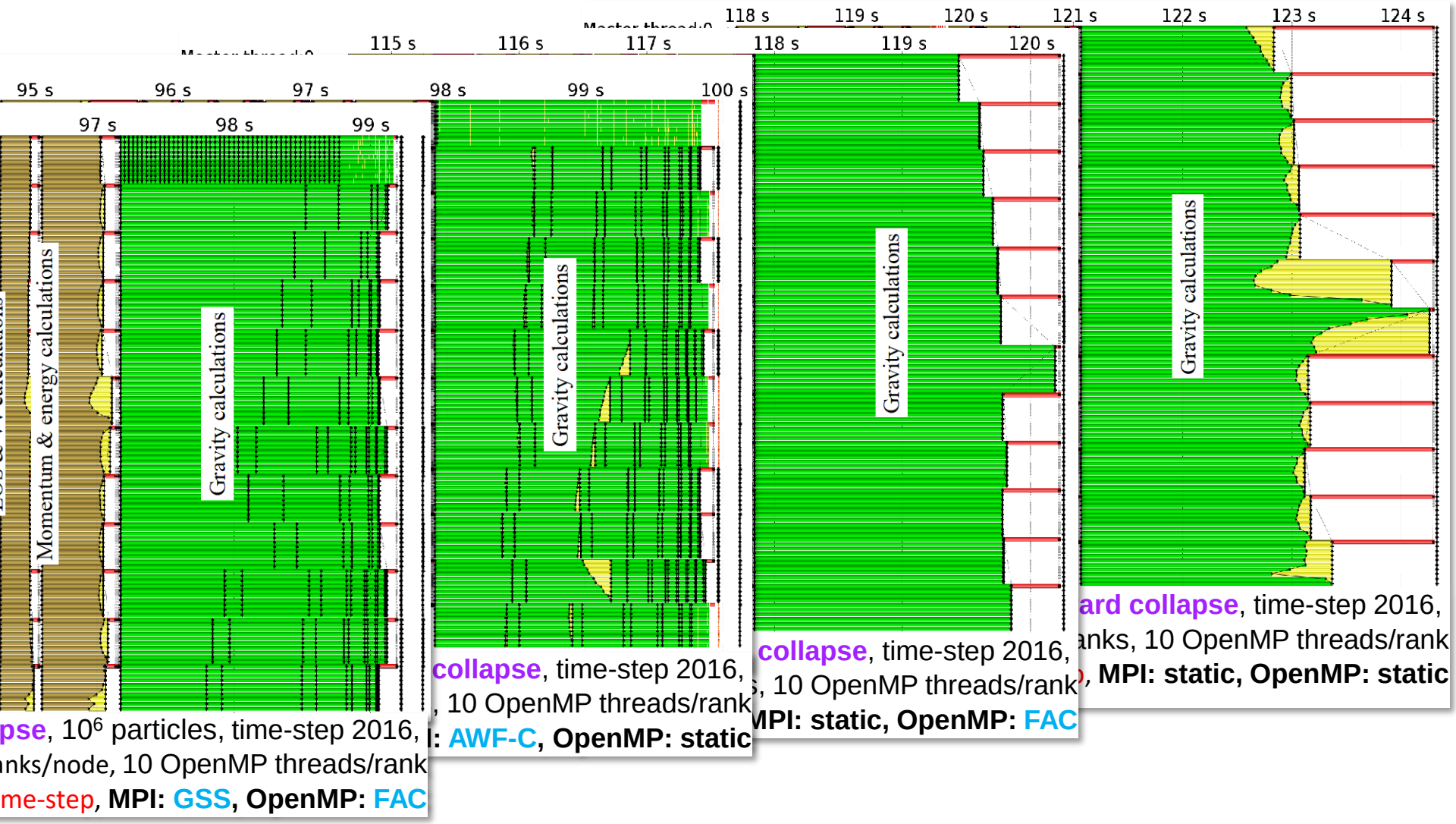
s 10^{118} terms to
hird!

rallel

he scaling

OpenMP


103 s
104 s
105 s
106 s
107 s
108 s
109 s
110 s
111 s
112 s
113 s
OMP thread 4:0
OMP thread 5:0
OMP thread 6:0
OMP thread 7:0
OMP thread 8:0
OMP thread 9:0
OMP th...d 10:0
OMP thread 11:0
Master thread:1
OMP thread 1:1
OMP thread 2:1
OMP thread 3:1
OMP thread 4:1
OMP thread 5:1
OMP thread 6:1
OMP thread 7:1
OMP thread 8:1
OMP thread 9:1
OMP thread 10:1
OMP thread 11:1
Master thread:2
OMP thread 1:2
OMP thread 2:2
OMP thread 3:2
OMP thread 4:2
OMP thread 5:2
OMP thread 6:2
OMP thread 7:2
OMP thread 8:2
OMP thread 9:2
OMP th...d 10:2
OMP thread 11:2
Master thread:3
OMP thread 1:3
OMP thread 2:3
OMP thread 3:3
OMP thread 4:3
OMP thread 5:3
OMP thread 6:3
OMP thread 7:3
OMP thread 8:3
OMP thread 9:3
OMP th...d 10:3
!$omp do @treewalkmod_grav_mefec.f90:84
buildtree
findneigh
findneigh
density
iad
divv
momeq
treewalk
update
buildtree
MPI_BARRIER
MPI_Allreduce



pse, 10^6 particles, time-step 2016,
nks/node, 10 OpenMP threads/rank
me-step, **MPI: GSS, OpenMP: FAC**

collapse, time-step 2016,
, 10 OpenMP threads/rank
I: AWF-C, OpenMP: static

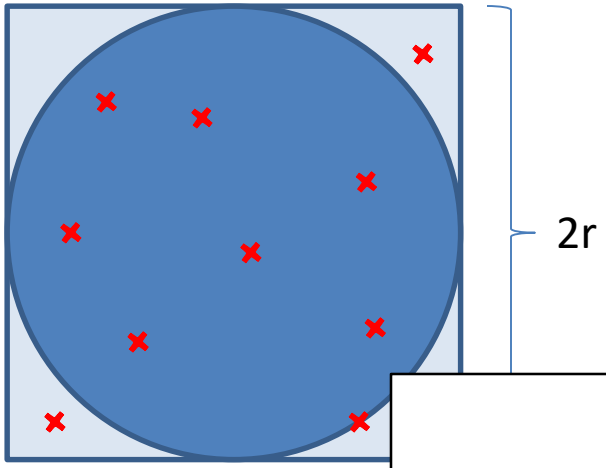
collapse, time-step 2016,
s, 10 OpenMP threads/rank
MPI: static, OpenMP: FAC

ard collapse, time-step 2016,
anks, 10 OpenMP threads/rank
, MPI: static, OpenMP: static

OpenMP

Two-level dynamic load balancing for high performance scientific applications
Mohammed, A. et al (Siam PP20)

Calculating Pi with darts



Throwing random points, the probability to fall inside of the circle is equal to the ratio of the areas:

$$P = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

Therefore π is 4 times the fraction of points that fall inside of the circle.

1. Compile and execute the serial code

gfortran montecarlo_pi.f90 –o pi_serial
time ./pi_serial

2. Parallelize the code with OpenMP

3. Compile and execute the parallel code

gfortran –fopenmp montecarlo_pi.f90 –o pi_parallel
time ./pi_parallel

4. Change the number of threads to 16 and compare the execution time with the serial version of the code

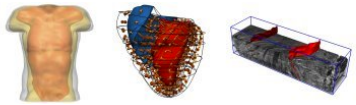
export OMP_NUM_THREADS = 16

Google

random generation openmp fortran

All Images Videos News Shopping More Settings Tools

About 356'000 results (0.40 seconds)



CMISS

Search Site Search
only in current section

Home CM openCMISS CMGUI Documentation Data

Log in Register

You are here: Home / openCMISS / Wiki / Random number generation with OpenMP

Navigation

- Technical committee meetings
- Issues with Interfaces Between C and Fortran
- Minutes 7 November 2005
- Outcomes
- References
- Agenda 14 November 2005
- ProjectManagementSoftware
- Development Program
- Agenda 21 November 2005

Random number generation with OpenMP

Generating random numbers in the shared-memory parallel processing environment of OpenMP has some traps for the unwary. The problem is immediately seen when using the Fortran intrinsic function 'random_number()', which generates a pseudo-random sequence of uniformly distributed random variates (RVs). Invoking 'random_number()' from different threads can make an OpenMP program run surprisingly slowly. What's more, the random performance of the generator may be compromised. The reason for this problem is discussed, and a solution is described.

The Cause of the Problem

Because each new state value is computed from the previous one, the integer variable IR4 must have the 'SAVE' attribute (equivalently, in C, it is static), i.e. its value is retained in memory between calls to 'random_number()'. The fact that this variable is not local means that invocations of 'random_number()' from different threads write to the same memory address for IR4. This leads to contention between threads, and results in big delays.

Because each new state value is computed from the previous one, the integer variable IR4 must have the 'SAVE' attribute (equivalently, in C, it is static), i.e. its value is retained in memory between calls to 'random_number()'. The fact that this variable is not local means that invocations of 'random_number()' from different threads write to the same memory address for IR4. This leads to contention between threads, and results in big delays.

The simple test program test_random_number.f90
illustrates the behaviour. The program should be built with the command:

```
xlf90_r -q64 -qsmp=omp -qsuffix=f=f90 test_random_number.f90 -o test_random_f90
```

and executed with:

```
test_random_number N
```

where N is the number of threads to use.

The following table shows execution time for generating 100 million uniform random variates using different numbers of threads.

News

candidate release 2012-01-31

Zinc 0.7.0.0 beta release 2011-11-10

More news...

You visited this page on 5/2/20.

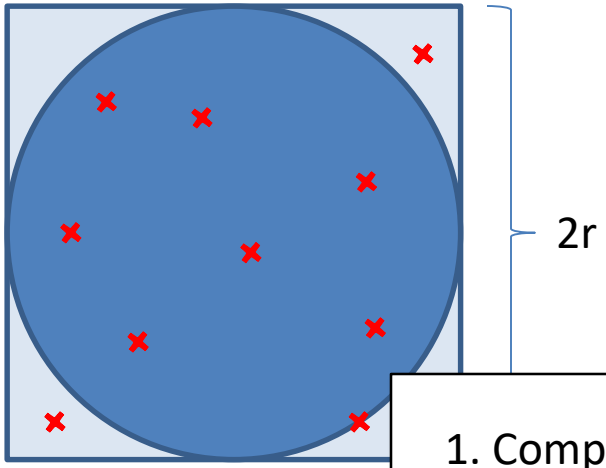
www.bnl.gov › files › pdf › OpenMPTutorial PDF

Introduction to Parallel Programming with OpenMP

pragma as **OpenMP** specific. Non. **OpenMP** compilers will ignore. In **Fortran** !**$omp** … c**$omp** … *$**omp** . … **Random** Number **Generator**. #include <stdio.h>.

OpenMP

Calculating Pi with darts



Throwing random points, the probability to fall inside of the circle is equal to the ratio of the areas:

$$P = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

Therefore π is 4 times the fraction of points that fall inside of the circle.

1. Compile and execute the alternative serial montecarlo code

gfortran montecarlo_pi_v2.f90 –o pi_serial_v2
time ./pi_serial_v2

2. Parallelize the code with OpenMP

3. Compile and execute the parallel code

gfortran –fopenmp montecarlo_pi_v2.f90 –o pi_parallel_v2
time ./pi_parallel_v2

4. Change the number of threads to 16 and compare the execution time with the serial version of the code

export OMP_NUM_THREADS = 16

Can we do any of this in Python? **Kind of…**

```python
from multiprocessing import Process


def func1():
  j=0
  print ('func1: starting')
  for i in range(1000000000):
      j=j+1
  print ('func1: finishing',j)

def func2():
  j=0
  print ('func2: starting')
  for i in range(1000000000):
      j=j+1
  print ('func2: finishing',j)

def func3():
  j=0
  print ('func3: starting')
  for i in range(1000000000):
      j=j+1
  print ('func3: finishing',j)


if __name__ == '__main__':
  p1 = Process(target=func1)
  p1.start()
  p2 = Process(target=func2)
  p2.start()
  p3 = Process(target=func3)
  p3.start()
  p1.join()
  p2.join()
  p3.join()
```

```
top - 11:28:00 up 2 days,  2:02,  1 user,  load average: 0,57, 0,56, 0,48
Tasks: 306 total,   4 running, 301 sleeping,   0 stopped,   1 zombie
%Cpu(s): 38,3 us,  0,4 sy,  0,0 ni, 61,1 id,  0,1 wa,  0,0 hi,  0,1 si,  0,0 st
KiB Mem :  8079052 total,   294168 free,  5072908 used,  2711976 buff/cache
KiB Swap: 19529724 total, 19506740 free,    22984 used.  1438376 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26027 ruben     20   0   50896   8488   3072 R 100,0  0,1   0:09.43 python
26028 ruben     20   0   50896   8488   3072 R 100,0  0,1   0:09.43 python
26026 ruben     20   0   50896   8488   3072 R  99,0  0,1   0:09.43 python
24587 ruben     20   0 4736720 2,196g 2,119g S   4,0 28,5   7:45.81 VirtualBox
 1128 root      20   0  756332 231528 197116 S   1,0  2,9  11:15.62 Xorg
 7796 ruben     20   0 1325324 184588  52036 S   1,0  2,3  21:43.70 skypeforli+
26032 ruben     20   0  437152  21932  18568 S   1,0  0,3   0:00.25 gnome-scre+
    1 root      20   0  185332   4984   3008 S   0,0  0,1   0:01.58 systemd
```

```
ruben@jarvis:~/test/parallelpython$ time python test.py
func1: starting
func2: starting
func3: starting
func3: finishing 1000000000
func2: finishing 1000000000
func1: finishing 1000000000
[1]+  Done                    emacs test.py

real    0m36.208s
user    1m50.284s
sys     0m0.544s
```

Can we do any of this in Python? **Kind of…**

```
from multiprocessing import Process


def func1():
  j=0
  print ('func1: starting')
  for i in range(1000000000):
      j=j+1
  print ('func1: finishing',j)

def func2():
  j=0
  print ('func2: starting')
  for i in range(1000000000):
      j=j+1
  print ('func2: finishing',j)

def func3():
  j=0
  print ('func3: starting')
  for i in range(1000000000):
      j=j+1
  print ('func3: finishing',j)

def runInParallel(*fns):
  proc = []
  for fn in fns:
    p = Process(target=fn)
    p.start()
    proc.append(p)
  for p in proc:
    p.join()


if __name__ == '__main__':
    runInParallel (func1, func2, func3)
```

```
top - 11:28:00 up 2 days,  2:02,  1 user,  load average: 0,57, 0,56, 0,48
Tasks: 306 total,   4 running, 301 sleeping,   0 stopped,   1 zombie
%Cpu(s): 38,3 us,  0,4 sy,  0,0 ni, 61,1 id,  0,1 wa,  0,0 hi,  0,1 si,  0,0 st
KiB Mem :  8079052 total,   294168 free,  5072908 used,  2711976 buff/cache
KiB Swap: 19529724 total, 19506740 free,    22984 used.  1438376 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26027 ruben     20   0   50896   8488   3072 R 100,0  0,1   0:09.43 python
26028 ruben     20   0   50896   8488   3072 R 100,0  0,1   0:09.43 python
26026 ruben     20   0   50896   8488   3072 R  99,0  0,1   0:09.43 python
24587 ruben     20   0 4736720 2,196g 2,119g S   4,0 28,5   7:45.81 VirtualBox
 1128 root      20   0  756332 231528 197116 S   1,0  2,9  11:15.62 Xorg
 7796 ruben     20   0 1325324 184588  52036 S   1,0  2,3  21:43.70 skypeforli+
26032 ruben     20   0  437152  21932  18568 S   1,0  0,3   0:00.25 gnome-scre+
    1 root      20   0  185332   4984   3008 S   0,0  0,1   0:01.58 systemd
```

```
ruben@jarvis:~/test/parallelpython$ time python test.py
func1: starting
func2: starting
func3: starting
func3: finishing 1000000000
func2: finishing 1000000000
func1: finishing 1000000000
[1]+  Done                    emacs test.py

real    0m36.208s
user    1m50.284s
sys     0m0.544s
```

http://stackabuse.com/parallel-processing-in-python/

We want to sum up the integer half-value of the first 3e8 integers.

```python
from multiprocessing import Process,Manager

def func1(id,results):
    j=0
    print('func1: starting')
    for i in range(100000000):
        j=j+i/2
    print('func1: finishing')
    results[id]=j

def func2(id,results):
    j=0
    print('func2: starting')
    for i in range(100000001,200000000):
        j=j+i/2
    print('func2: finishing')
    results[id]=j

def func3(id,results):
    j=0
    print('func3: starting')
    for i in range(200000001,300000000):
        j=j+i/2
    print('func3: finishing')
    results[id]=j

def runInParallel(*fns):
    proc=[]
    for i,fn in enumerate(fns):
        p = Process(target=fn,args=(i,results))
        p.start()
        proc.append(p)
    for p in proc:
        p.join()
    print(results)
    print(sum(results.values()))


if __name__ == '__main__':
    results=Manager().dict()
    runInParallel (func1,func2,func3)
```

```
[cabezon@login10 openmp]$ time python python_multiproc_2.py
func1: starting
func2: starting
func3: starting
func3: finishing
func2: finishing
func1: finishing
{0: 2499999950000000, 1: 7499999900000000, 2: 12499999850000000}
22499999700000000

real    0m12.819s
user    0m27.136s
sys     0m8.865s
[cabezon@login10 openmp]$
```

Note that threads finish in an unstructured way!

We store the results in a dictionary and sum them up.
(mimicking a ‘reduce’ in OpenMP)

More elegant way to do this is using ‘numba’. https://numba.pydata.org/

Adding a decorator numba compiles the code on-the-fly creating an optimized machine code

```
from numba import jit

@jit()
def func1():
    j=0
    print('func1: starting')
    for i in range(300000000):
        j=j+i/2
    print('func1: finishing',j)

if __name__ == '__main__':
    func1()
```

```
[cabezon@login10 openmp]$ time python python_multiproc_2.py
func1: starting
func2: starting
func3: starting
func3: finishing
func2: finishing
func1: finishing
{0: 2499999950000000, 1: 7499999900000000, 2: 12499999850000000}
22499999700000000

real    0m12.819s
user    0m27.136s
sys     0m8.865s
[cabezon@login10 openmp]$
```

```
[cabezon@login10 openmp]$ source numbatest/bin/activate
(numbatest) [cabezon@login10 openmp]$ time python python_numba.py
func1: starting
func1: finishing 2.249999988355443e+16

real    0m1.142s
user    0m1.205s
sys     0m1.294s
(numbatest) [cabezon@login10 openmp]$
```

More elegant way to do this is using ‘numba’. https://numba.pydata.org/

Adding a decorator numba compiles the code on-the-fly creating an optimized machine code

```
[cabezon@login10 openmp]$ time python python_multiproc_2.py
func1: starting
func2: starting
func3: starting
func3: finishing
func2: finishing
func1: finishing
{0: 2499999950000000, 1: 7499999900000000, 2: 12499999850000000}
22499999700000000

real    0m12.819s
user    0m27.136s
sys     0m8.865s
```

```
from numba import jit

@jit()
def func1():
    j=0
    print('func1: starting'
    for i in range(30000000
        j=j+i/2
    print('func1: finishing

if __name__ == '__main__':
    func1()
```

We need to install numba. For that we will install a virtual environment of Python:

virtualenv testnumba

source testnumba/bin/activate

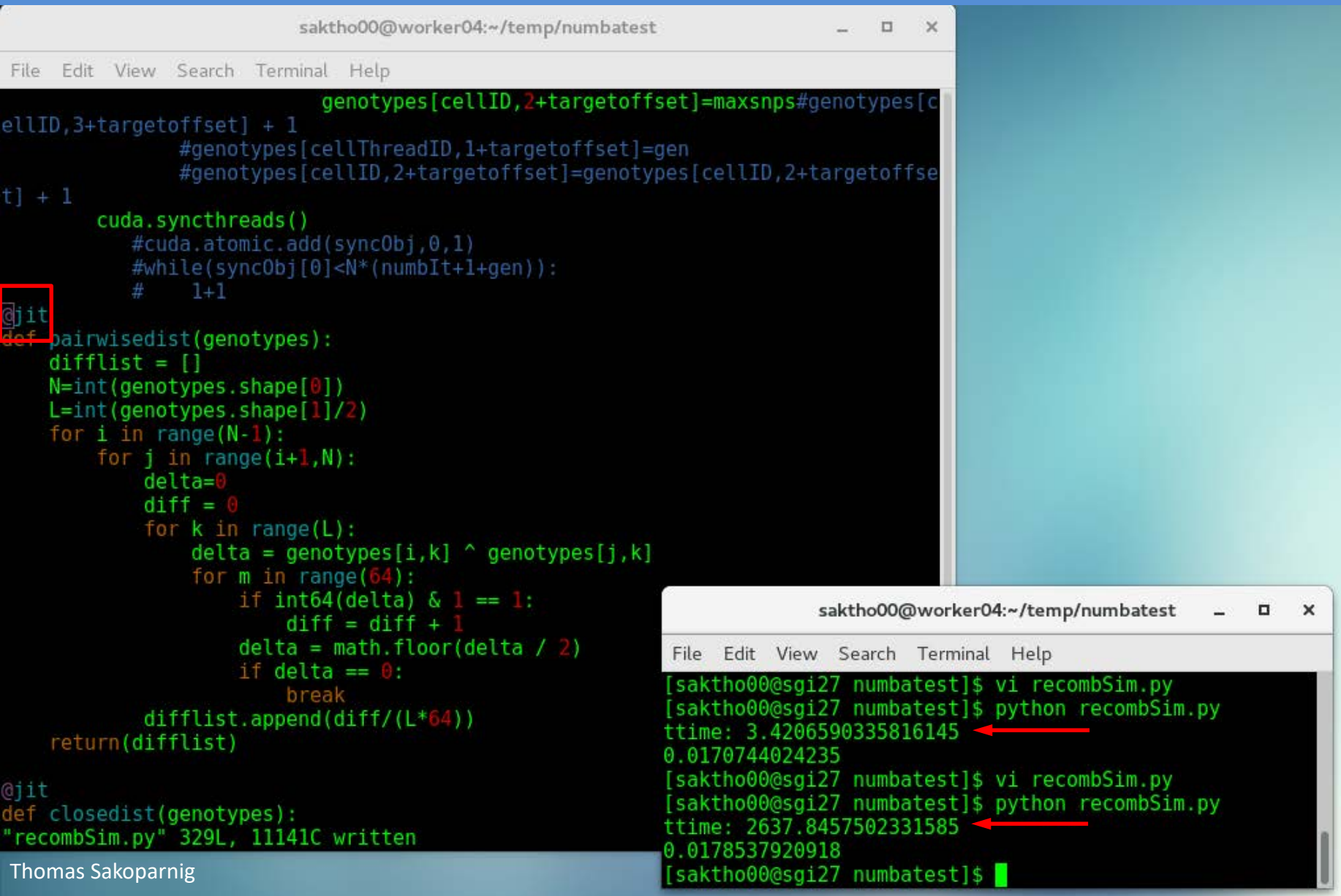
2. Install numba

pip install numba

3. Execute the serial version with the numba decorator

(don’t forget to import jit at the beginning of the Python script)

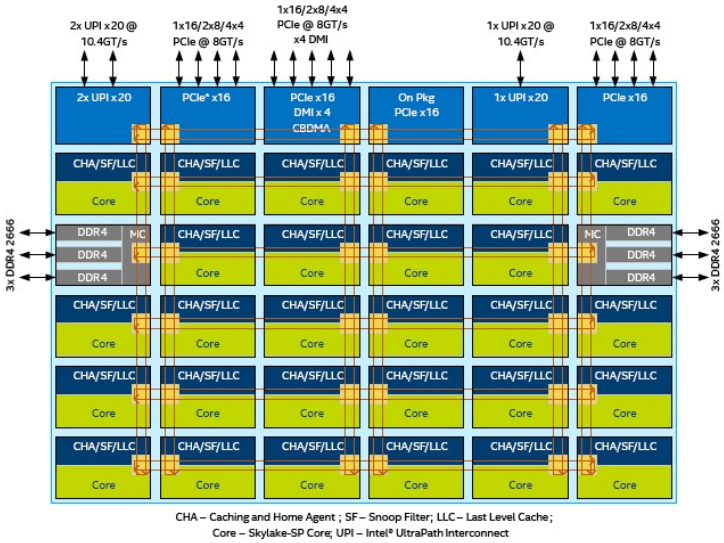
time python python_serial.py



Thomas Sakoparnig

What about GPU?

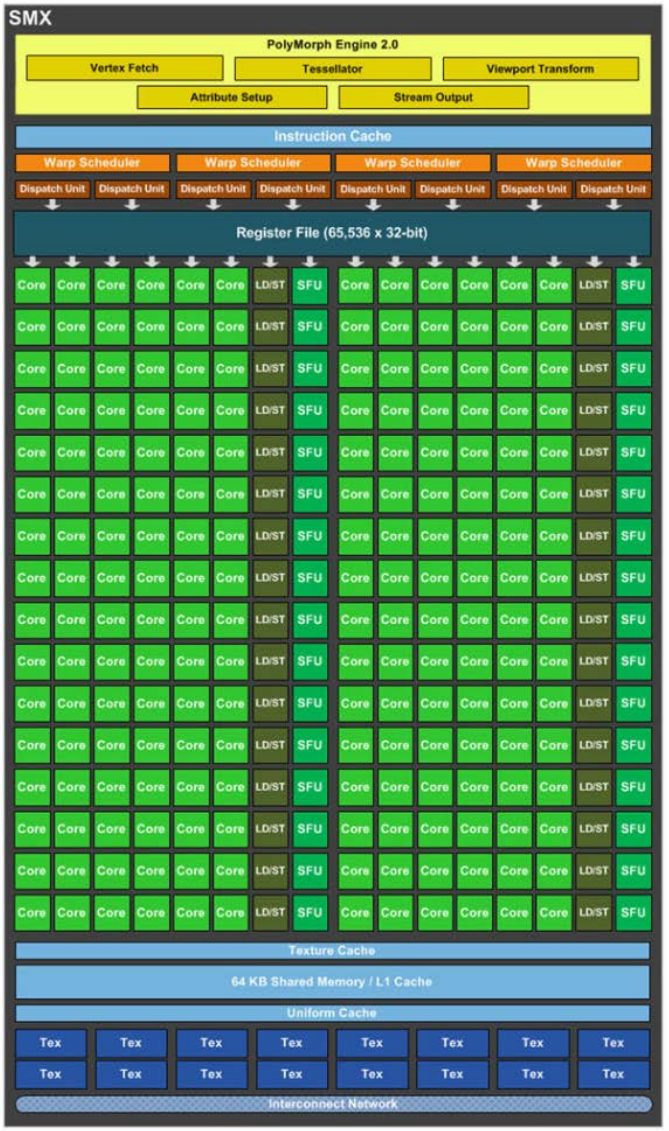
Intel Skylake (2017)



CHA – Caching and Home Agent ; SF – Snoop Filter; LLC – Last Level Cache ;
Core – Skylake-SP Core; UPI – Intel® UltraPath Interconnect

Tesla SM unit (2007)

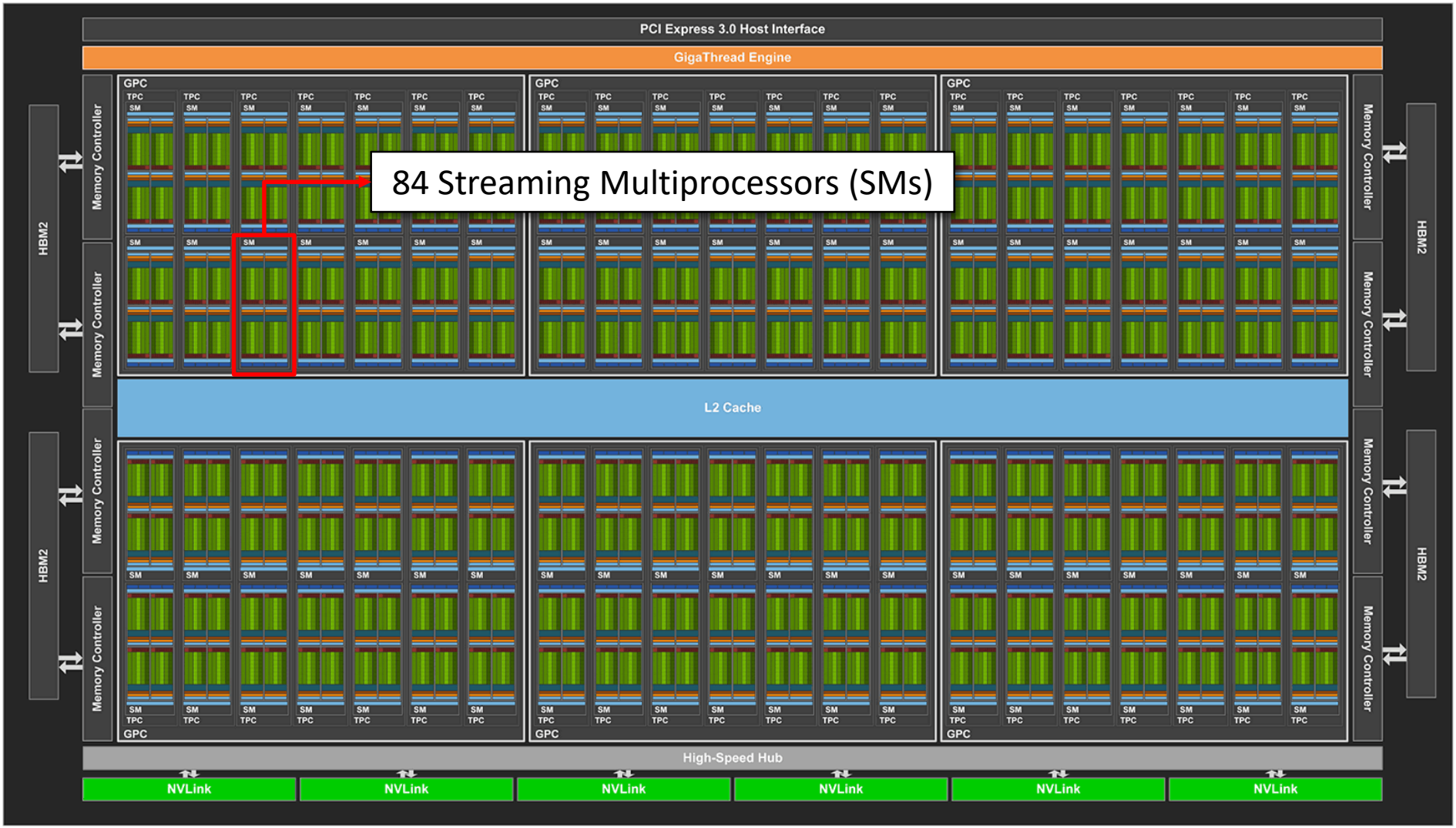
Fermi SMX unit (2011)



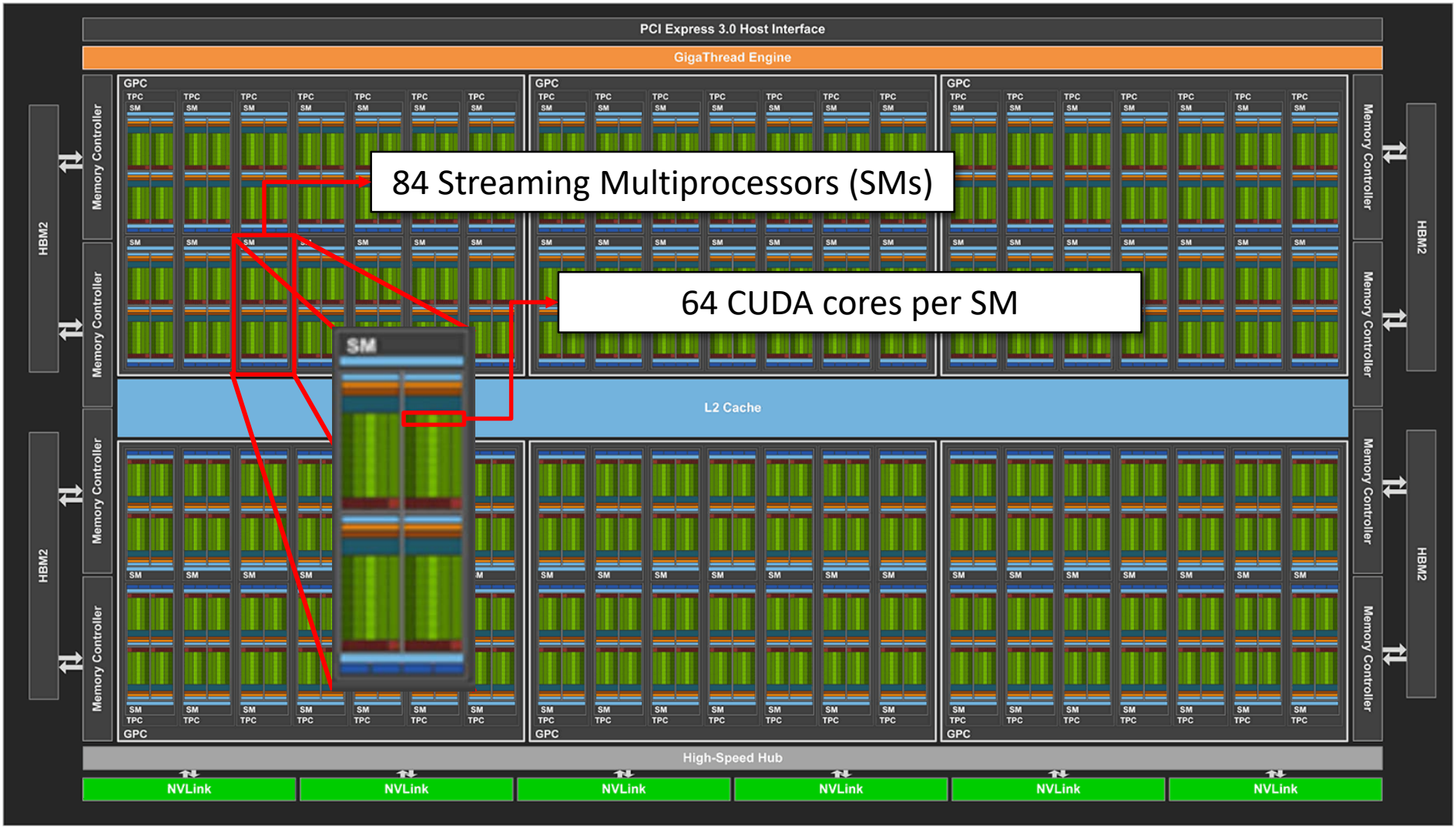
Volta (2017)


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

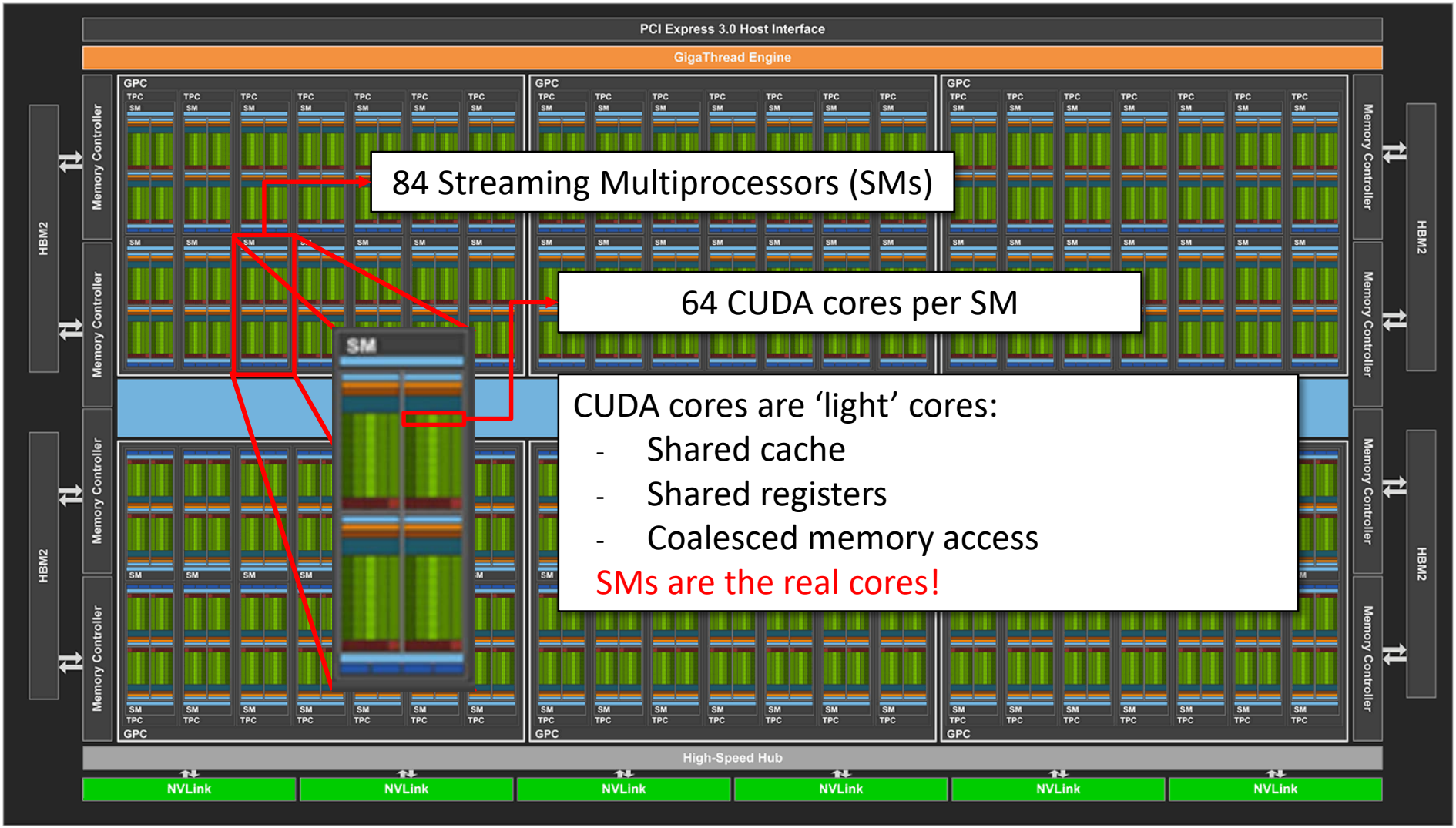
Volta (2017)


PCI Express 3.0 Host Interface
GigaThread Engine
84 Streaming Multiprocessors (SMs)
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

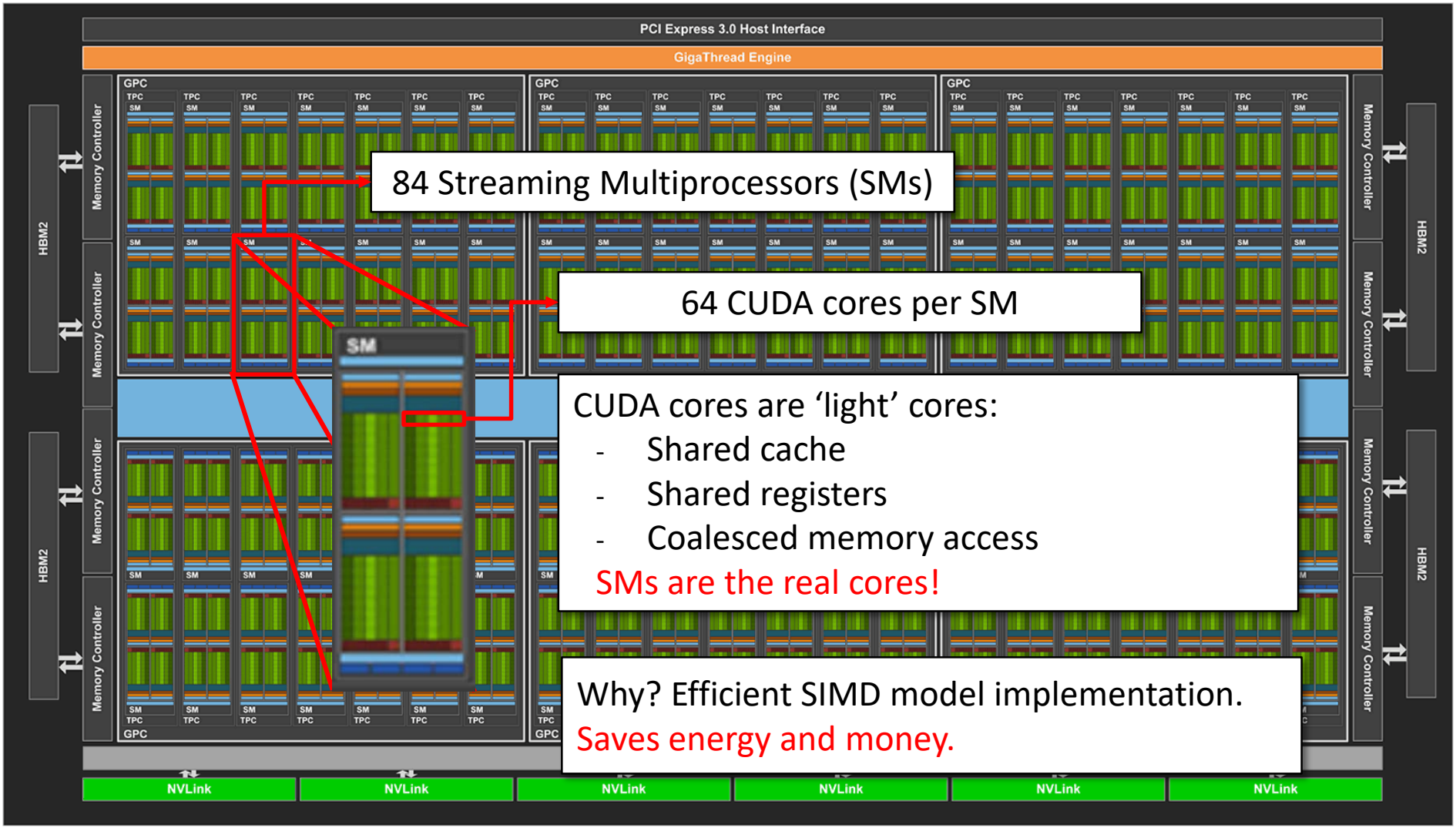
Volta (2017)


PCI Express 3.0 Host Interface
GigaThread Engine
84 Streaming Multiprocessors (SMs)
64 CUDA cores per SM
SM
L2 Cache
GPC
TPC
SM
Memory Controller
HBM2
High-Speed Hub
NVLink

Volta (2017)


PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
HBM2
GPC
TPC
SM
84 Streaming Multiprocessors (SMs)
64 CUDA cores per SM
CUDA cores are 'light' cores:
- Shared cache
- Shared registers
- Coalesced memory access
SMs are the real cores!
High-Speed Hub
NVLink

Volta (2017)


PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
HBM2
GPC
TPC
SM
84 Streaming Multiprocessors (SMs)
64 CUDA cores per SM
CUDA cores are ‘light’ cores:
- Shared cache
- Shared registers
- Coalesced memory access
SMs are the real cores!
Why? Efficient SIMD model implementation.
Saves energy and money.
NVLink

Since version 4.0, openMP can do GPU offloading.

```
int A[1] = {-1};
#pragma omp target
{
   A[0] = omp_is_initial_device();
}
if (!A[0]) {
    printf("Able to use offloading!\n");
}
```

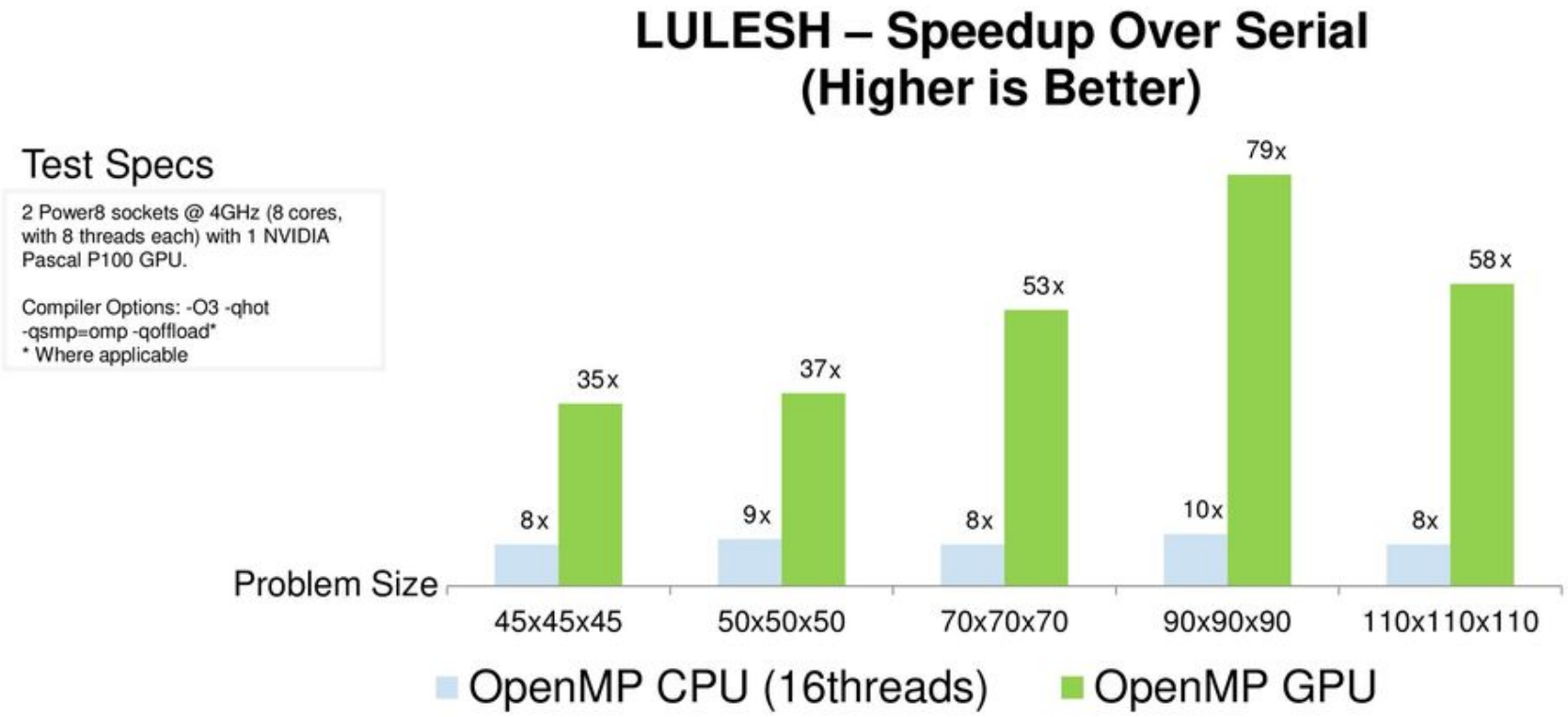
omp_is_initial_device() returns 0 when called from the accelerator

You can control the transfer of data

```
#pragma omp target data map(to: b[:n])
{
    // 'b' copied onto device

    // This region is offloaded to device
    #pragma omp target map(tofrom: a[:n])
    {
        // 'a' copied onto device
        #pragma omp teams distribute
        for(int ii = 0; ii < n; ++ii)
        {
            a[ii] = a[ii] + alpha * b[ii];
        }
    }
    // 'a' copied back

    // ... potentially more target regions
}
// 'b' is discarded by device
```

Since version 4.0, openMP can do GPU offloading.

Performance Results – End to End

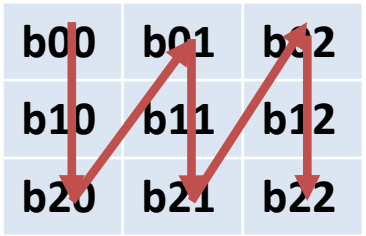


Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```

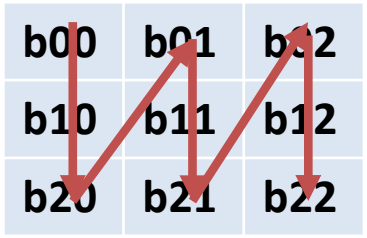


Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping him…

Processor cache

Cache Misses: 0

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

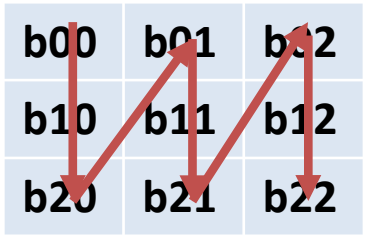
i	j	k
0	0	0

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
         c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping him…

Processor cache

Cache Misses: 1

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

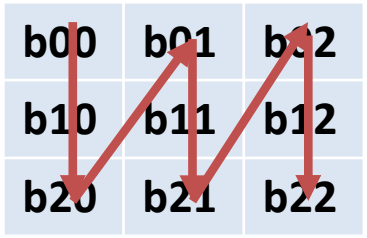
i	j	k
0	0	1

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping him…

Processor cache

Cache Misses: 2

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

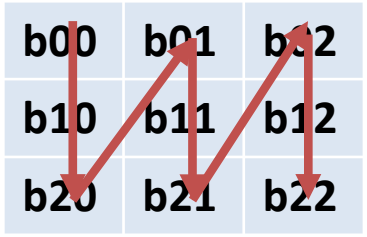
i	j	k
0	0	2

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping him…

Processor cache

Cache Misses: 3

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

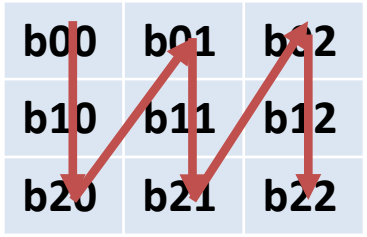
i	j	k
0	1	0

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping him…

Processor cache

Cache Misses: 4

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

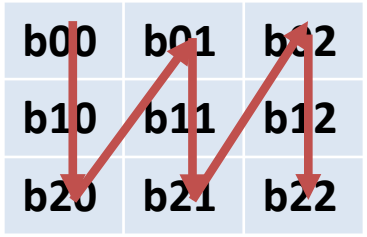
i	j	k
0	1	1

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping him…

Processor cache

Cache Misses: 5

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

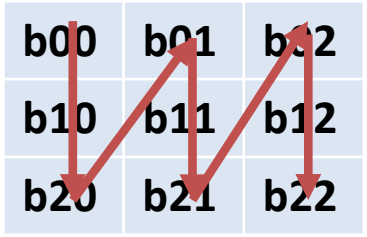
i	j	k
0	1	2

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping him…

Processor cache

Cache Misses: 6

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

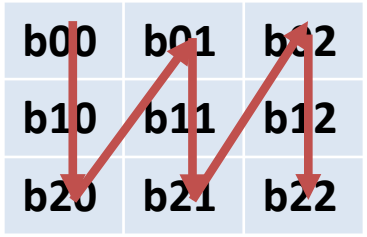
i	j	k
0	2	0

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping him…

Processor cache

Cache Misses: 7

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

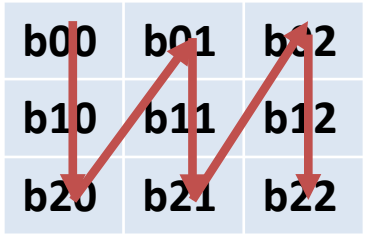
i	j	k
0	2	1

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



- Processors **load data into their cache** for fast reuse
- They always load a bunch of data at the same time
- They do **branch-prediction** and **pre-fetching**!

However here we are not helping him…

Processor cache

Cache Misses: 8

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

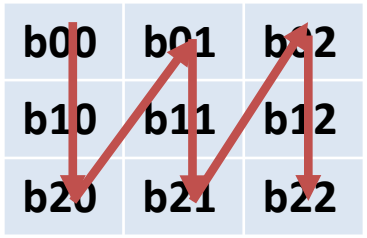
i	j	k
0	2	2

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
       for(int j=0; j<n; ++j)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```

Let's go…

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

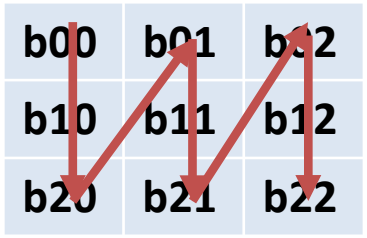
i	k	j
0	0	0

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

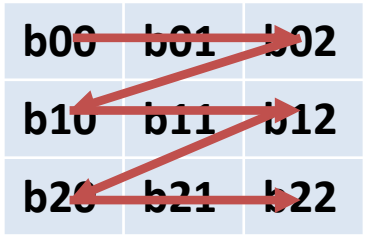
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
       for(int j=0; j<n; ++j)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



Cache Misses: 0

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

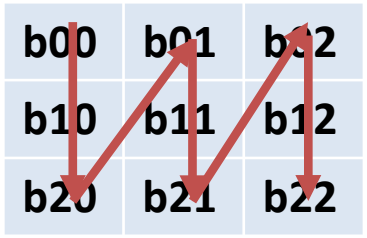
i	k	j
0	0	1

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

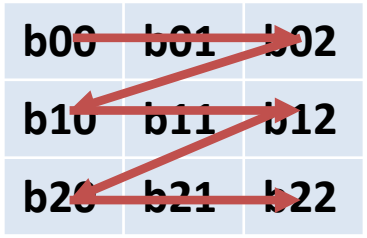
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
       for(int j=0; j<n; ++j)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



Cache Misses: 0

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

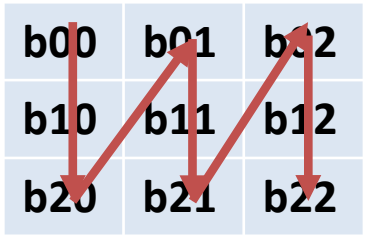
i	k	j
0	0	2

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

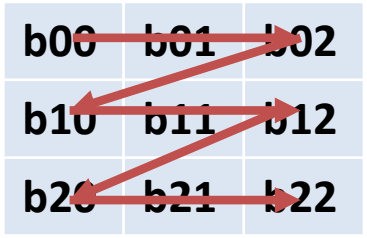
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
       for(int j=0; j<n; ++j)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



Cache Misses: 1

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

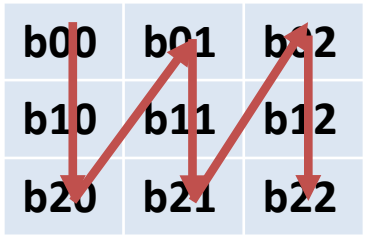
i	k	j
0	1	0

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

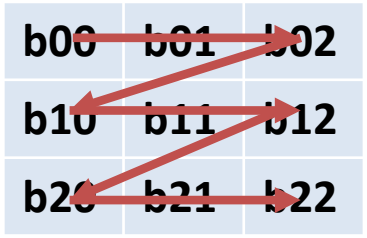
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
       for(int j=0; j<n; ++j)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



Cache Misses: 1

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

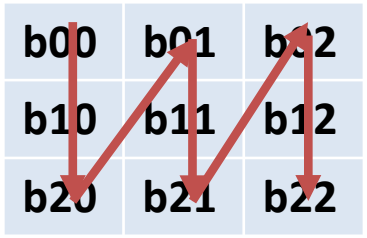
i	k	j
0	1	1

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

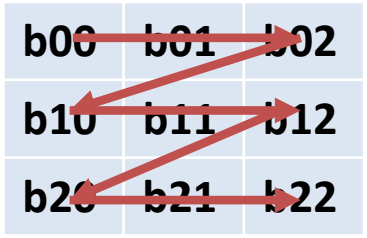
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
       for(int j=0; j<n; ++j)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



Cache Misses: 1

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

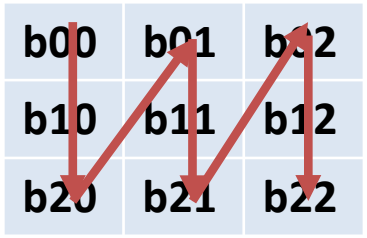
i	k	j
0	1	2

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

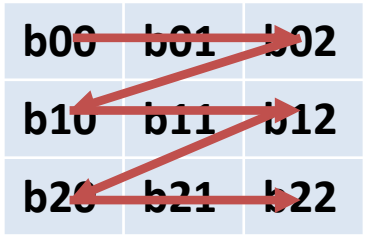
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
       for(int j=0; j<n; ++j)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



Cache Misses: 2

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

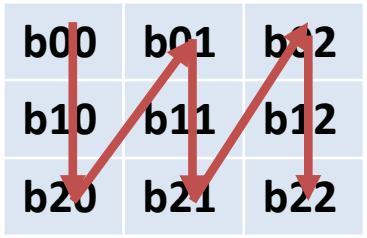
i	k	j
0	2	0

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

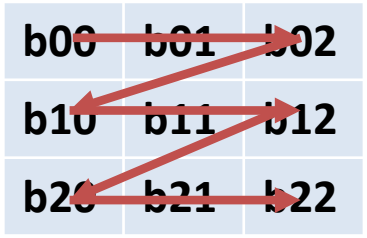
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
       for(int j=0; j<n; ++j)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



Cache Misses: 2

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

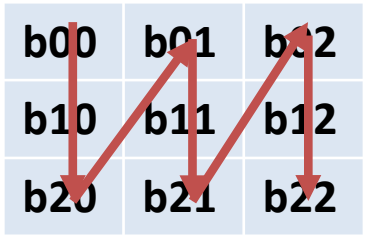
i	k	j
0	2	1

Full example: matrix multiplication

Sub-optimal memory access will kill the performance…

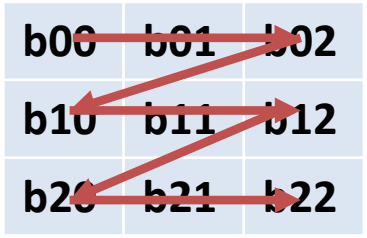
The less efficient way (column-major):

```
for(int i=0; i<n; ++i)
   for(int j=0; j<n; ++j)
      for(int k=0; k<n; ++k)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



The better way (row-major):

```
for(int i=0; i<n; ++i)
   for(int k=0; k<n; ++k)
       for(int j=0; j<n; ++j)
          c[i*n+j] += a[i*n+k] * b[k*n+j];
```



Cache Misses: 2

a00	a01	a02	a10	a11	a12	a20	a21	a22
b00	b01	b02	b10	b11	b12	b20	b21	b22
c00	c01	c02	c10	c11	c12	c20	c21	c22

i	k	j
0	2	2

Full example: matrix multiplication with OpenMP / Numba

```
#pragma omp parallel for
for(int i=0; i<size; ++i)
   for(int k=0; k<size; ++k)
      for(int j=0; j<size; ++j)
         c[i*size+j] += a[i*size+k] * b[k*size+j];
```

C++

```
!$omp parallel private(i,j,k)
!$omp do schedule(static)
do i=0,size-1
   do k=0,size-1
      do j=0,size-1
         c(i*size+j) = c(i*size+j) + a(i*size+k) * b(k*size+j)
      enddo
   enddo
enddo
!$omp end do
!$omp end parallel
```

FORTRAN

```
@jit(Parallel=True)
def mat_mul(a,b,c,size):
    for i in range(size):
        for k in range(size):
            for j in range(size):
                c[i*size+j] += a[i*size+k] + b[k*size+j]
```

Python

Full example: matrix multiplication with GPU offloading (OpenMP 4.5+)

```
!$omp target map(tofrom:c) map(to:a,b)
!$omp teams distribute parallel do private(i,j,k) collapse(2)
do i=0,size-1
   do k=0,size-1
      do j=0,size-1
         c(i*size+j) = c(i*size+j) + a(i*size+k) * b(k*size+j)
      enddo
   enddo
enddo
!$omp end target
```

FORTRAN

```
#pragma omp target map(to: a[0:n2], b[0:n2]), map(tofrom: c[0:n2])
#pragma omp teams distribute parallel for collapse(2)
for(int i=0; i<size; ++i)
    for(int j=0; j<size; ++j)
        for (int k = 0; k < size; k++)
          c[i*size+j] += a[i*size+k] * b[k*size+j];
```

C++

Full example: matrix multiplication with CUDA

```
__global__ void cuda_mul(float* a, float* b, float* c, int size) {
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;
    for (int i = 0; i < size; i++)
        c[row*size+col] += a[row*size+i] * b[i*size+col];
}
```

CUDA Kernel

```
cudaMemcpy(dm1, a, sizeof(float)*size*size, cudaMemcpyHostToDevice);
cudaMemcpy(dm2, b, sizeof(float)*size*size, cudaMemcpyHostToDevice);
cudaMemcpy(dm3, c, sizeof(float)*size*size, cudaMemcpyHostToDevice);

dim3 blockSize = dim3(16, 16);
dim3 gridSize = dim3(size / blockSize.x, size/ blockSize.y);

cuda_mul<<<gridSize, blockSize>>>(dm1, dm2, dm3, size);

cudaMemcpy(c, dm3, sizeof(float)*size*size, cudaMemcpyDeviceToHost);
```

Kernel Call

Full example: matrix multiplication with CUDA

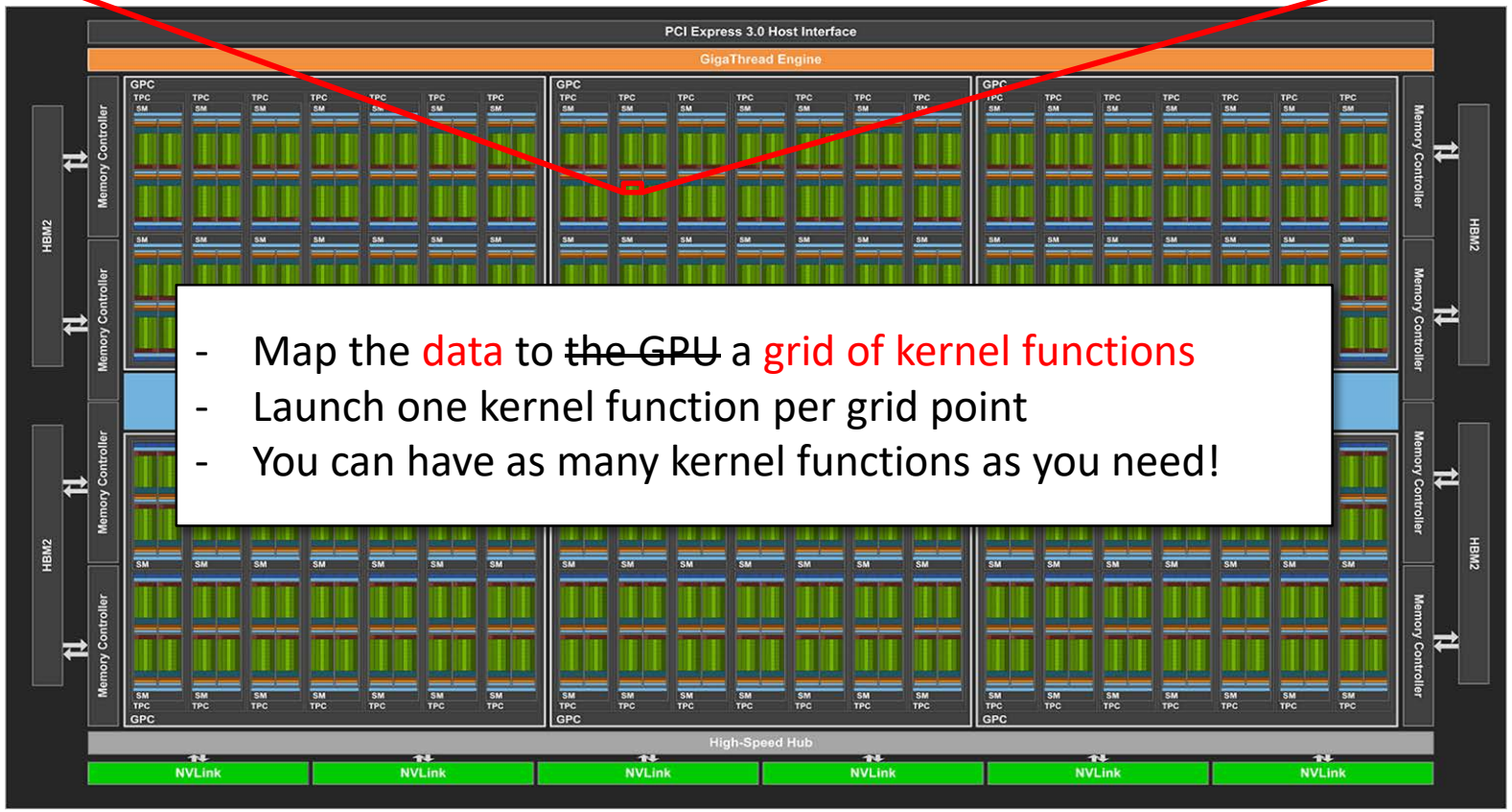
```
__global__ void cuda_mul(float* a, float* b, float* c, int size) {
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;
    for (int i = 0; i < size; i++)
        c[row*size+col] += a[row*size+i] * b[i*size+col];
}
```

CUDA Kernel

Full example: matrix multiplication with CUDA

```
__global__ void cuda_mul(float* a, float* b, float* c, int size) {
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;
    for (int i = 0; i < size; i++)
        c[row*size+col] += a[row*size+i] * b[i*size+col];
}
```

CUDA Kernel



- Map the data to ~~the GPU~~ a grid of kernel functions
- Launch one kernel function per grid point
- You can have as many kernel functions as you need!

Full example: matrix multiplication with CUDA and Numba

```
@cuda.jit
def mat_mul(a,b,c,size):
    row=cuda.blockIdx.y*cuda.blockDim.y+cuda.threadIdx.y
    col=cuda.blockIdx.x*cuda.blockDim.x+cuda.threadIdx.x
    for i in range(size):
        c[row*size+col] += a[row*size+i] * b[i*size+col]
```

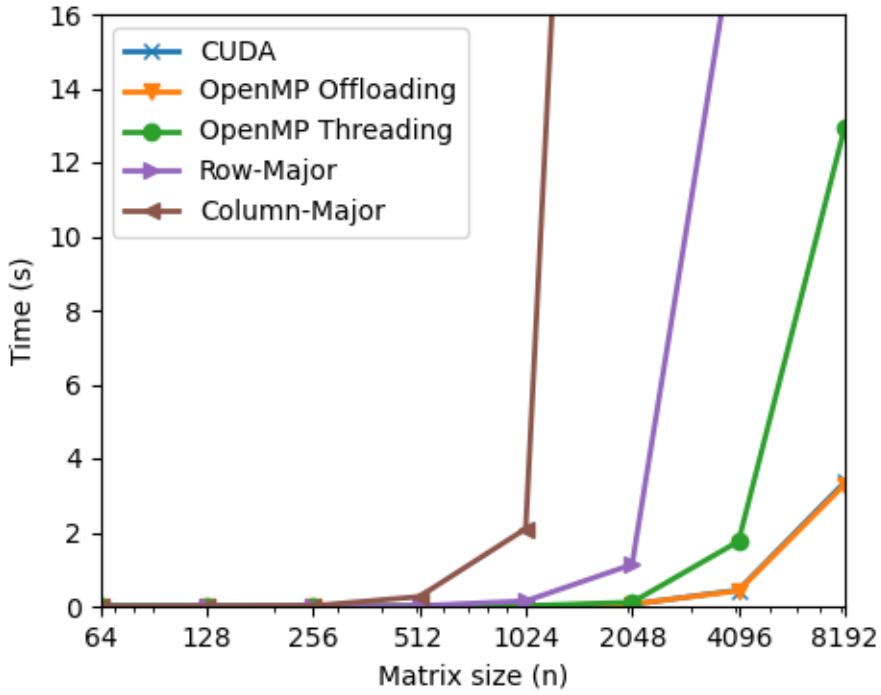
CUDA Kernel

```
threadsperblock = (16,16)
blockspergrid_x = int(np.ceil(size_array / threadsperblock[0]))
blockspergrid_y = int(np.ceil(size_array / threadsperblock[1]))
blockspergrid = (blockspergrid_x, blockspergrid_y)

mat_mul[blockspergrid, threadsperblock](a,b,c,size_array)
```

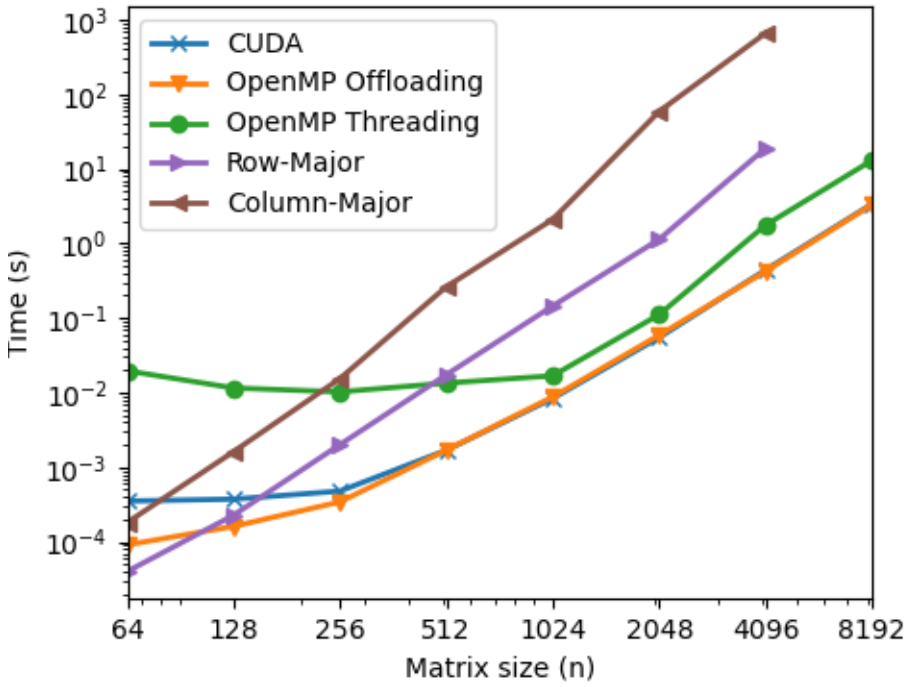
Kernel Call

Full example: matrix multiplication performance

Linear / log scale



Log / log scale



N = 4096
Column-Major: 658s
OpenMP 4.5/CUDA GPU: 0.43s

=> 1500x performance speedup!

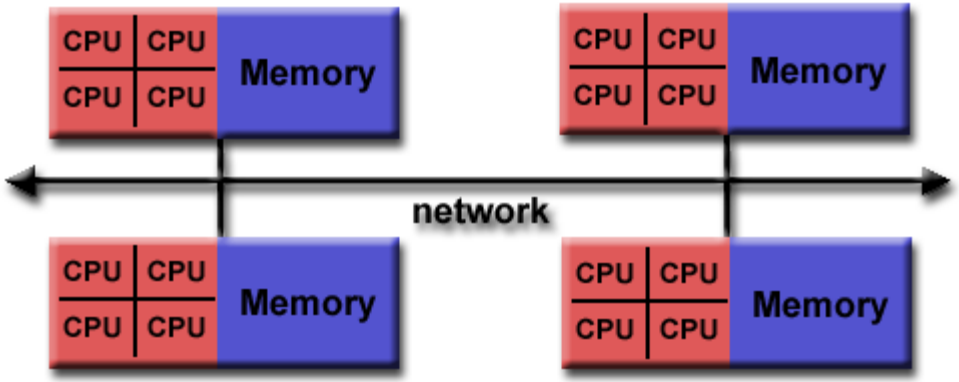
There are two main ways of parallelizing a code

Open **M**ulti-**P**rocessing

Message **P**assing **I**nterface

Both are APIs for C/C++ and Fortran

Works in shared memory systems	Works in shared and distributed memory systems
Can use as many cores as there are in the compute node	Can use as many cores as there are in the cluster
Uses threads to split the work	Uses procesess to split the work
Threads have both private and shared variables	Processes have only private variables
Uses preprocessor directives in commented lines	Loads a library and uses specific commands



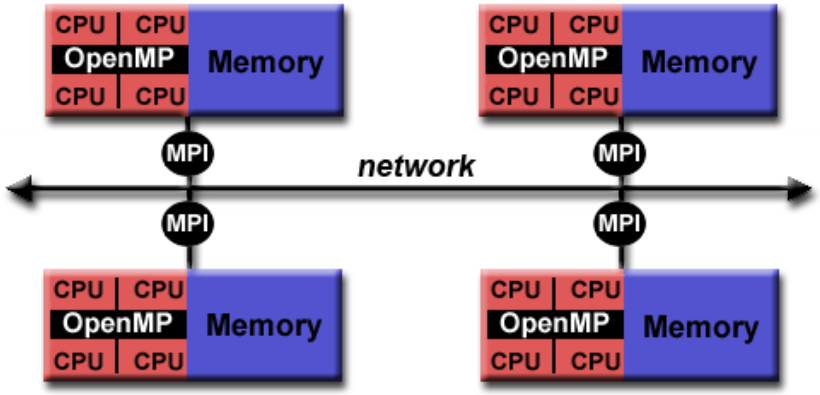
There are two main ways of parallelizing a code

Open **M**ulti-**P**rocessing

Message **P**assing **I**nterface

Both are APIs for C/C++ and Fortran

Works in shared memory systems	Works in shared and distributed memory systems
Can use as many cores as there are in the compute node	Can use as many cores as there are in the cluster
Uses threads to split the work	Uses procesess to split the work
Threads have both private and shared variables	Processes have only private variables
Uses preprocessor directives in commented lines	Loads a library and uses specific commands



What about MPI?

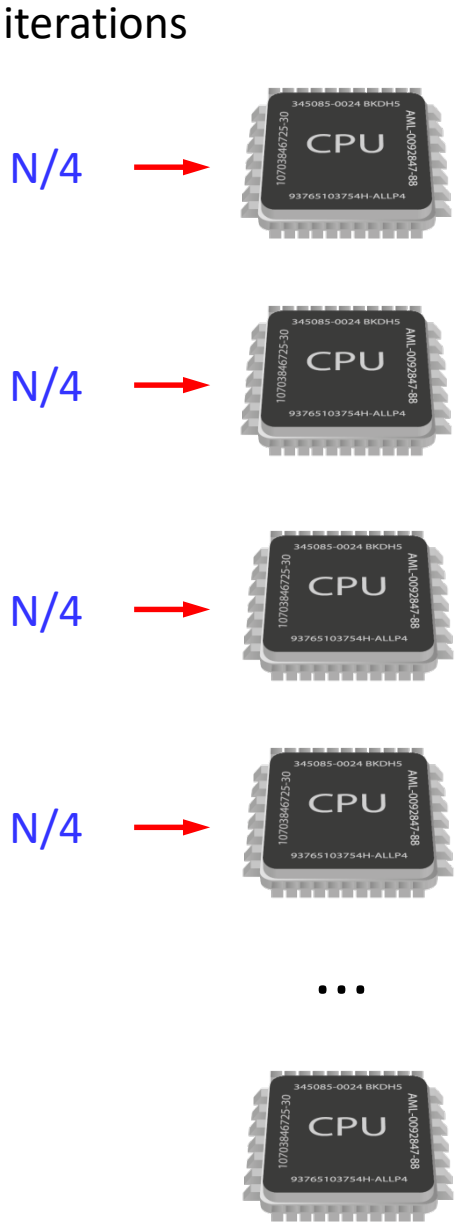
Loads a library and uses specific commands

```
include ‘mpif.h’

<variable declarations>

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,id,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)

<calculations>

call MPI_FINALIZE(ierr)
```

What about MPI?

Loads a library and uses specific commands

```
include ‘mpif.h’

<variable declarations>

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,id,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)

do i=1,N
   <calculations>
enddo

call MPI_FINALIZE(ierr)
```

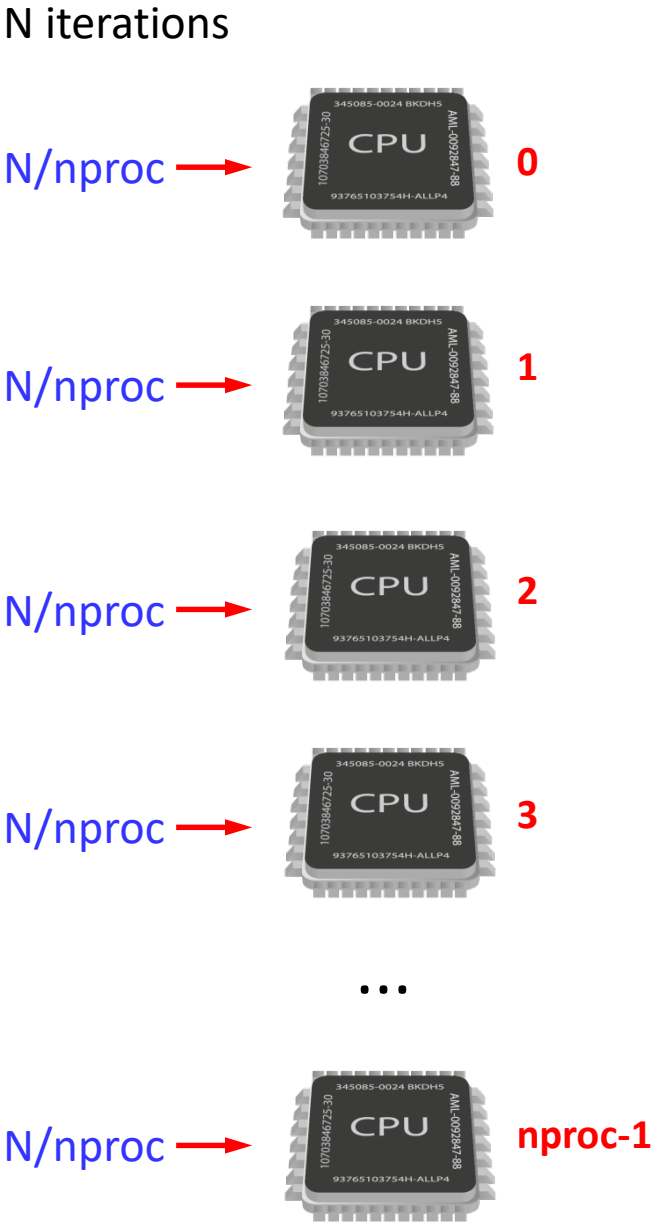


Open MPI

What about MPI?

Loads a library and uses specific commands

```
include ‘mpif.h’

<variable declarations>

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,id,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)

do i=ini,fin
   <calculations>
enddo

call MPI_FINALIZE(ierr)
```

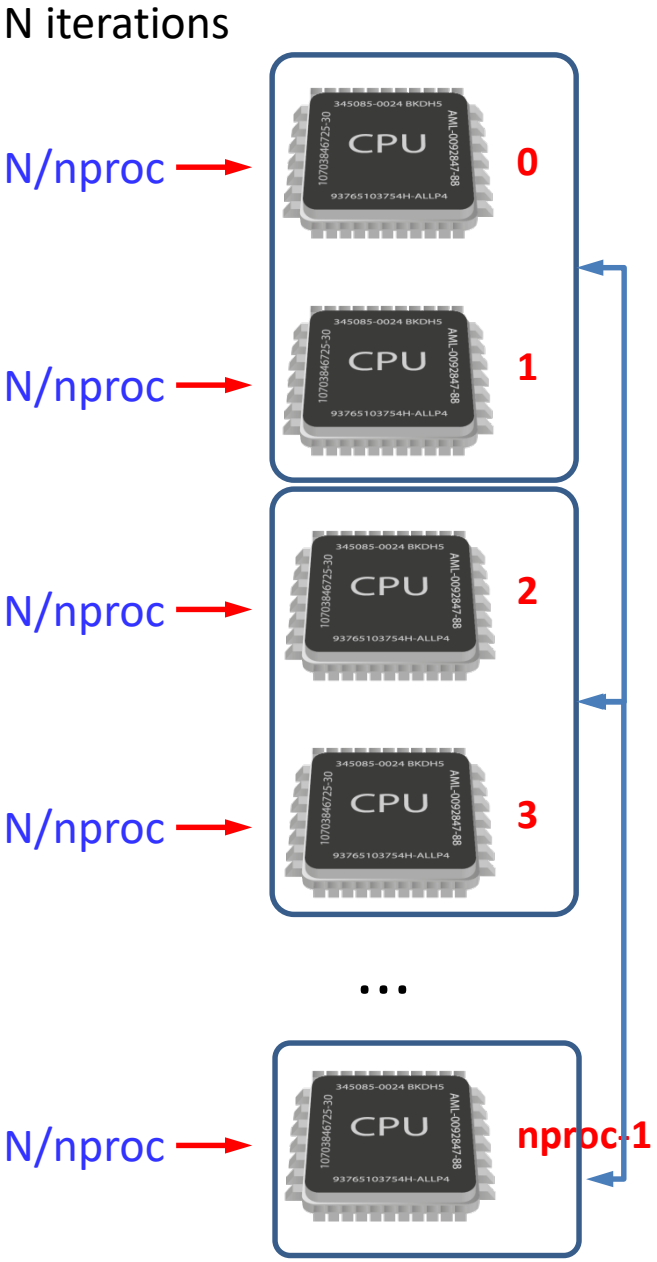


Open MPI

What about MPI?

Loads a library and uses specific commands

```
include ‘mpif.h’

<variable declarations>

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,id,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)

ini=id*(N/nproc)+1
fin=(id+1)*(N/nproc)

do i=ini,fin
   <calculations>
enddo

call MPI_FINALIZE(ierr)
```



Open MPI


345085-0024 BKDH5
CPU
10703846725-30
AML-0092847-88
93765103754H-ALLP4

0


345085-0024 BKDH5
CPU
10703846725-30
AML-0092847-88
93765103754H-ALLP4

1

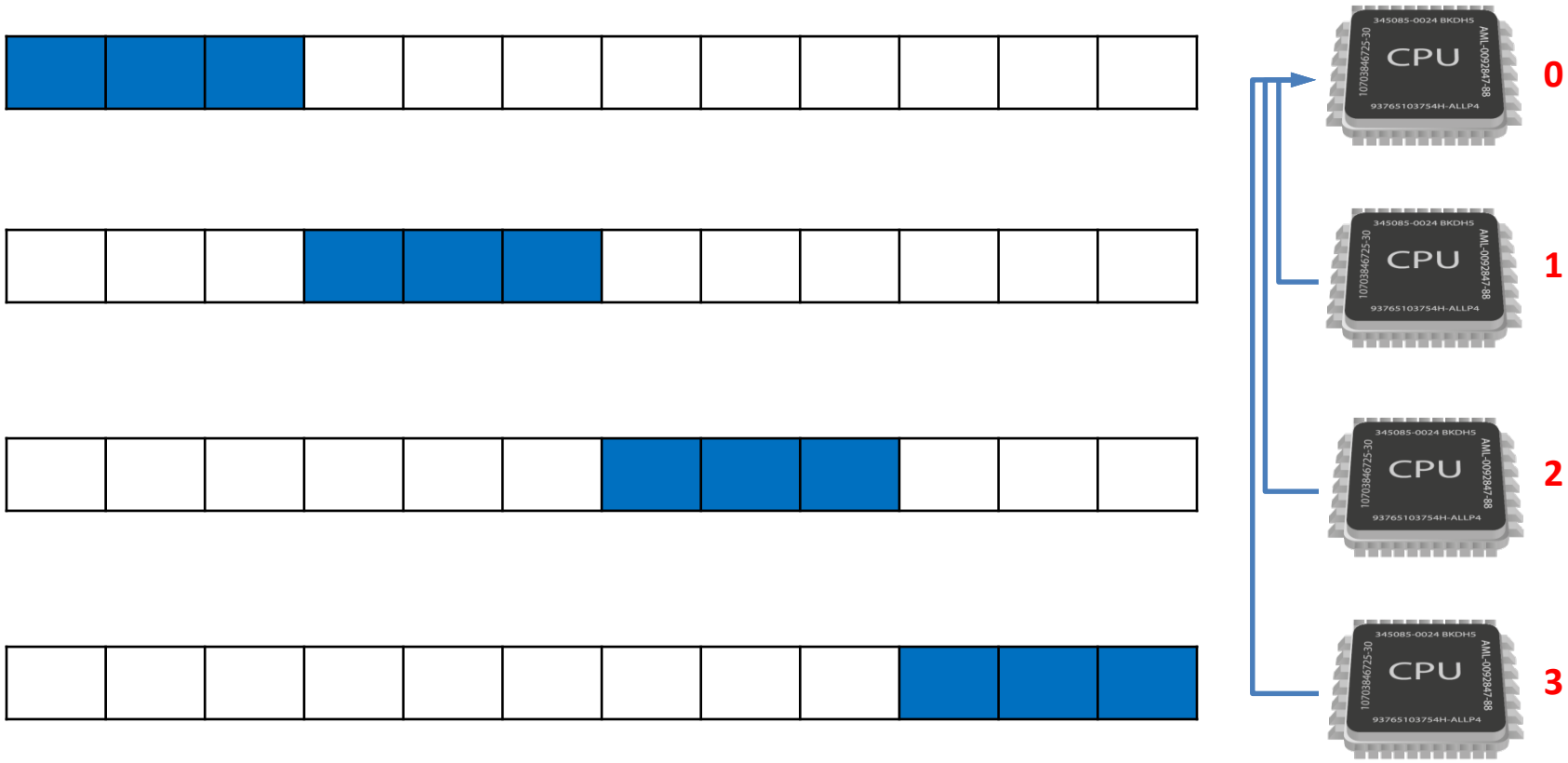

345085-0024 BKDH5
CPU
10703846725-30
AML-0092847-88
93765103754H-ALLP4

2


345085-0024 BKDH5
CPU
10703846725-30
AML-0092847-88
93765103754H-ALLP4

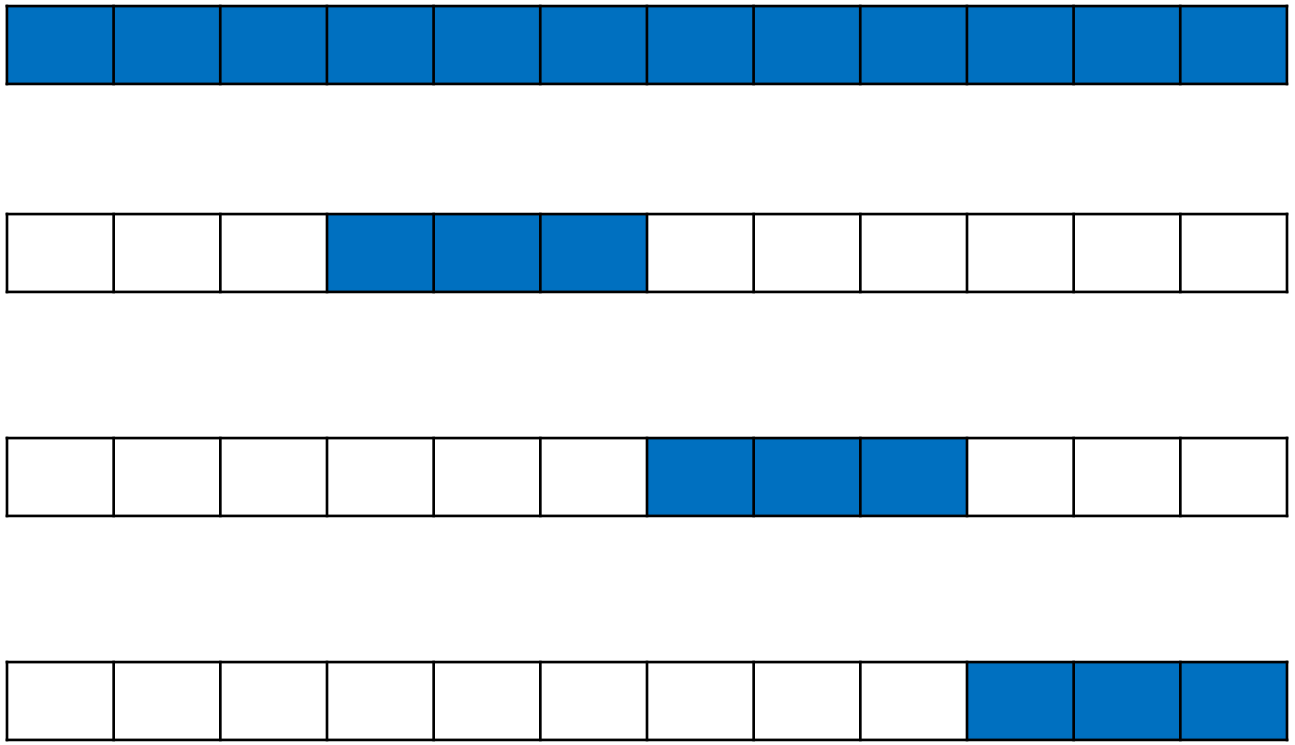
3

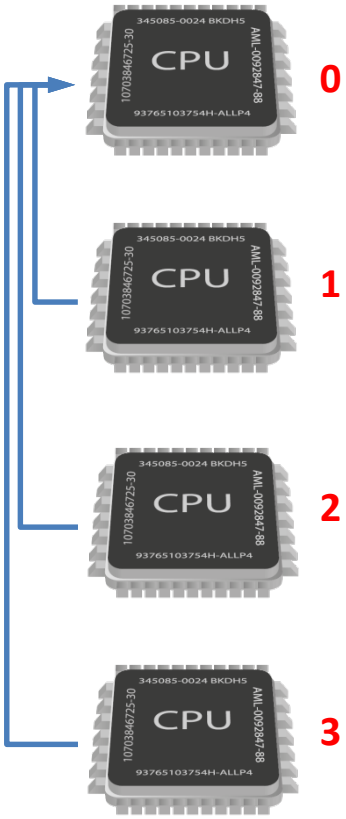

OPEN MPI



Open MPI

Only process 0 has the correct values!

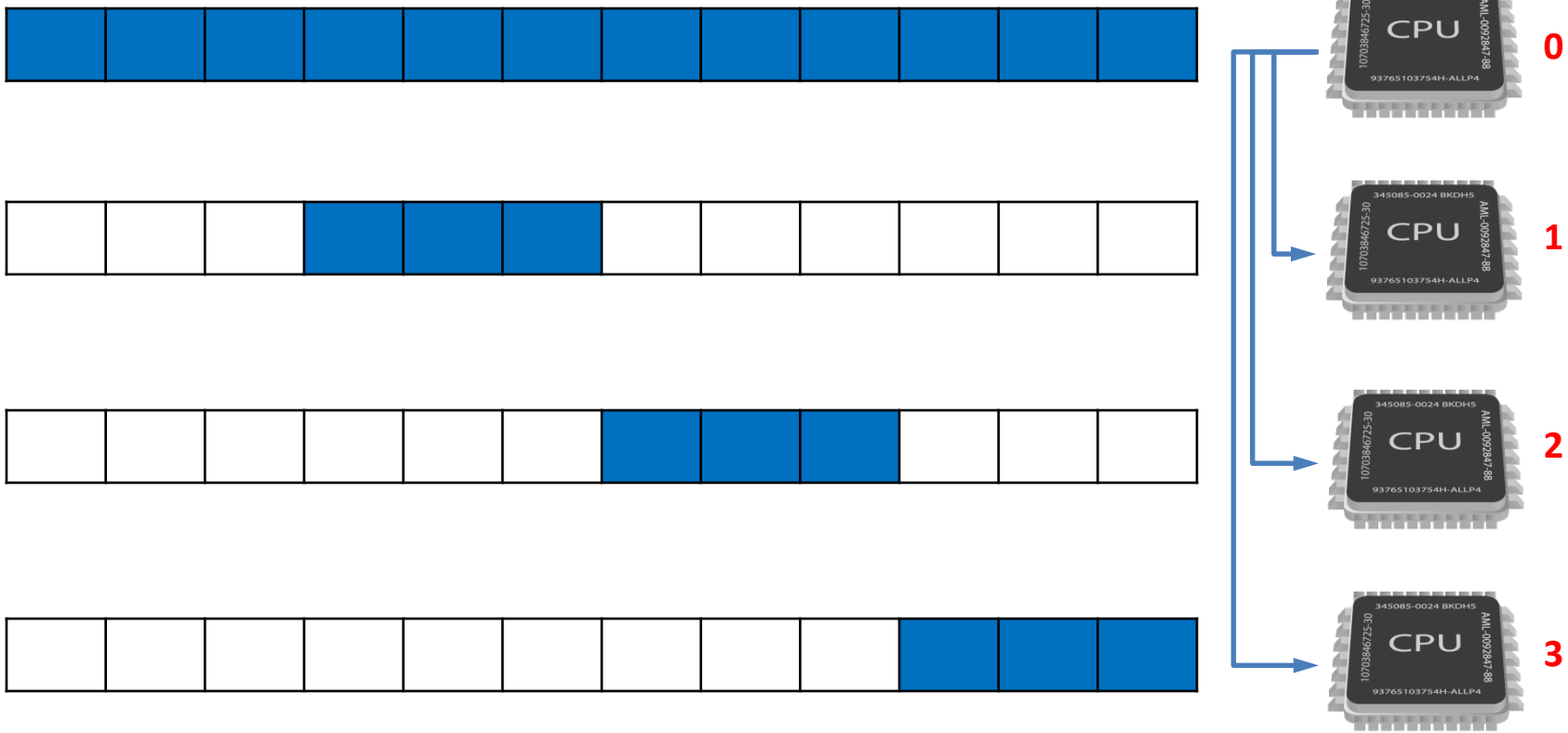





```
call MPI_REDUCE(arr,result,arraysize,<type>,<operation>,<dest_id>,MPI_WORLD_COMM,ierr)
call MPI_REDUCE(arr,result,12,MPI_INT,MPI_SUM,0,MPI_WORLD_COMM,ierr)
```

Open MPI

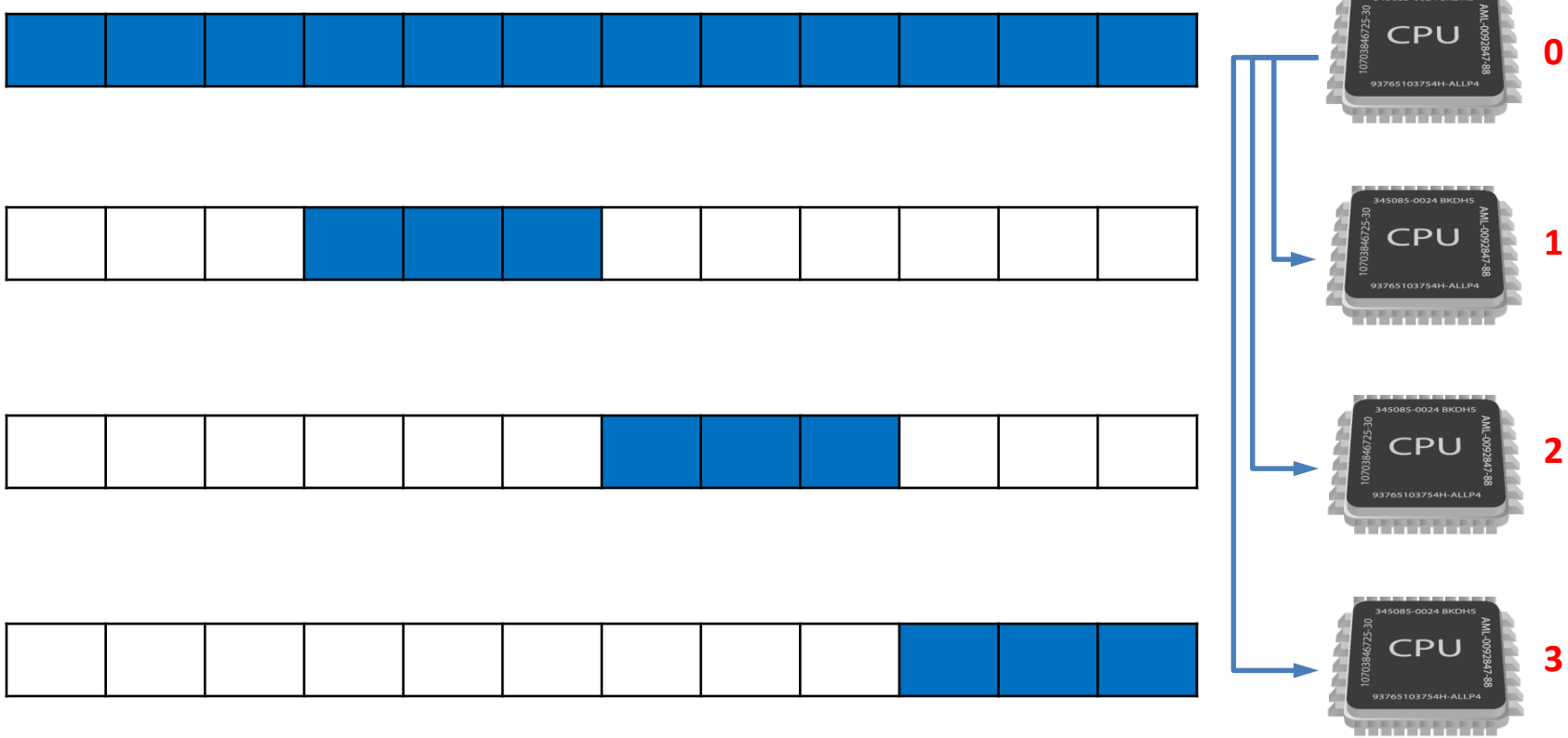
Only process 0 has the correct values!




```
call MPI_REDUCE(arr,result,arraysize,<type>,<operation>,<dest_id>,MPI_WORLD_COMM,ierr)
call MPI_REDUCE(arr,result,12,MPI_INT,MPI_SUM,0,MPI_WORLD_COMM,ierr)
```

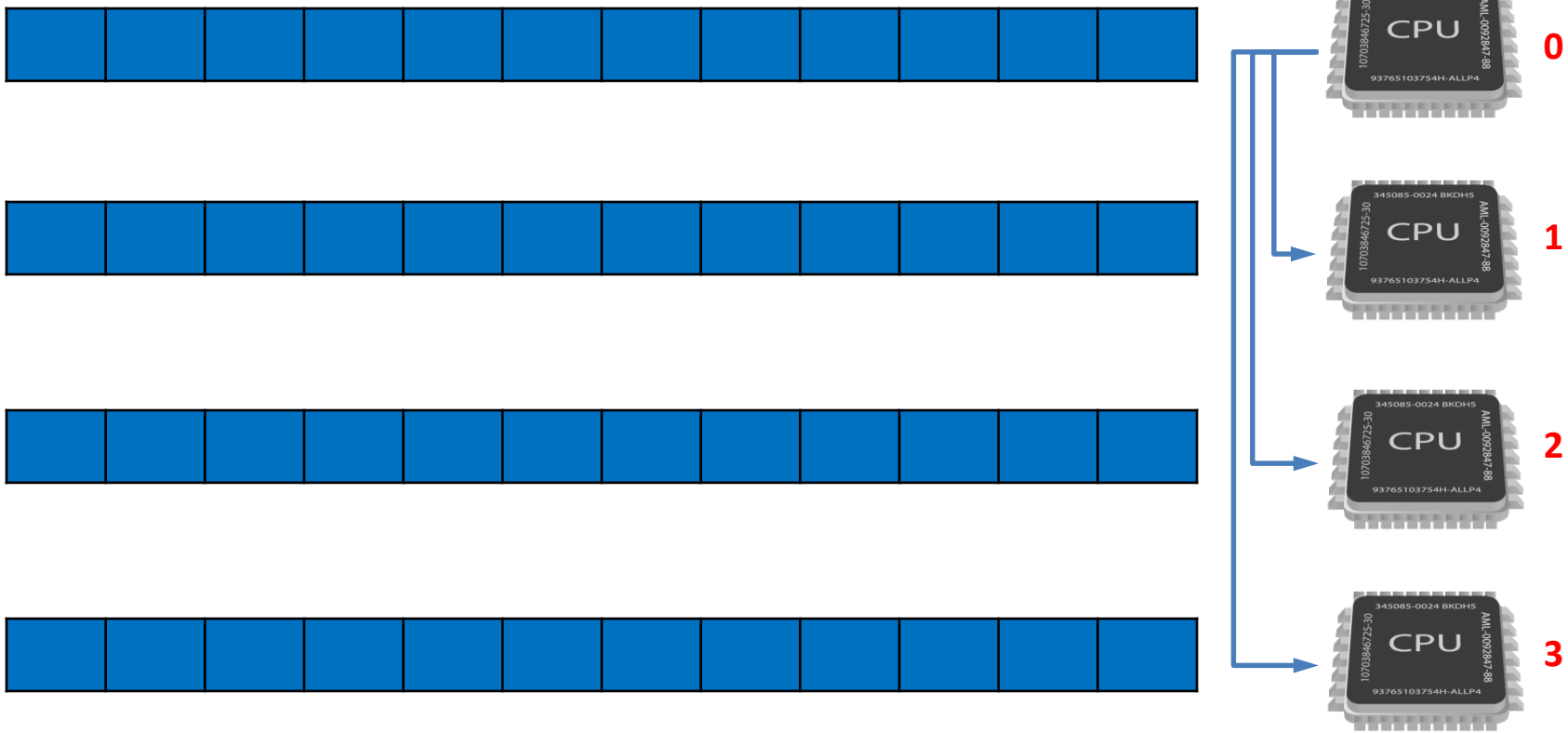
Open MPI

Only process 0 has the correct values!

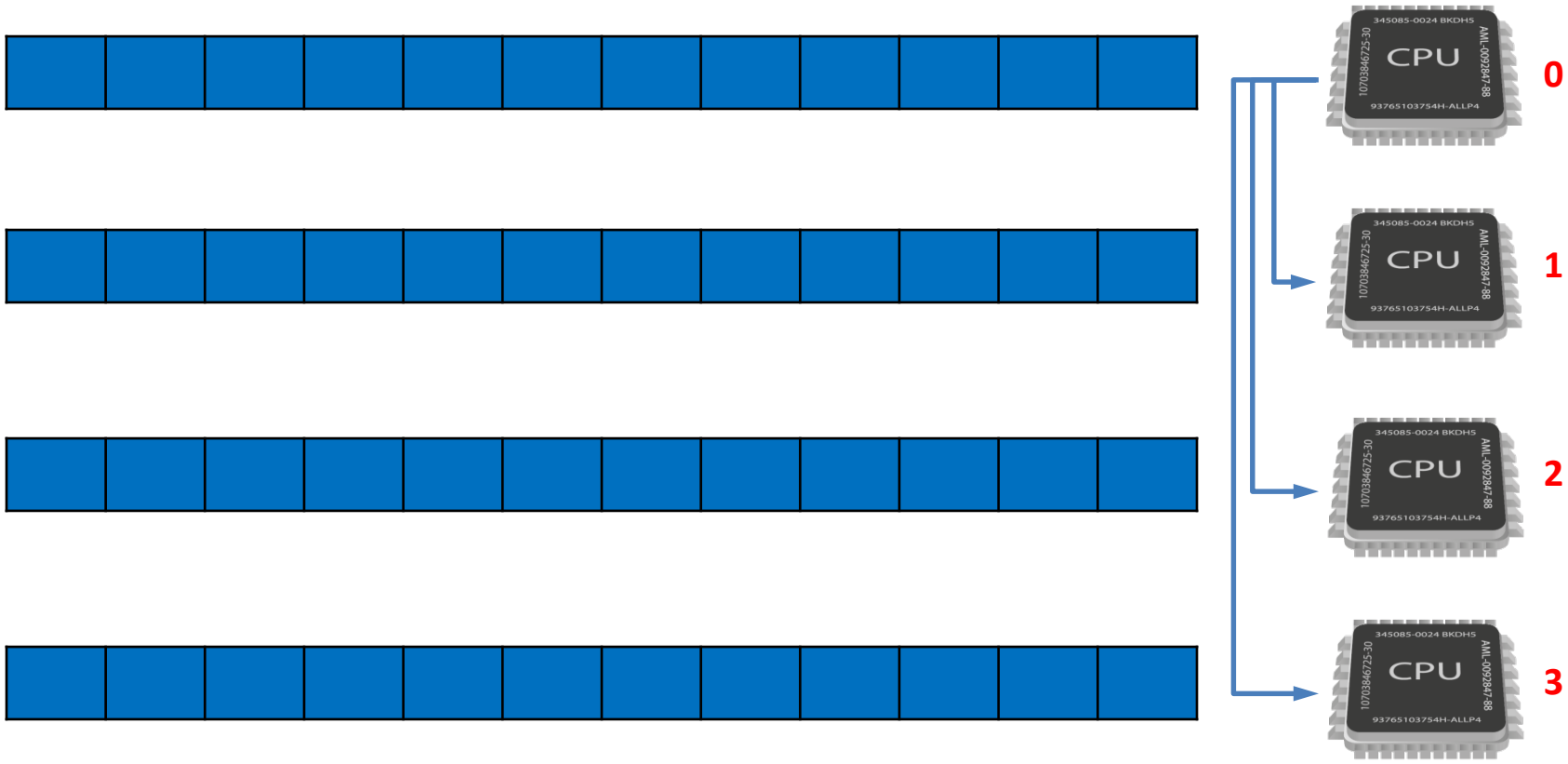



```
call MPI_REDUCE(arr,result,arraysize,<type>,<operation>,<dest_id>,MPI_WORLD_COMM,ierr)
call MPI_REDUCE(arr,result,12,MPI_INT,MPI_SUM,0,MPI_WORLD_COMM,ierr)
call MPI_BCAST(result,12,MPI_INT,0,MPI_WORLD,COMM,ierr)
```

Only process 0 has the correct values!




```
call MPI_REDUCE(arr,result,arraysize,<type>,<operation>,<dest_id>,MPI_WORLD_COMM,ierr)
call MPI_REDUCE(arr,result,12,MPI_INT,MPI_SUM,0,MPI_WORLD_COMM,ierr)
call MPI_BCAST(result,12,MPI_INT,0,MPI_WORLD,COMM,ierr)
```




```
call MPI_ALLREDUCE(arr,result,12,MPI_INT,MPI_SUM,MPI_WORLD_COMM,ierr)
```

Open MPI

Example: Numerical integration

```
include 'mpif.h'

call MPI_INIT(ierr); comm = MPI_COMM_WORLD
call MPI_COMM_RANK(comm, rank, ierr)
call MPI_COMM_SIZE(comm, size, ierr)
```

Initialize MPI

FORTRAN

Example: Numerical integration

```
include 'mpif.h'

call MPI_INIT(ierr); comm = MPI_COMM_WORLD
call MPI_COMM_RANK(comm, rank, ierr)
call MPI_COMM_SIZE(comm, size, ierr)

totalsteps = 1000000
steps = totalsteps/size

ini = rank * steps
fin = (rank+1) * steps

if(fin.ge.totalsteps) then
    fin = totalsteps
endif
```

Initialize MPI

Define the range

FORTRAN

Example: Numerical integration

```
include 'mpif.h'

call MPI_INIT(ierr); comm = MPI_COMM_WORLD
call MPI_COMM_RANK(comm, rank, ierr)
call MPI_COMM_SIZE(comm, size, ierr)

totalsteps = 1000000
steps = totalsteps/size

ini = rank * steps
fin = (rank+1) * steps

if(fin.ge.totalsteps) then
    fin = totalsteps
endif

dx = 1.d0 / totalsteps
x = -0.5d0 * dx

localSum = 0.d0
do i = ini, fin
    x = (i-0.5d0)*dx
    localSum = localSum + 4.d0 / (1.d0 + x*x)
enddo
localSum = localSum * dx
```

Initialize MPI

Define the range

FORTRAN

Do the computation

Example: Numerical integration

```
include 'mpif.h'

call MPI_INIT(ierr); comm = MPI_COMM_WORLD
call MPI_COMM_RANK(comm, rank, ierr)
call MPI_COMM_SIZE(comm, size, ierr)

totalsteps = 1000000
steps = totalsteps/size

ini = rank * steps
fin = (rank+1) * steps

if(fin.ge.totalsteps) then
    fin = totalsteps
endif

dx = 1.d0 / totalsteps
x = -0.5d0 * dx

localSum = 0.d0
do i = ini, fin
    x = (i-0.5d0)*dx
    localSum = localSum + 4.d0 / (1.d0 + x*x)
enddo
localSum = localSum * dx

call MPI_ALLREDUCE(localSum,globalSum,1,MPI_DOUBLE_PRECISION,MPI_SUM,comm,ierr)

write(*,*) localSum, globalSum
```

Initialize MPI

Define the range

Do the computation

Sum the data across nodes

FORTRAN

Example: Numerical integration

```
include 'mpif.h'

call MPI_INIT(ierr); comm = MPI_COMM_WORLD
call MPI_COMM_RANK(comm, rank, ierr)
call MPI_COMM_SIZE(comm, size, ierr)

totalsteps = 1000000
steps = totalsteps/size

ini = rank * steps
fin = (rank+1) * steps

if(fin.ge.totalsteps) then
    fin = totalsteps
endif

dx = 1.d0 / totalsteps
x = -0.5d0 * dx

localSum = 0.d0
do i = ini, fin
    x = (i-0.5d0)*dx
    localSum = localSum + 4.d0 / (1.d0 + x*x)
enddo
localSum = localSum * dx

call MPI_ALLREDUCE(localSum,globalSum,1,MPI_DOUBLE_PRECISION,MPI_SUM,comm,ierr)

write(*,*) localSum, globalSum

call MPI_FINALIZE(ierr)
```

Initialize MPI

Define the range

FORTRAN

Do the computation

Sum the data across nodes

Don’t forget to Finalize MPI

Example: Numerical integration

```python
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

totalsteps = 1000000
steps = totalsteps/size

ini = rank * steps
fin = (rank+1) * steps

if fin > totalsteps:
      fin = totalsteps

dx = 1./totalsteps
x = -0.5*dx

localSum = 0.
for i in range(int(ini), int(fin)):
      x = (i-0.5)*dx
      localSum = localSum + 4./(1. + x*x)

localSum = localSum * dx

globalSum = comm.allreduce(localSum, op=MPI.SUM)

print(localSum, globalSum)
```

Initialize MPI

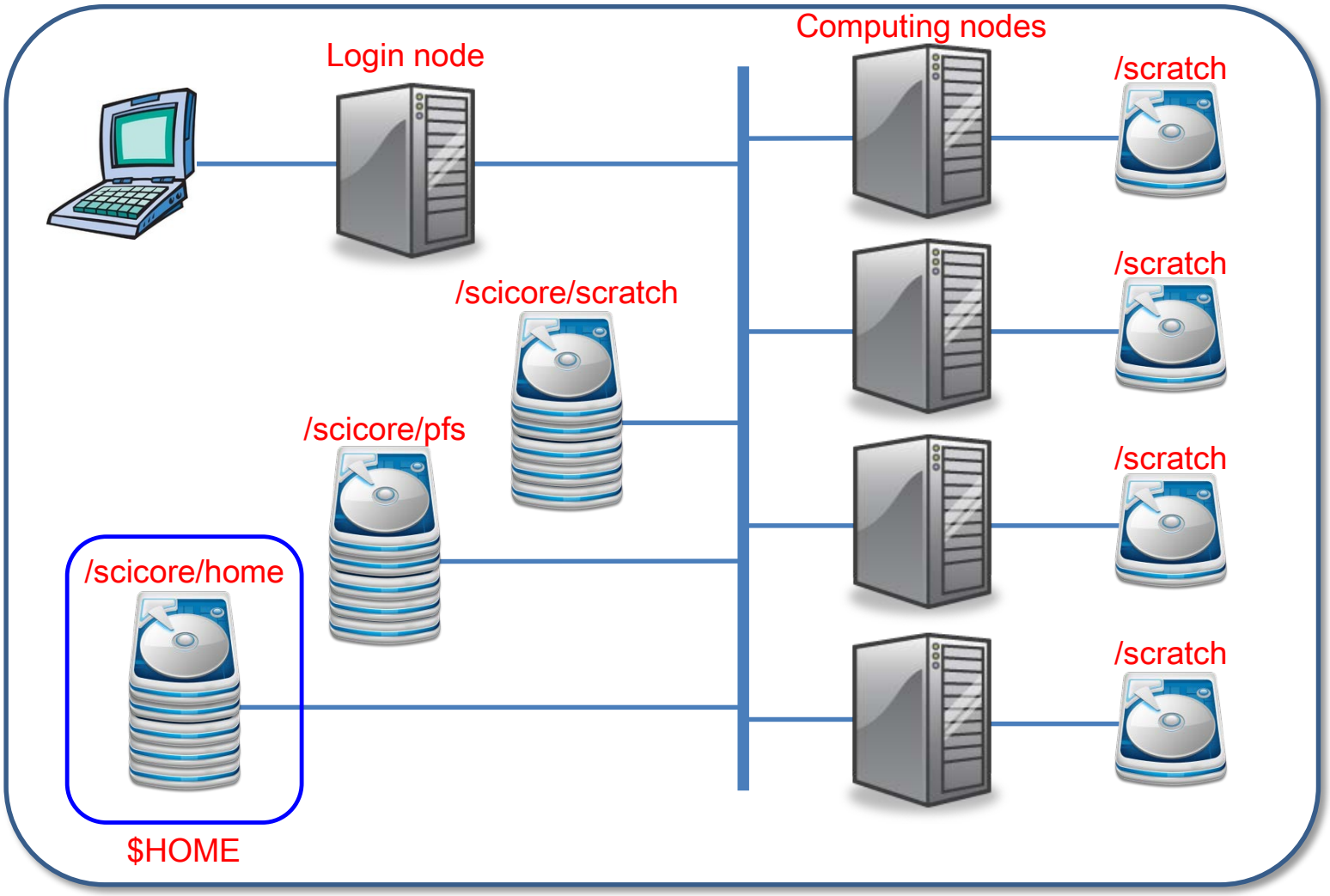
Define the range

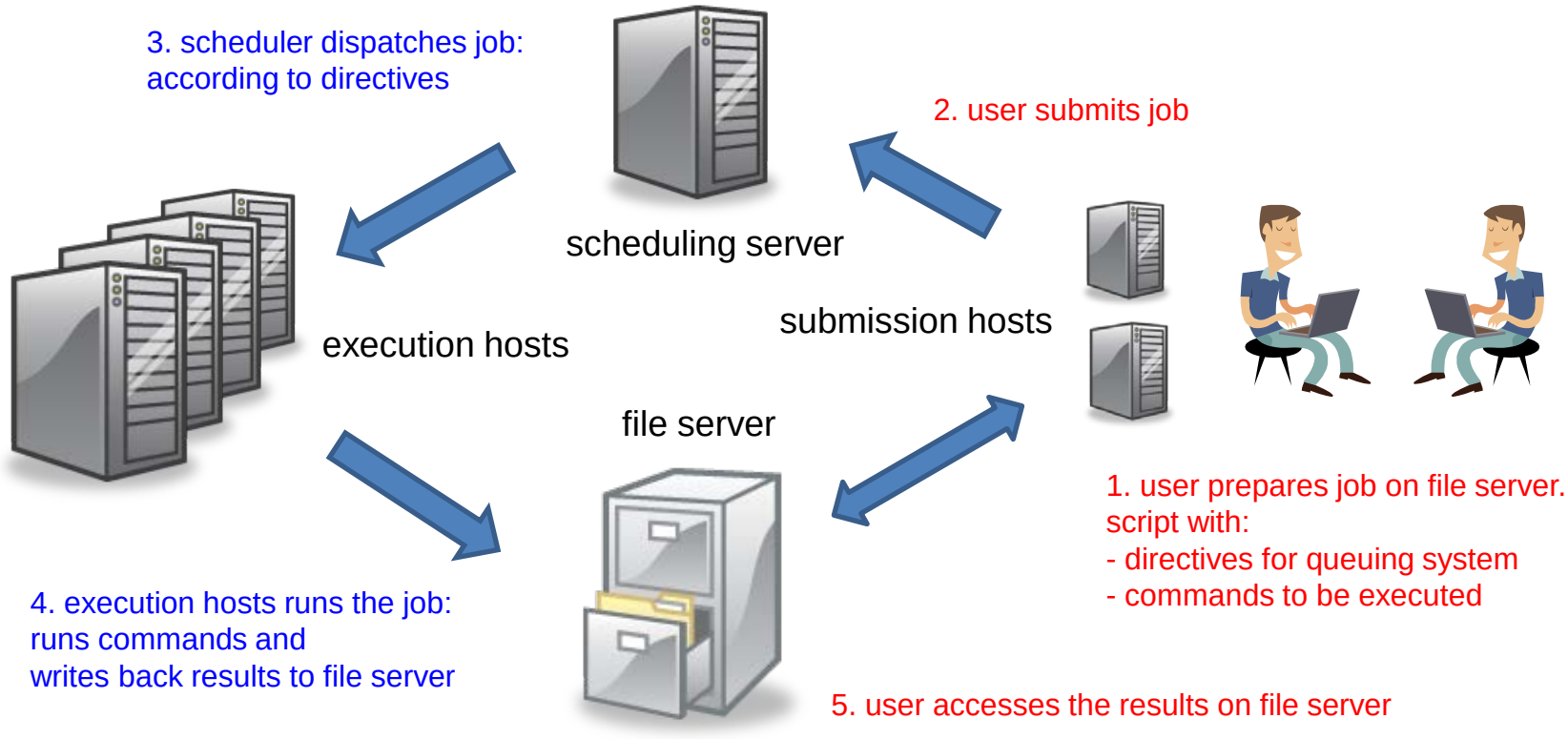
Python

Do the computation

Sum the data across nodes

Mpi4py tutorial at: https://mpi4py.readthedocs.io/en/stable/tutorial.html


Computing nodes
Login node
/scratch
/scratch
/scicore/scratch
/scicore/pfs
/scratch
/scicore/home
/scratch
$HOME


3. scheduler dispatches job:
according to directives
scheduling server
2. user submits job
execution hosts
submission hosts
file server
1. user prepares job on file server.
script with:
- directives for queuing system
- commands to be executed
4. execution hosts runs the job:
runs commands and
writes back results to file server
5. user accesses the results on file server
User takes care
Cluster takes care

Lest’s play “Spot the Differences”

Lest's play "Spot the Differences"

Which script launches an OpenMP job and which one an MPI?

```
#!/bin/bash

#SBATCH --job-name=myJob
#SBATCH --cpus-per-task=8
#SBATCH --mem=3.5G
#SBATCH --time=05:00:00
#SBATCH --qos=6hour
#SBATCH --output=/path/to/stdout/folder
#SBATCH --error=/path/to/stderr/folder
#SBATCH --mail-type=END,FAIL,TIME_LIMIT
#SBATCH --mail-user=mailaddress@unibas.ch

# Tell OpenMP how many threads to use
#########################
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

# load your required modules
########################
module load <module name>
…

# and here goes your command line
./my_openmp_program

# Some third party programs include a threading
# option, configuration or environment variable
```

```
#!/bin/bash

#SBATCH --job-name=myJob
#SBATCH --ntasks=8
#SBATCH --mem=3.5G
#SBATCH --time=05:00:00
#SBATCH --qos=6hour
#SBATCH --output=/path/to/stdout/folder
#SBATCH --error=/path/to/stderr/folder
#SBATCH --mail-type=END,FAIL,TIME_LIMIT
#SBATCH --mail-user=mailaddress@unibas.ch

# Tell OpenMP how many threads to use
#########################
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

# load your required modules
########################
module load <module name>
…

# and here goes your command line
srun ./my_openmp_program

# Some third party programs include a threading
# option, configuration or environment variable
```

Lest's play "Spot the Differences"

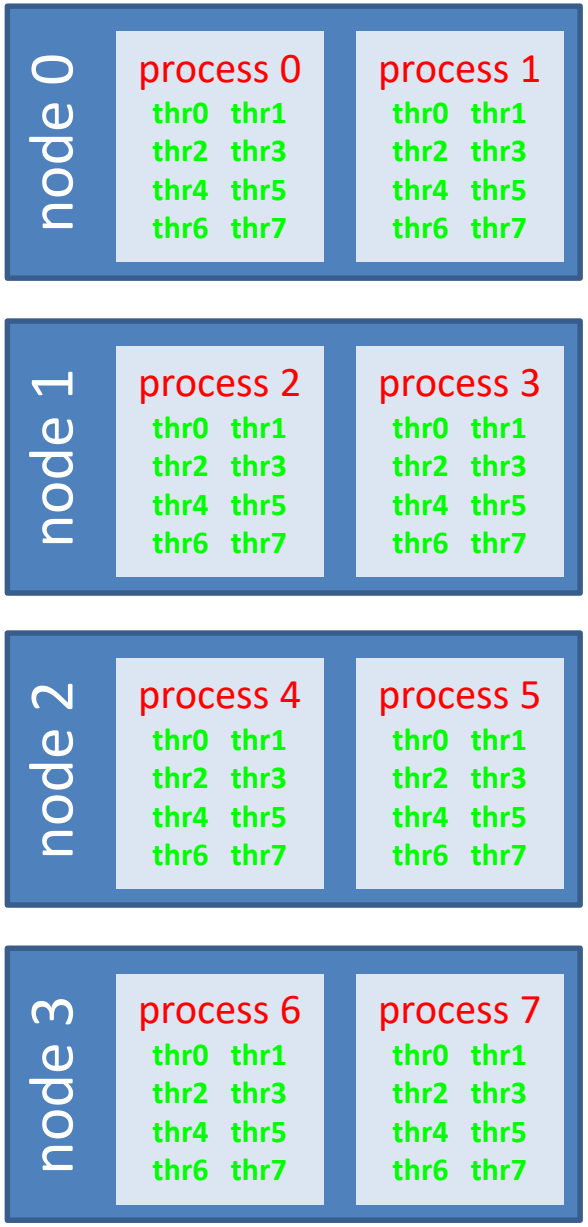
Which script launches an OpenMP job and which one an MPI?

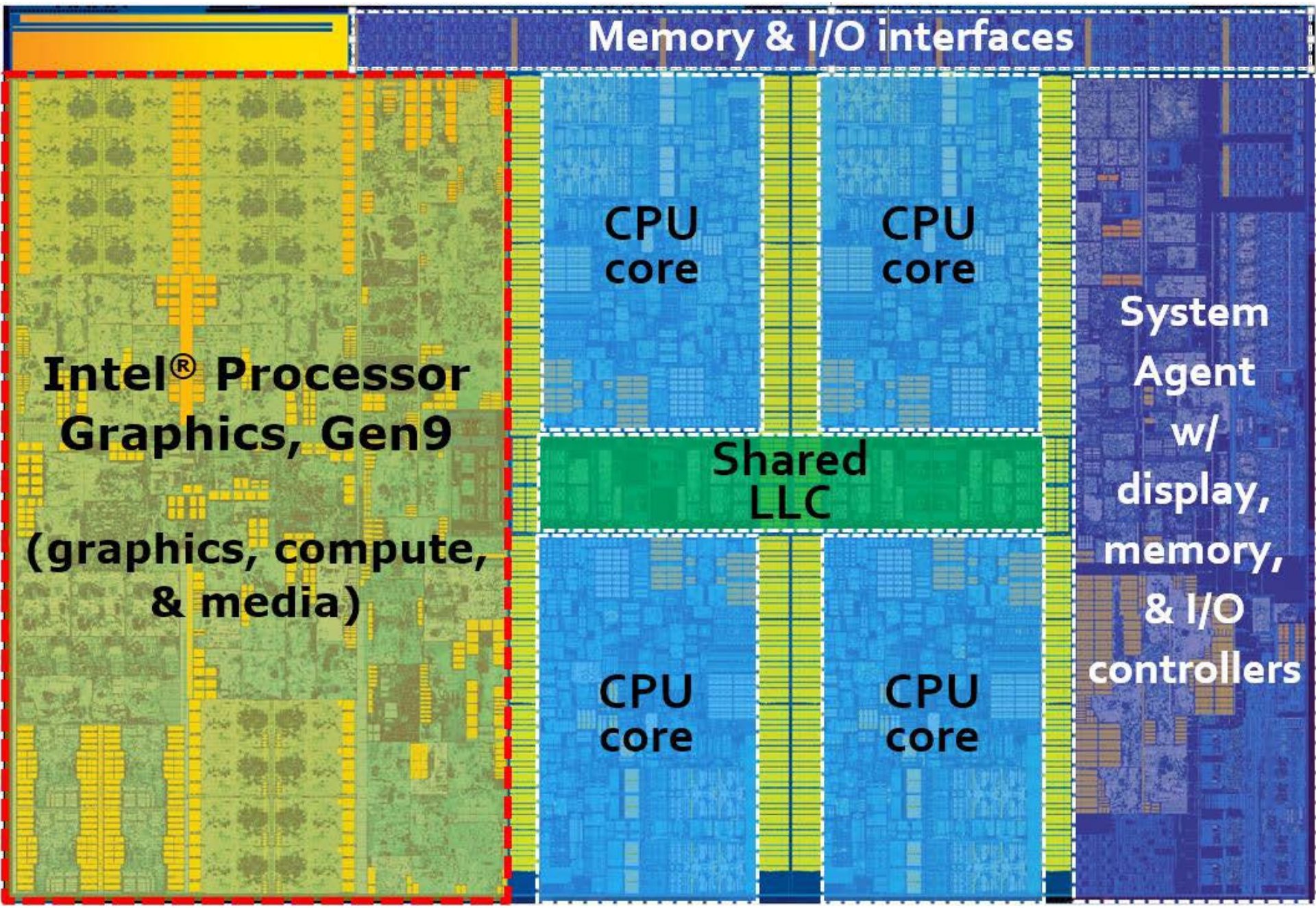

```
#!/bin/bash

#SBATCH --job-name=myJob
#SBATCH --cpus-per-task=8
#SBATCH --mem=3.5G
#SBATCH --time=05:00:00
#SBATCH --qos=6hour
#SBATCH --output=/path/to/stdout/folder
#SBATCH --error=/path/to/stderr/folder
#SBATCH --mail-type=END,FAIL,TIME_LIMIT
#SBATCH --mail-user=mailaddress@unibas.ch

# Tell OpenMP how many threads to use
#########################
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

# load your required modules
########################
module load <module name>
…

# and here goes your command line
./my_openmp_program

# Some third party programs include a threading
# option, configuration or environment variable
```


```
#!/bin/bash

#SBATCH --job-name=myJob
#SBATCH --ntasks=8
#SBATCH --mem=3.5G
#SBATCH --time=05:00:00
#SBATCH --qos=6hour
#SBATCH --output=/path/to/stdout/folder
#SBATCH --error=/path/to/stderr/folder
#SBATCH --mail-type=END,FAIL,TIME_LIMIT
#SBATCH --mail-user=mailaddress@unibas.ch

# Tell OpenMP how many threads to use
#########################
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

# load your required modules
########################
module load <module name>
…

# and here goes your command line
srun ./my_openmp_program

# Some third party programs include a threading
# option, configuration or environment variable
```

How do I launch an hybrid job?

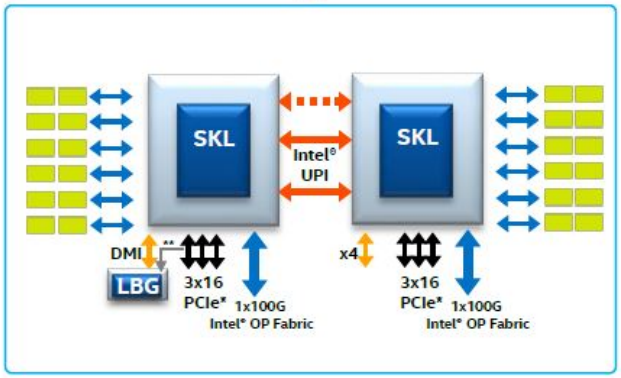
```
#!/bin/bash

#SBATCH --job-name=myJob
#SBATCH --ntasks=?
#SBATCH --tasks-per-node=?
#SBATCH --cpus-per-task=?
#SBATCH --mem=3.5G
#SBATCH --time=05:00:00
#SBATCH --qos=6hour
#SBATCH --output=/path/to/stdout/folder
#SBATCH --error=/path/to/stderr/folder
#SBATCH --mail-type=END,FAIL,TIME_LIMIT
#SBATCH --mail-user=mailaddress@unibas.ch

# Tell OpenMP how many threads to use
############################
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

# load your required modules
############################
module load <module name>
…

# and here goes your command line
srun my_openmp_program

# Some third party programs include a threading
# option, configuration or environment variable
```




Memory & I/O interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers

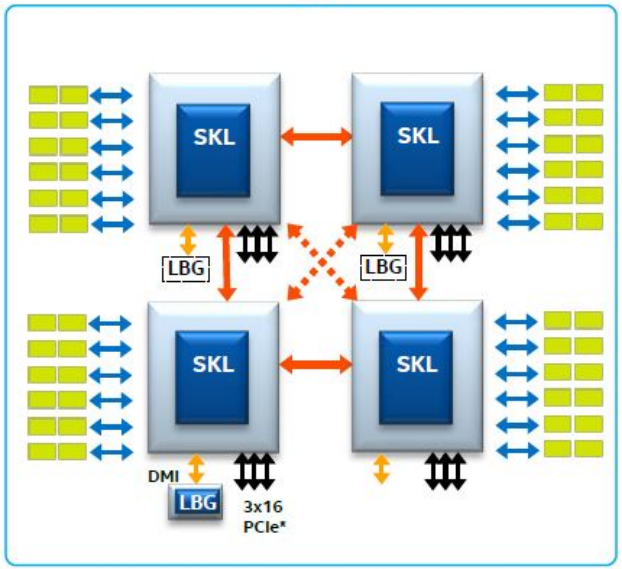
Platform Topologies

2S Configurations



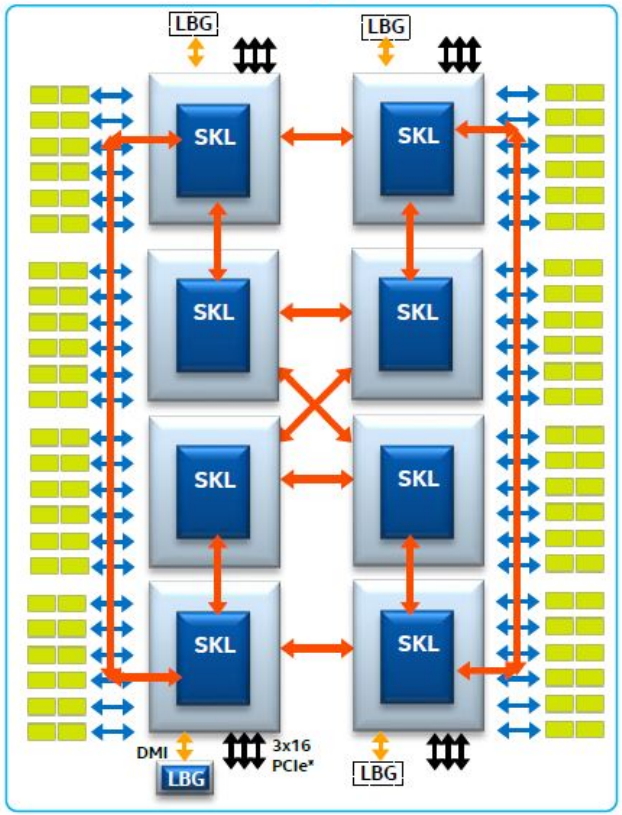
(2S-2UPI & 2S-3UPI shown)

4S Configurations



(4S-2UPI & 4S-3UPI shown)

8S Configuration



INTEL® XEON® SCALABLE PROCESSOR SUPPORTS CONFIGURATIONS RANGING FROM 2S-2UPI TO 8S

When using parallel programs you must be more aware of the hardware topology!

numactl --hardware

```
[cabezon@login10 ~]$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 32742 MB
node 0 free: 188 MB
node 1 cpus: 8 9 10 11 12 13 14 15
node 1 size: 32768 MB
node 1 free: 10518 MB
node distances:
node   0   1
  0:  10  11
  1:  11  10
```

← Relative distances

lscpu

```
[cabezon@login10 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    1
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 45
Model name:            Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz
Stepping:              7
CPU MHz:               1298.578
BogoMIPS:              5205.23
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
NUMA node0 CPU(s):     0-7
NUMA node1 CPU(s):     8-15
```

Two sockets!

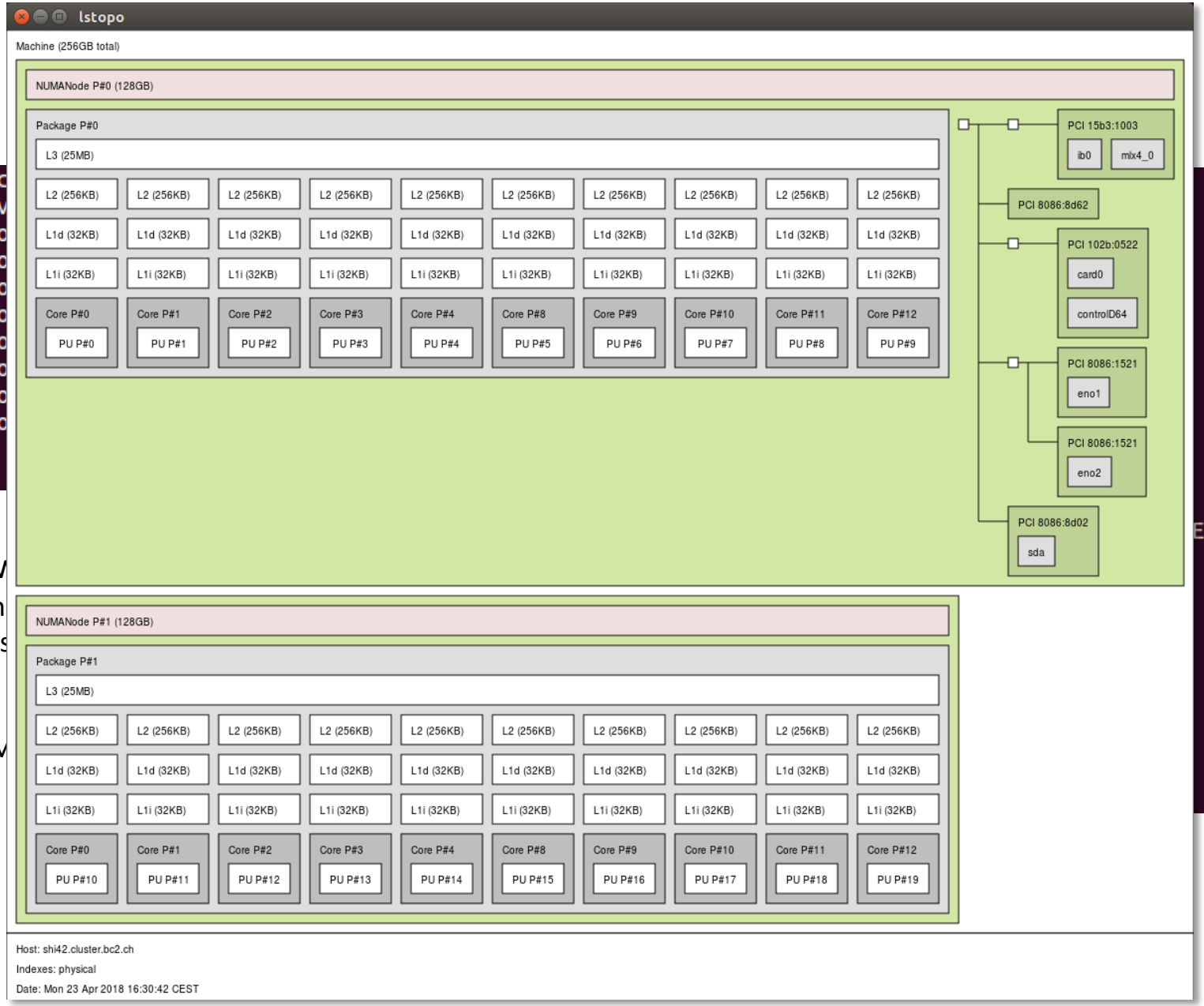
This is a NUMA (non-uniform memory access) system.

vs.

SMP (symmetric multi-processor):

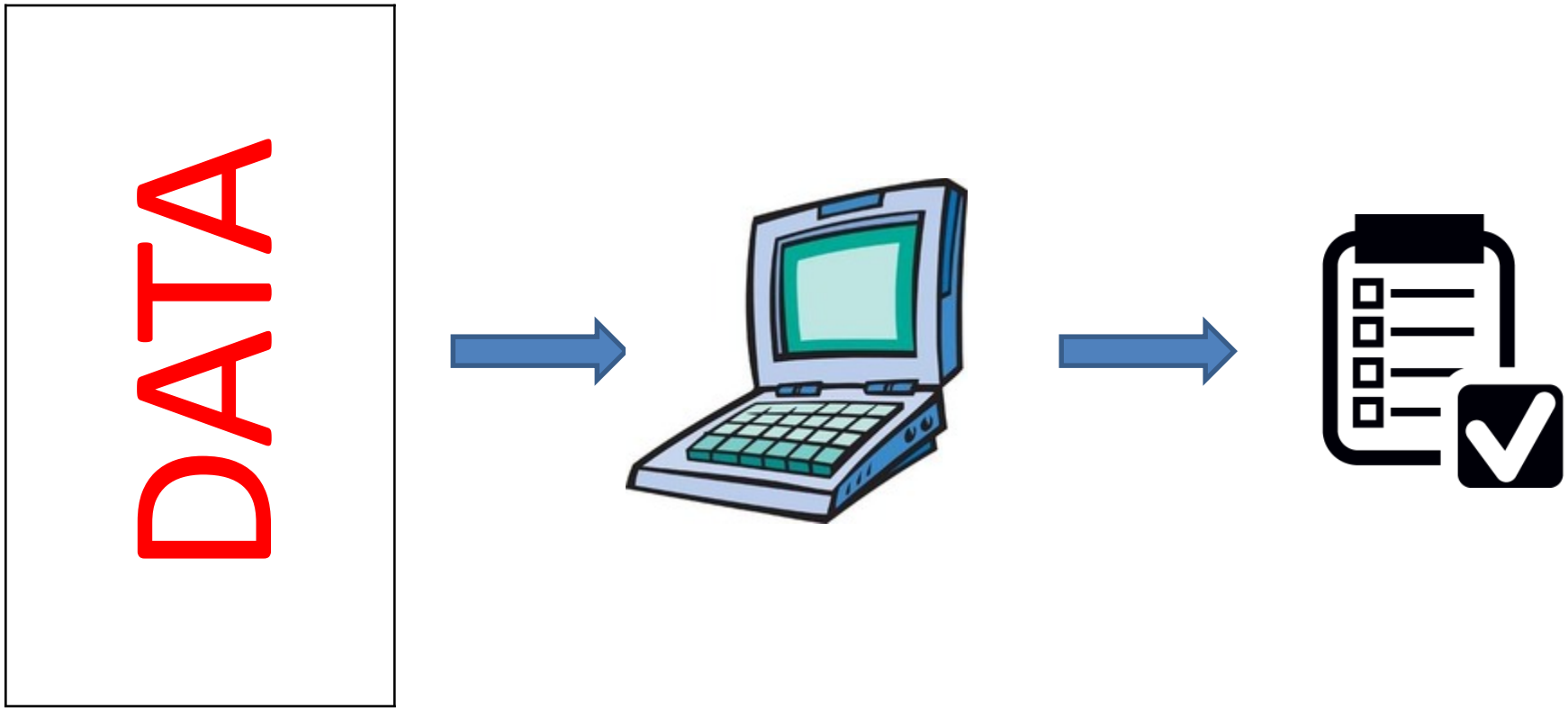
```
ruben@jarvis:~$ numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 7885 MB
node 0 free: 252 MB
node distances:
node   0
  0:  10
```

lstopo
(from hwloc package)

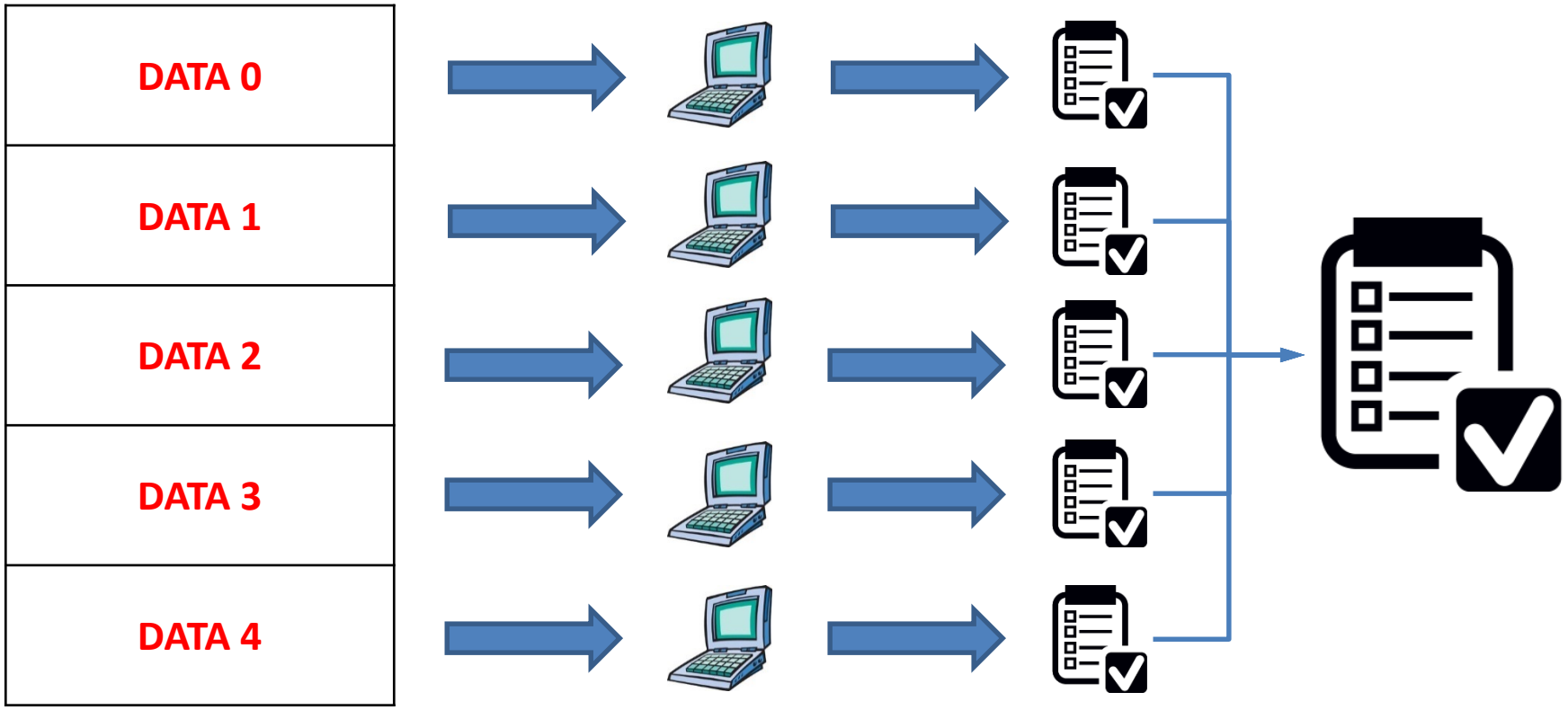


E5-2670 0 @ 2.60GHz

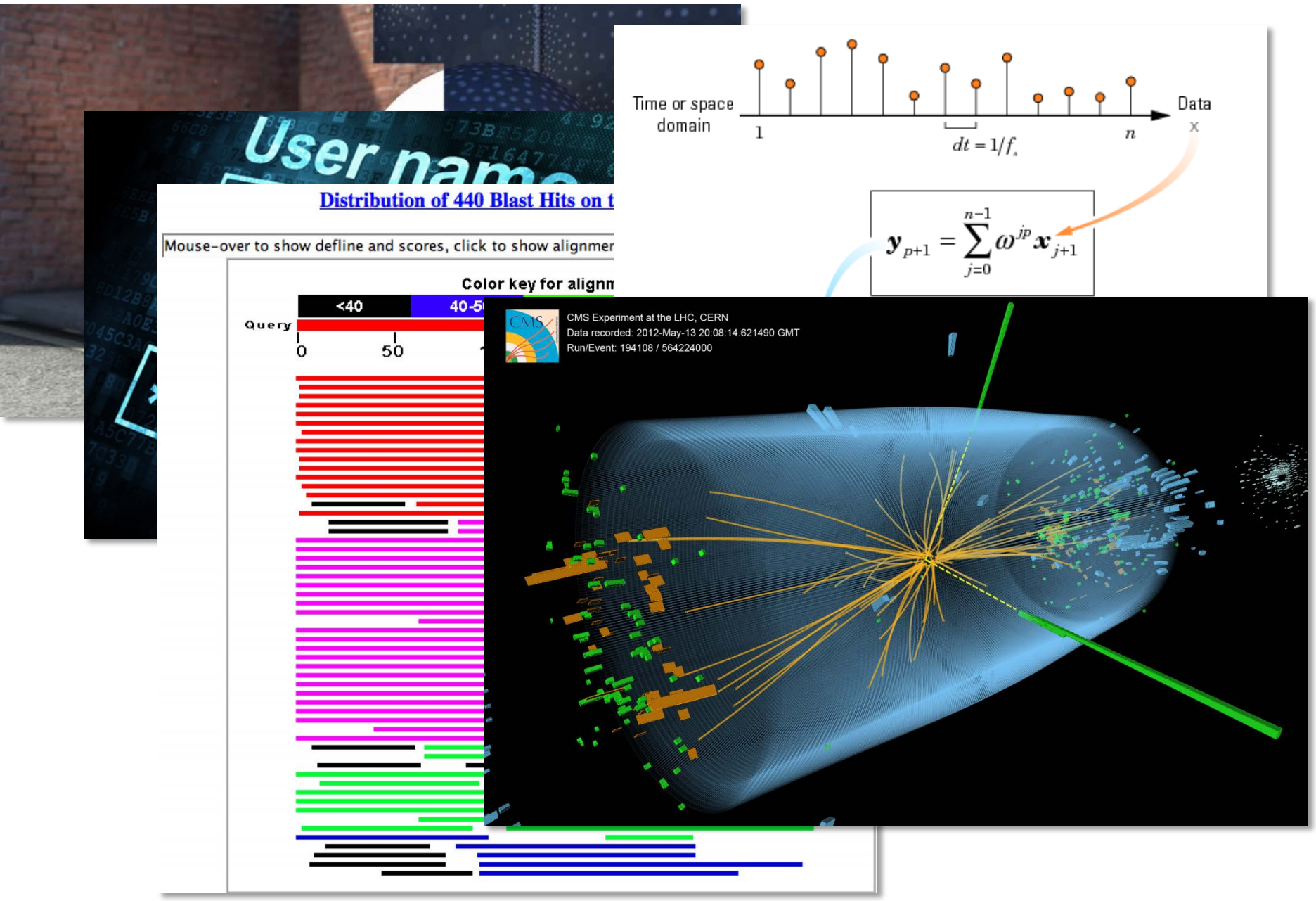
Array jobs (embarrasingly parallel calculations)



Array jobs (embarrasingly parallel calculations)



Perfect scaling: 5x speed-up in this example.


User name
Distribution of 440 Blast Hits on t
Mouse-over to show defline and scores, click to show alignmer
Color key for alignm
<40
40-5
Query
0
50
Time or space domain
Data
x
1
n
$dt = 1/f_s$
$y_{p+1} = \sum_{j=0}^{n-1} \omega^{jp} x_{j+1}$
CMS Experiment at the LHC, CERN
Data recorded: 2012-May-13 20:08:14.621490 GMT
Run/Event: 194108 / 564224000

Array jobs (embarrasingly parallel calculations)

You should use array jobs:

- You only write one script
- You don't have to worry about deleting thousands of scripts
- If you submit and array job, and realize that you made a mistake, you only have one jod id to qdel, instead of 100s.
- You put less burden on the head node.

Script for submitting an array job

```
#!/bin/bash

#SBATCH –-job-name=myJob
#SBATCH –-cpus-per-task=1
#SBATCH --mem=3.5G
#SBATCH --time=05:00:00
#SBATCH --qos=6hour
#SBATCH --output=/path/to/stdout/folder
#SBATCH --error=/path/to/stderr/folder
#SBATCH --mail-type=END,FAIL,TIME_LIMIT
#SBATCH --mail-user=mailaddress@unibas.ch

# Tell SLURM that this is an array of jobs
##########################
#SBATCH --array=1-50%5
```

This will launch 50 tasks to be numbered from 1 to 50.

Optionally, we can limit the amount of tasks running simultaneously.

```
# load your required modules
########################
module load Java


# and here goes your command line
$(head –n $SLURM_ARRAY_TASK_ID commands.cmd | tail -1)
```

When a task in the array job is sent to a compute node, its task number is stored in the variable `SLURM_ARRAY_TASK_ID`, so we can use it to select the input and output data that we want.

Array jobs (embarrasingly parallel calculations)

You should use array jobs:

- You only write one script
- You don't have to worry about deleting thousands of scripts
- If you submit and array job, and realize that you made a mistake, you only have one jod id to qdel, instead of 100s.
- You put less burden on the head node.

Script for submitting an array job

```
#!/bin/bash

#SBATCH –-job-name=myJob
#SBATCH –-cpus-per-task=1
#SBATCH --mem=3.5G
#SBATCH --time=05:00:00
#SBATCH --qos=6hour
#SBATCH --output=/path/to/stdout/folder
#SBATCH --error=/path/to/stderr/folder
#SBATCH --mail-type=END,FAIL,TIME_LIMIT
#SBATCH --mail-user=mailaddress@unibas.ch

# Tell SLURM that this is an array of jobs
##########################
#SBATCH --array=1-50%5

# load your required modules
########################
module load Java


# and here goes your command line
$(head –n $SLURM_ARRAY_TASK_ID commands.cmd | tail -1)
```

This will launch 50 tasks to be numbered from 1 to 50.

Optionally, we can limit the amount of tasks running simultaneously.

When a task in the array job is sent to a compute node, its task number is stored in the variable `SLURM_ARRAY_TASK_ID`, so we can use it to select the input and output data that we want.


Thank you
for your attention