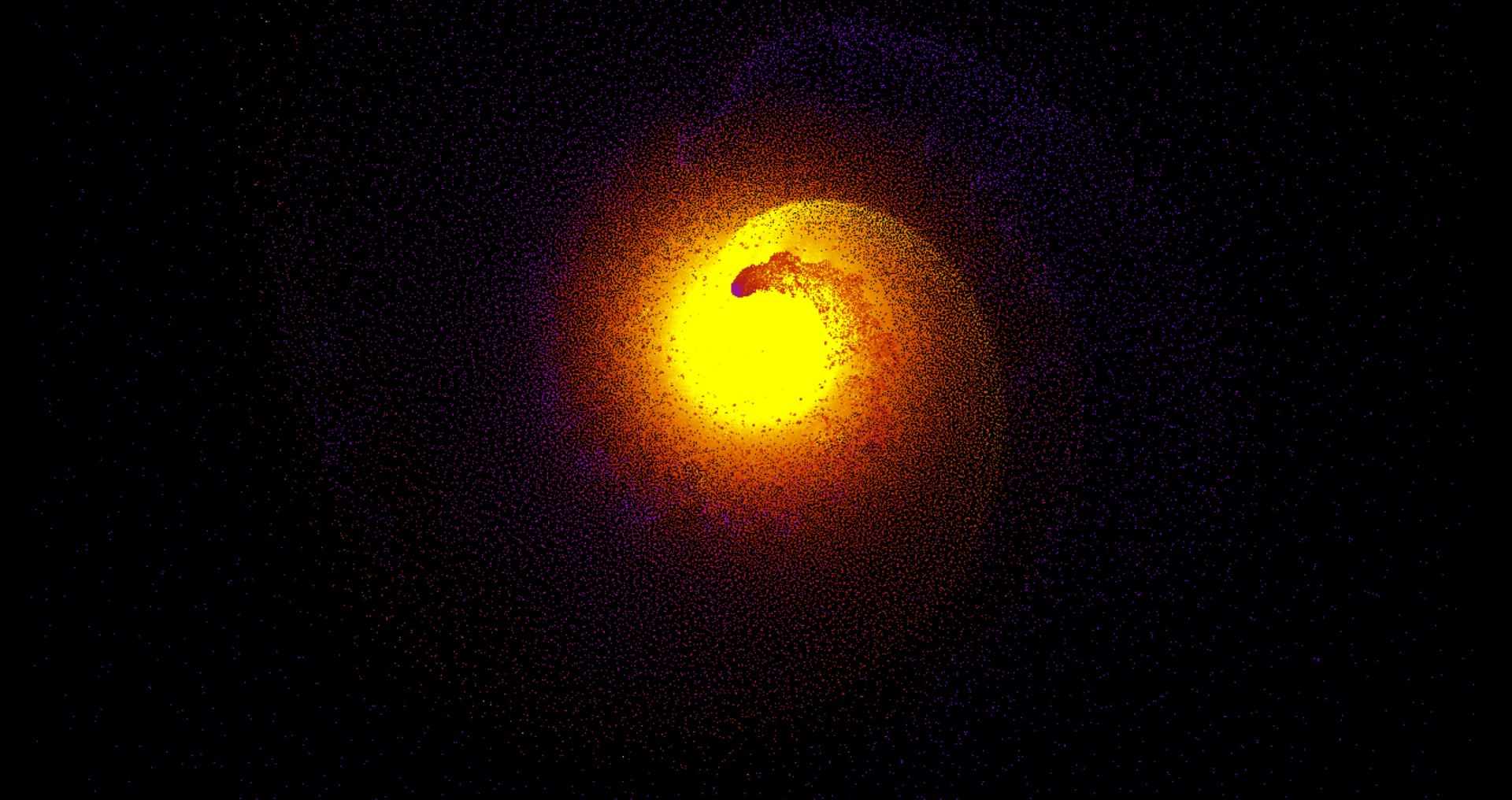
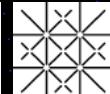


# Introduction to Parallel Computing



Dr. Rubén M. Cabezón  
Dr. Aurélien Cavelan



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

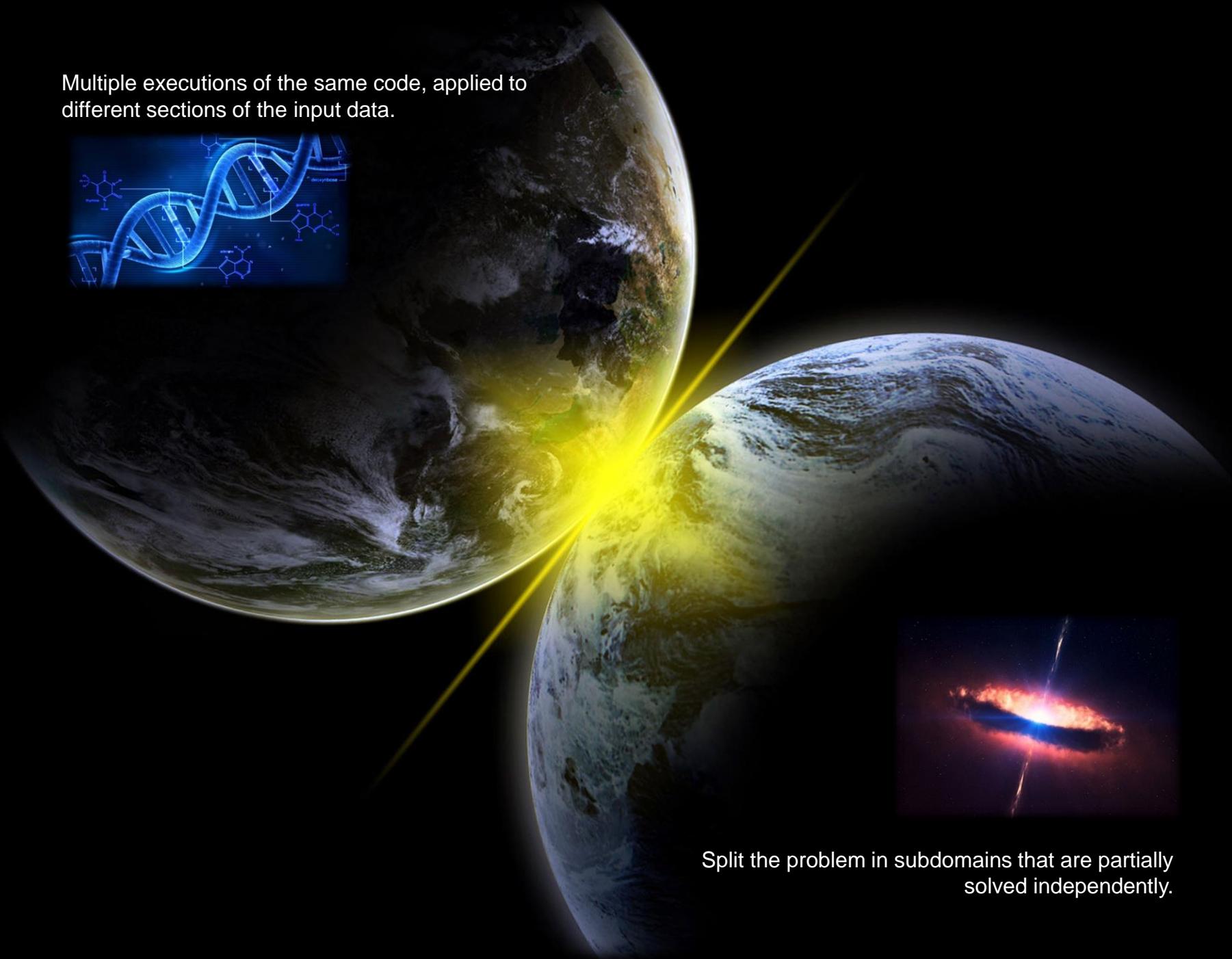
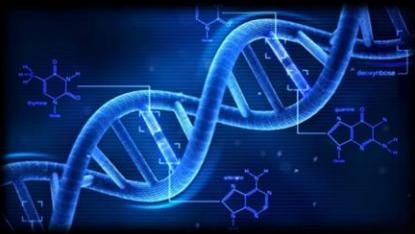
13:30 – 15:00: Lecture

15:00 – 15:15: Coffee break

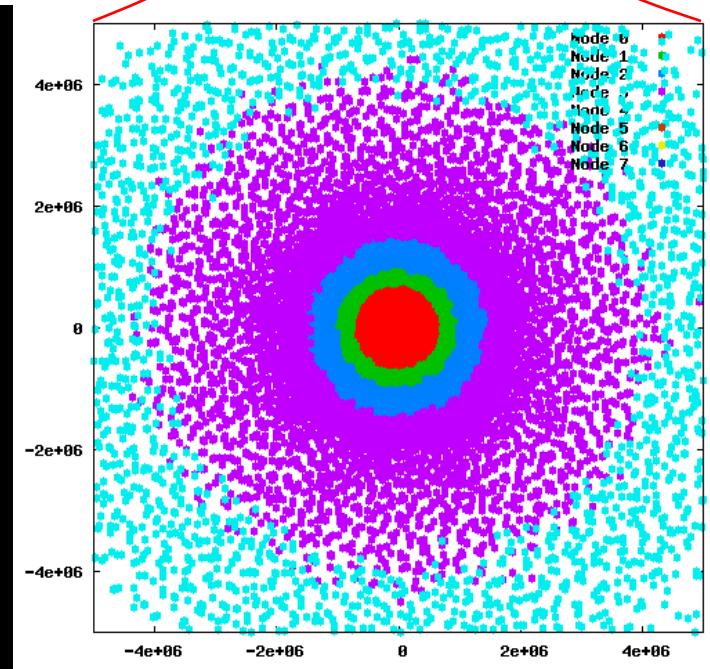
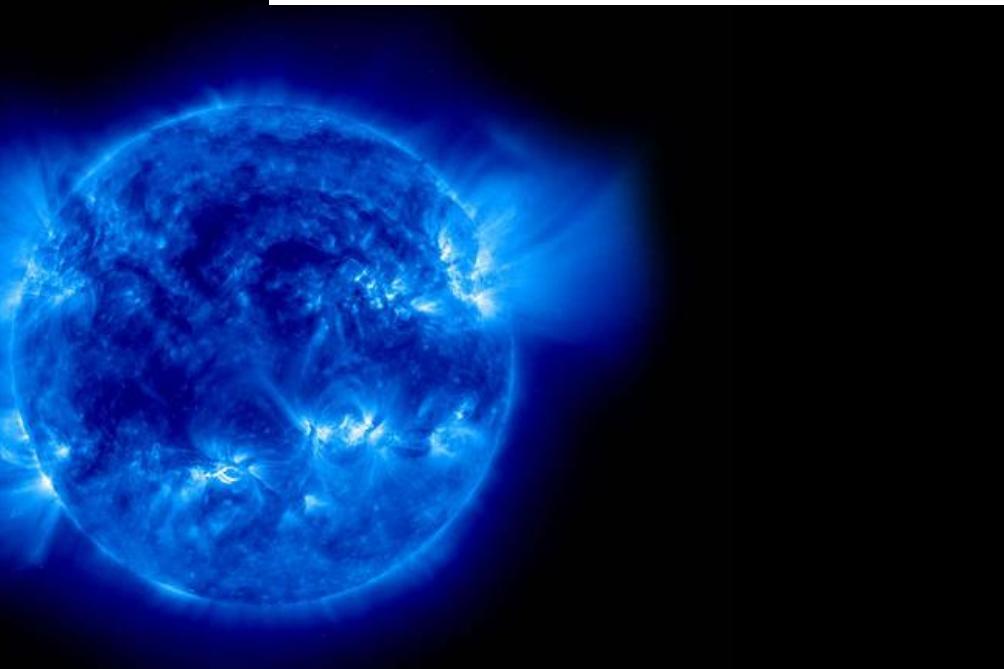
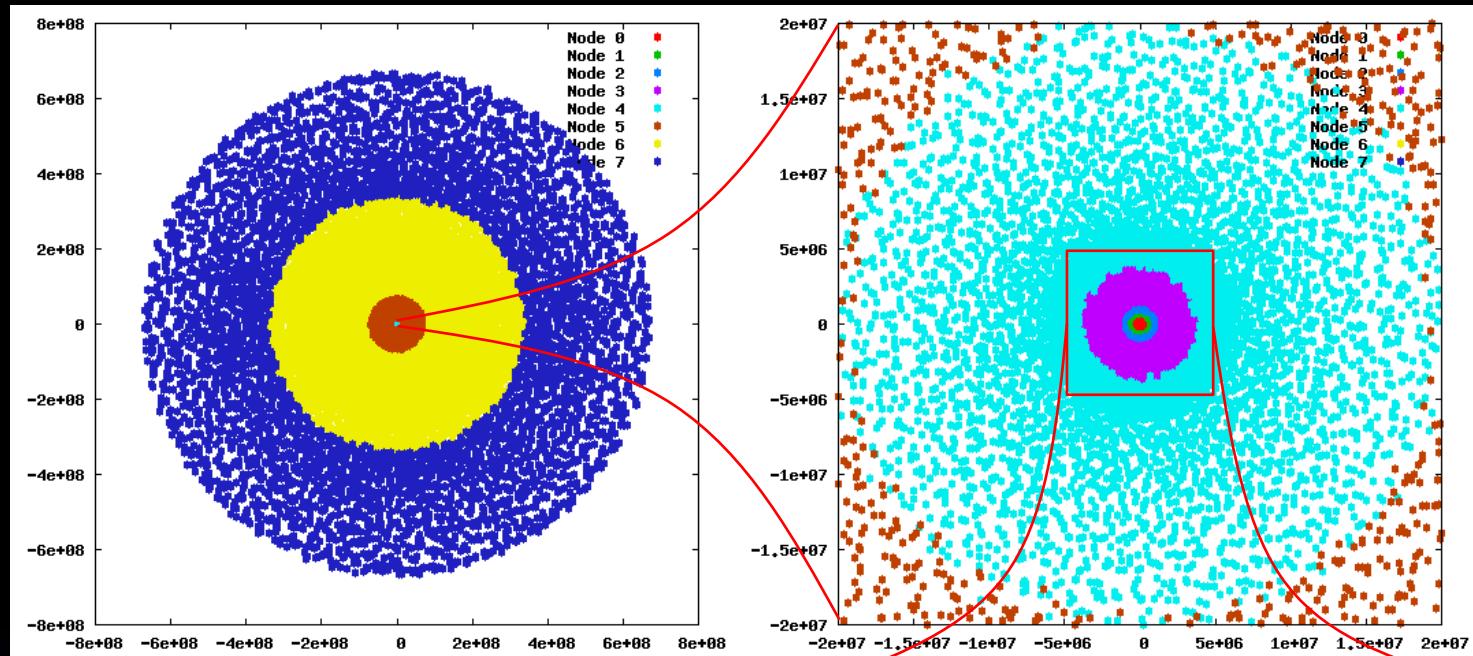
15:15 – 17:00: Lecture

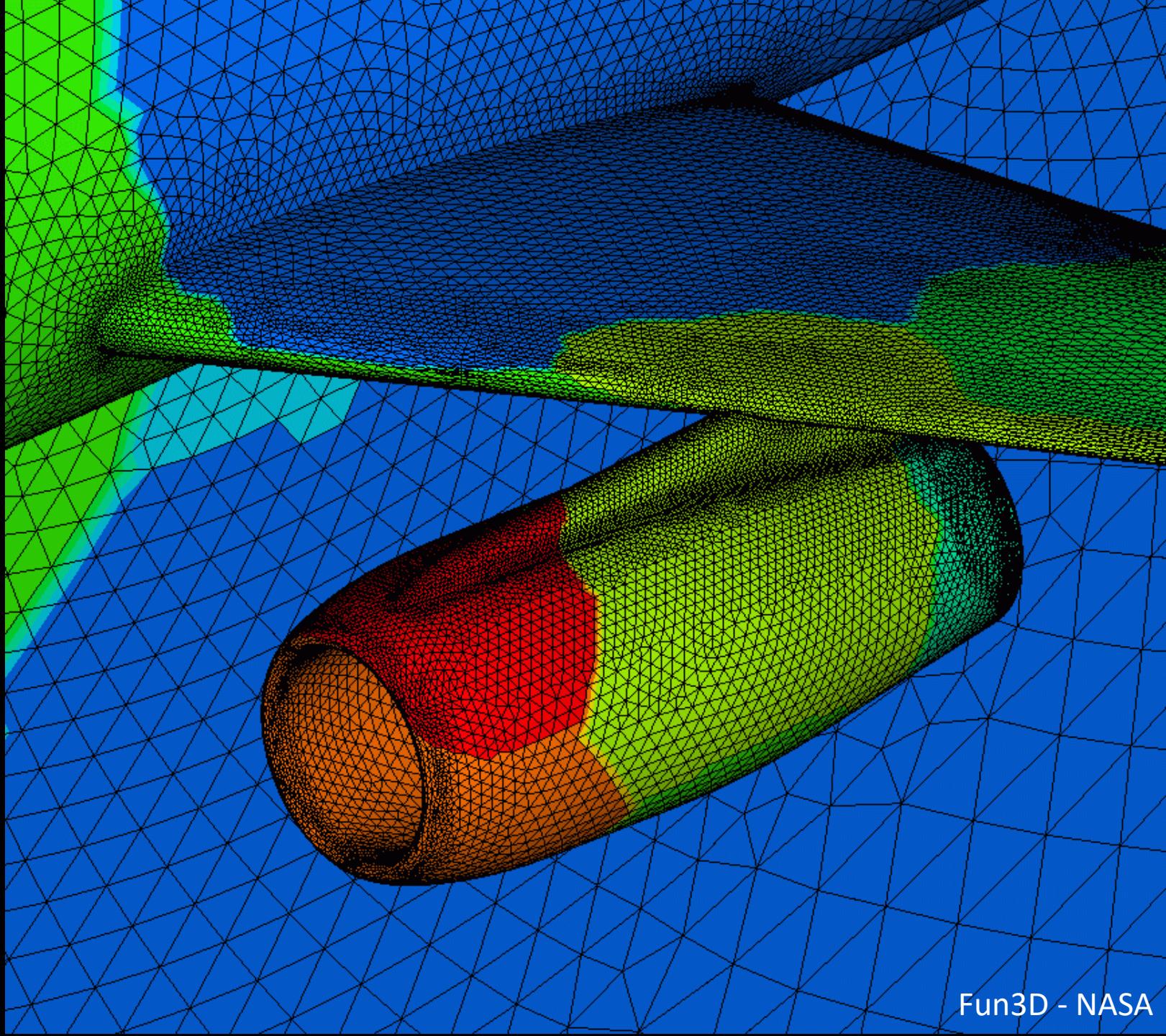


Multiple executions of the same code, applied to different sections of the input data.



Split the problem in subdomains that are partially solved independently.

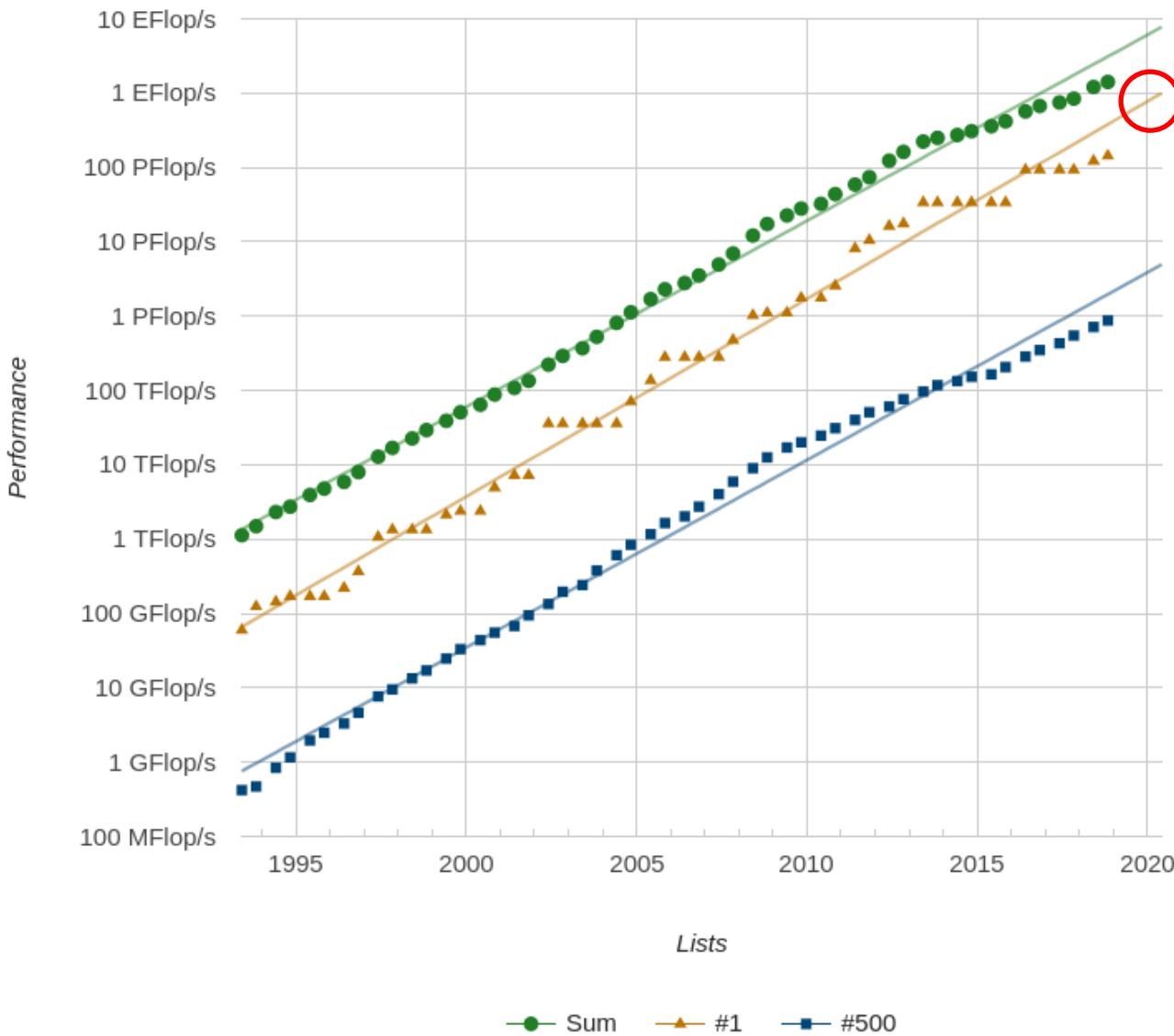




Fun3D - NASA

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	<b>Summit</b> - 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	<b>Sierra</b> - 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	<b>Sunway TaihuLight</b> - 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	<b>Tianhe-2A</b> - 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	<b>Frontera</b> - 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	<b>Piz Daint</b> - 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	<b>Trinity</b> - 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 Technology, Korea Korea	<b>AI Bridging Cloud Infrastructure</b>	391,680	19,880.0	32,576.6	1,649

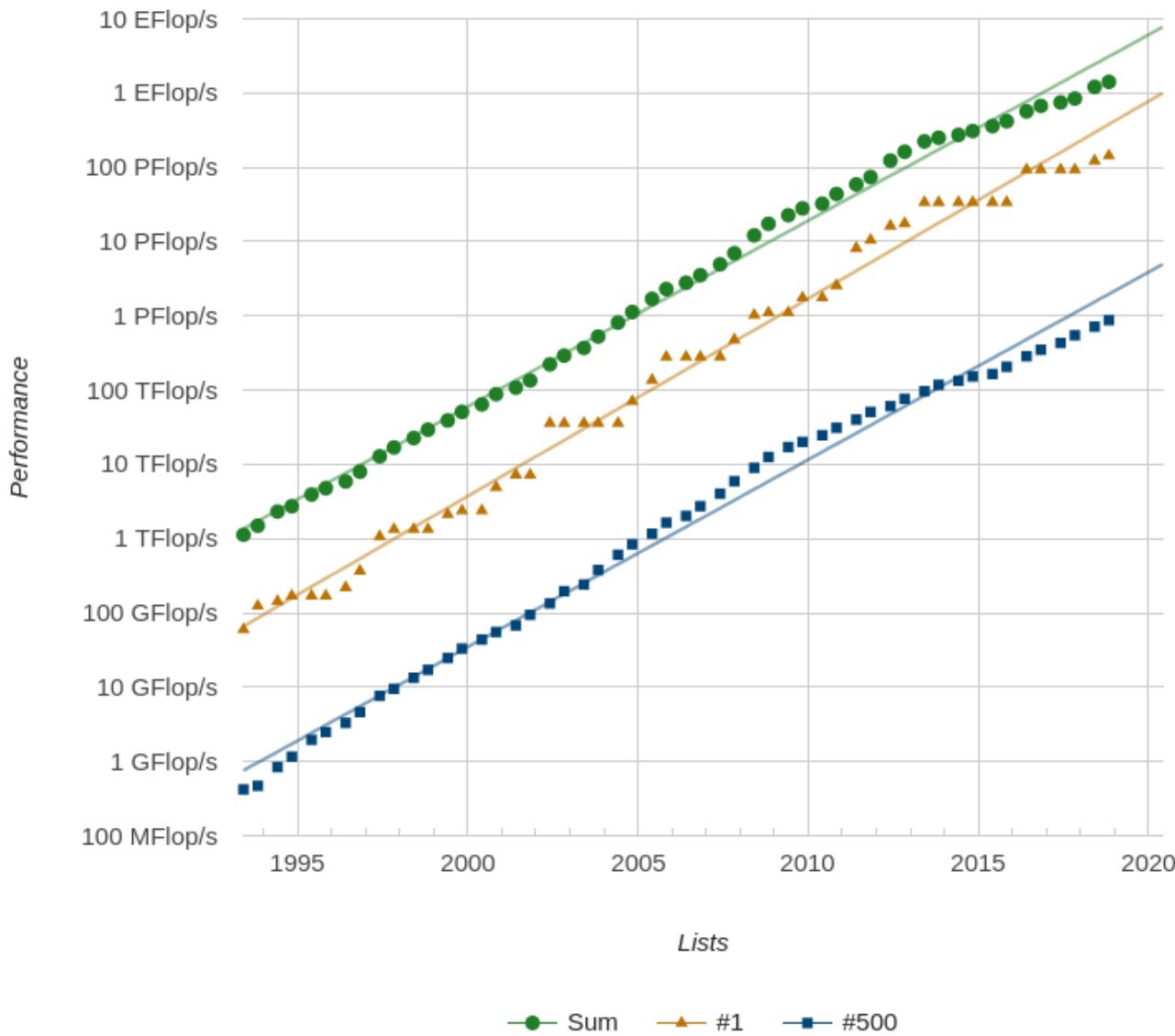
## Projected Performance Development



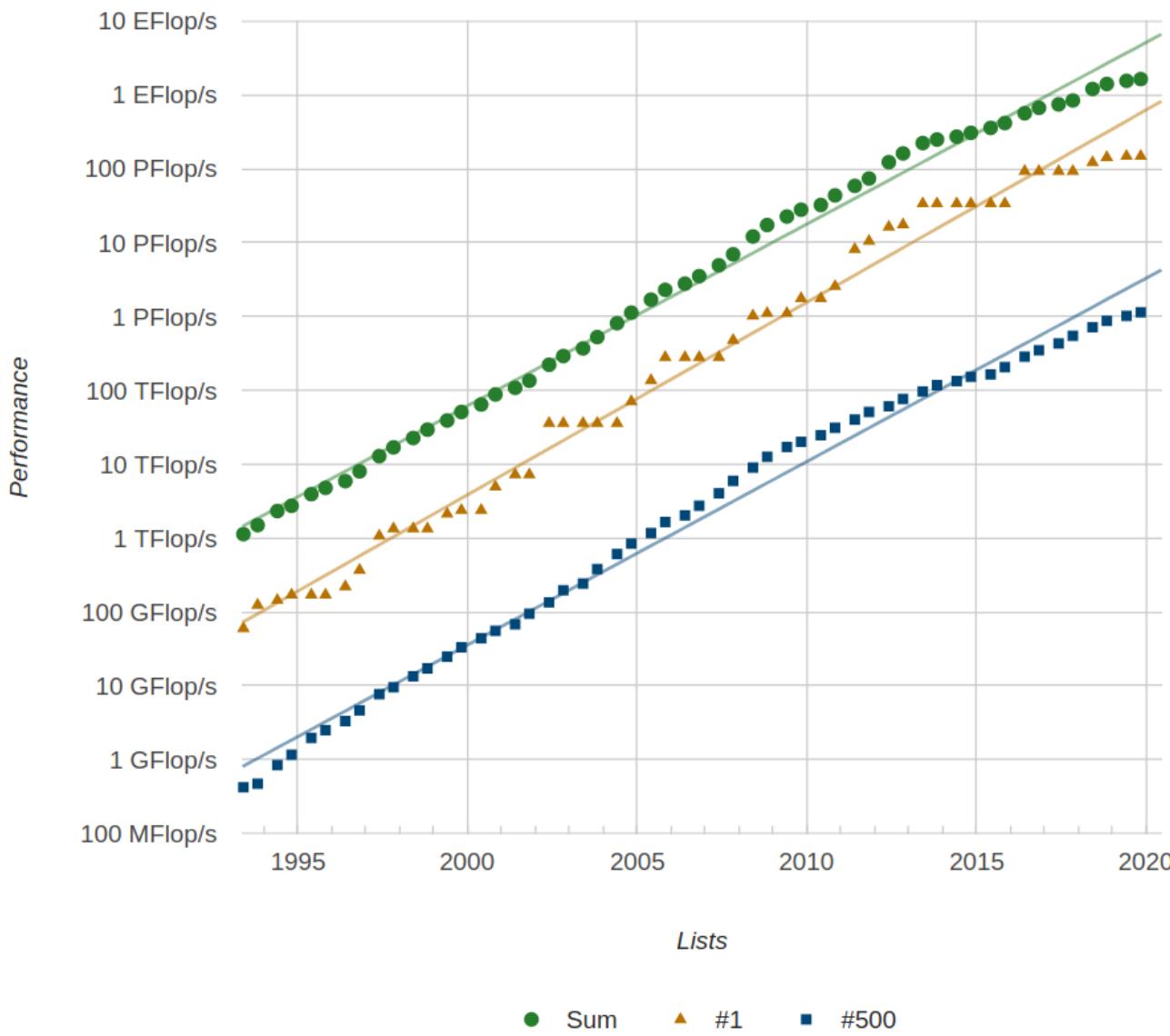
Exascale will be reached close to 2020

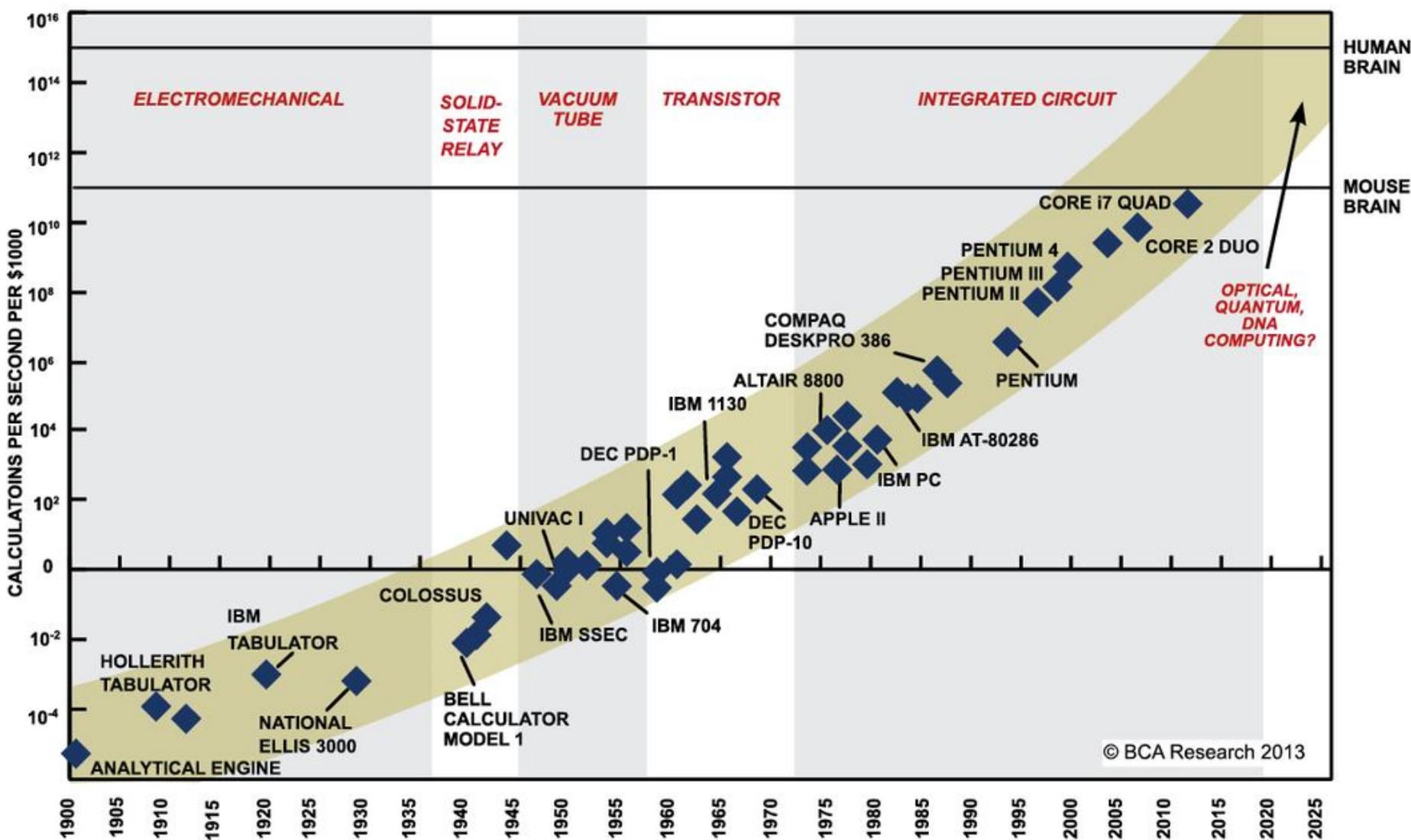
- Aerospace, Airframes, Jet Turbines
- Astrophysics
- Biological and medical systems
- Climate and weather
- Combustion
- Materials science
- Fusion energy
- National security
- Nuclear engineering

## Projected Performance Development

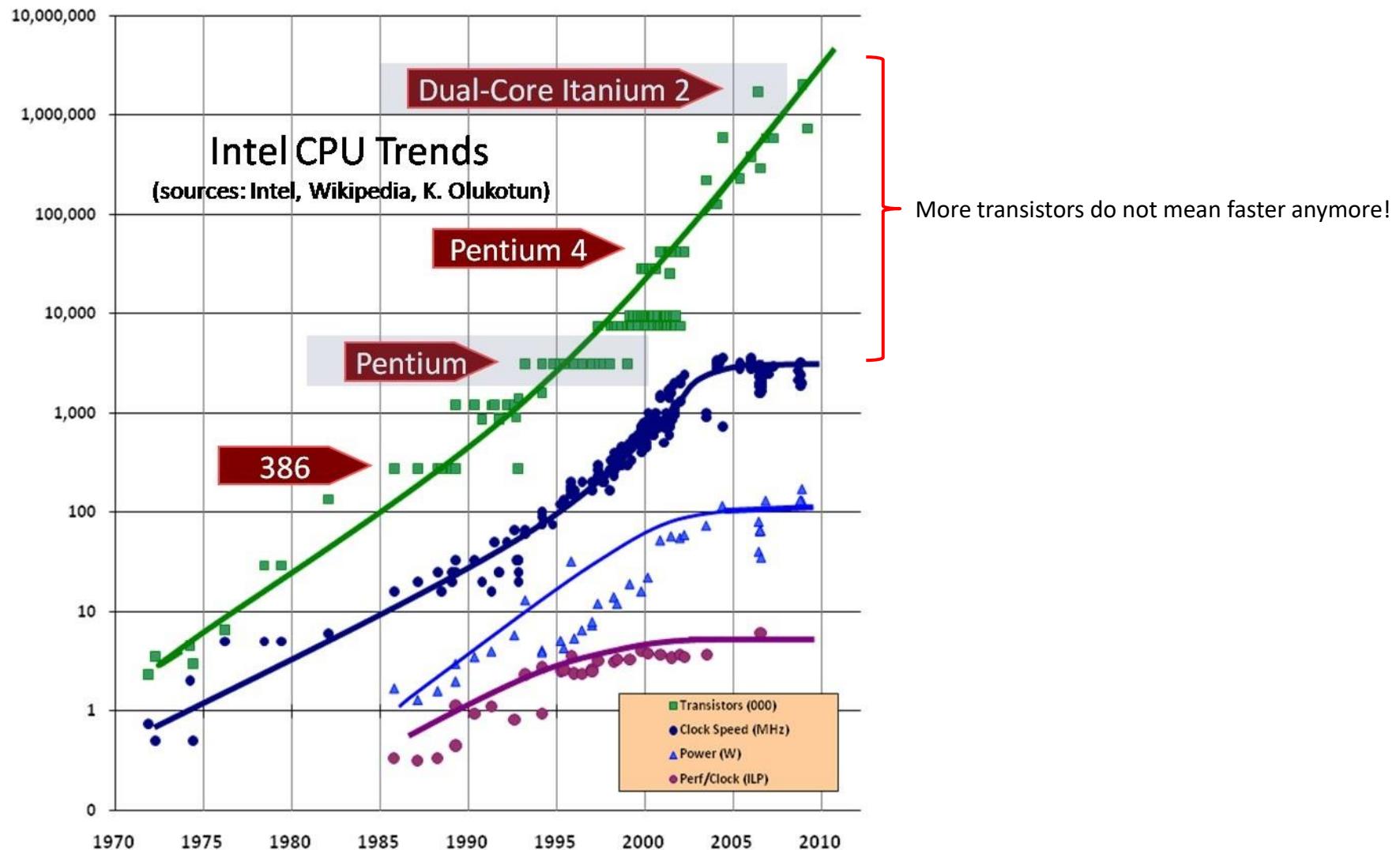


## Projected Performance Development

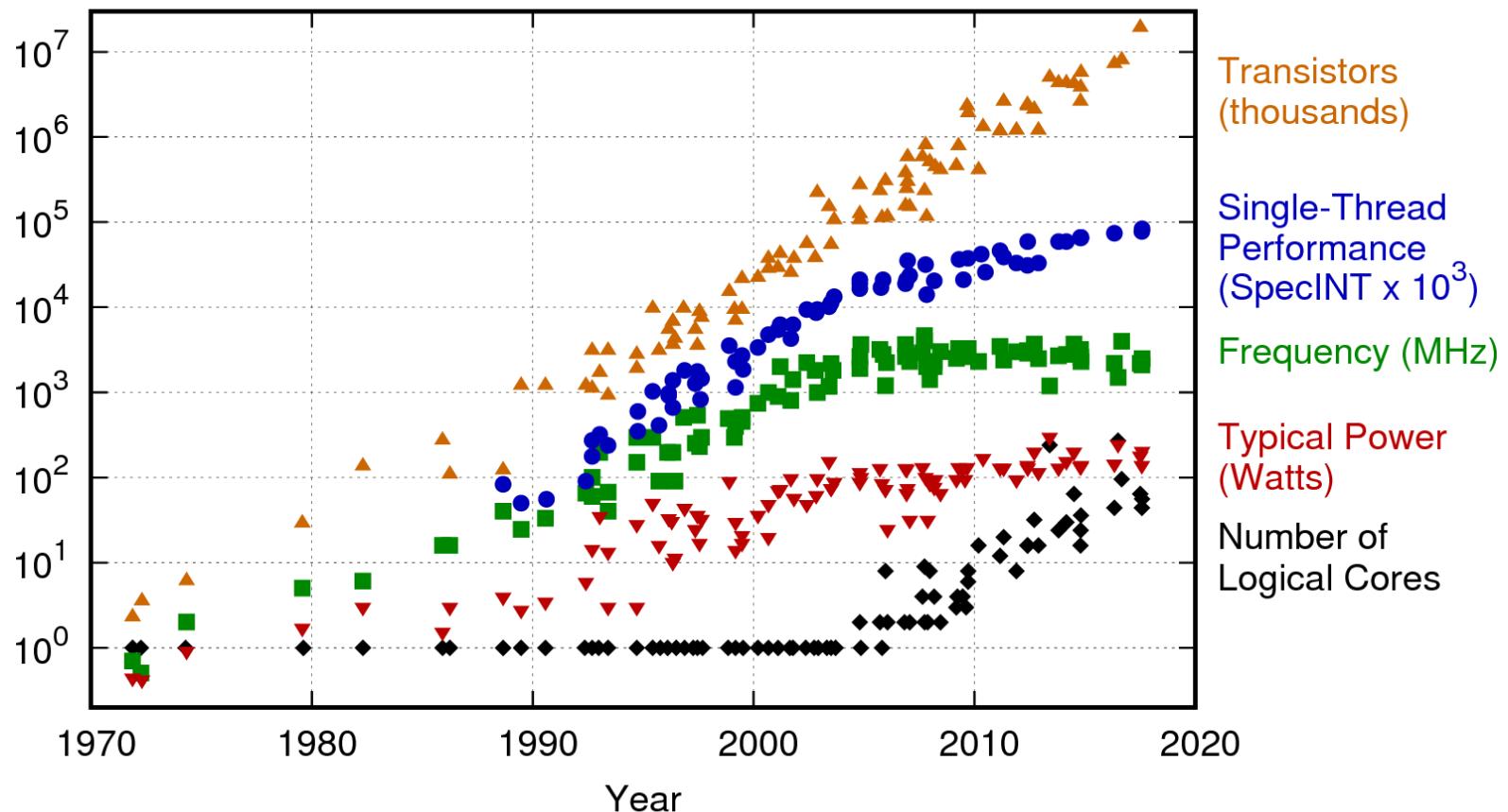




SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, THE VIKING PRESS, 2006. DATAPoints BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

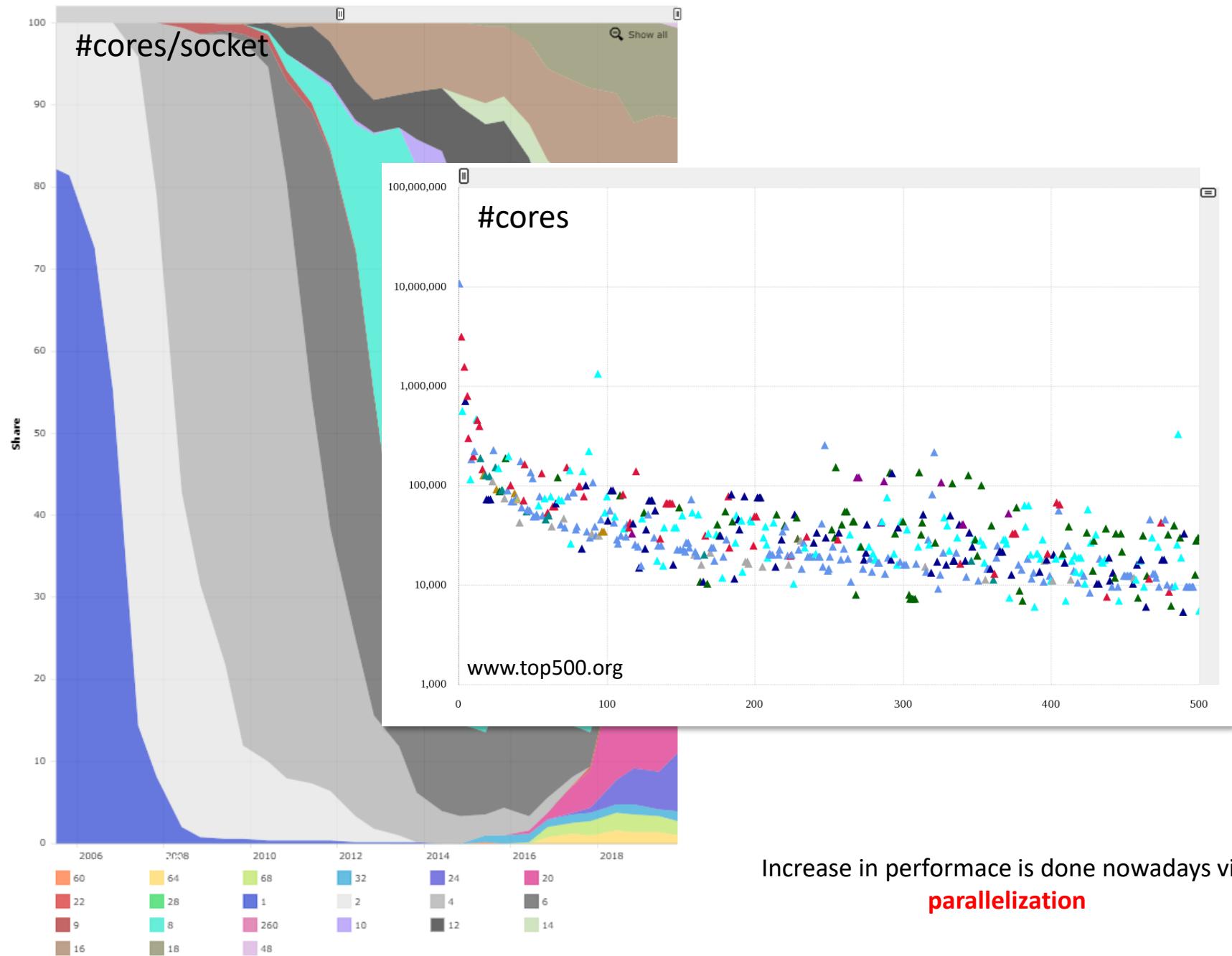


## 42 Years of Microprocessor Trend Data



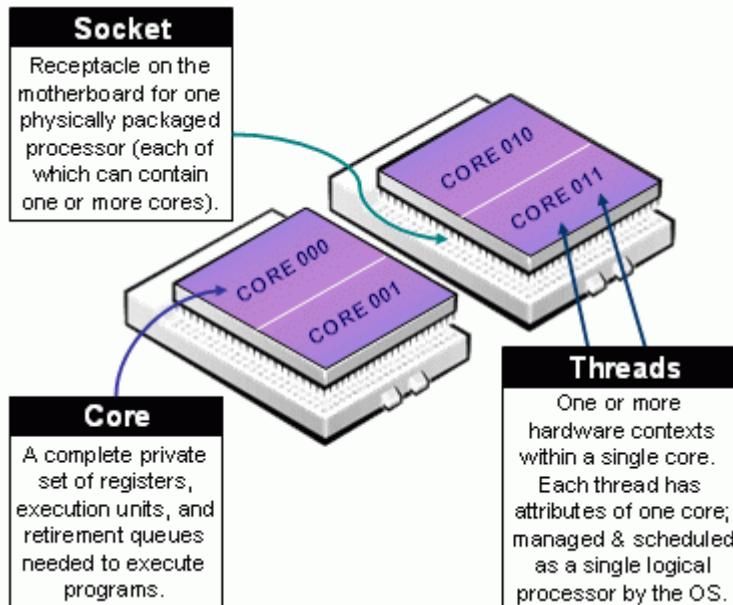
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2017 by K. Rupp

Increase in performance 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

node 0	process 0 thr0 thr1 thr2 thr3 thr4 thr5 thr6 thr7	process 1 thr0 thr1 thr2 thr3 thr4 thr5 thr6 thr7
node 1	process 2 thr0 thr1 thr2 thr3 thr4 thr5 thr6 thr7	process 3 thr0 thr1 thr2 thr3 thr4 thr5 thr6 thr7
node 2	process 4 thr0 thr1 thr2 thr3 thr4 thr5 thr6 thr7	process 5 thr0 thr1 thr2 thr3 thr4 thr5 thr6 thr7
node 3	process 6 thr0 thr1 thr2 thr3 thr4 thr5 thr6 thr7	process 7 thr0 thr1 thr2 thr3 thr4 thr5 thr6 thr7

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

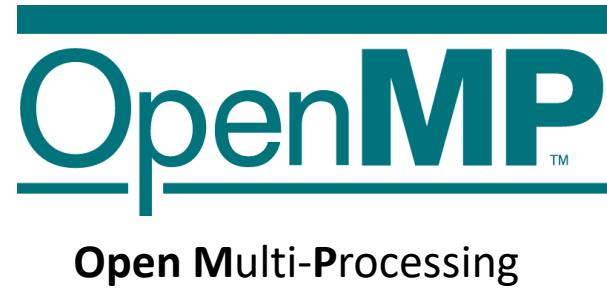


No

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



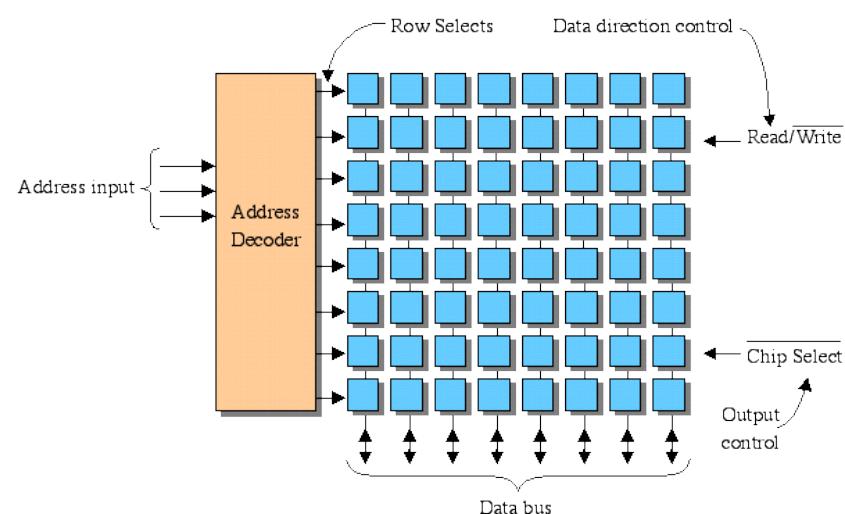
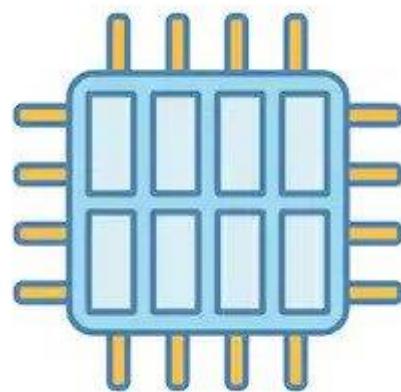
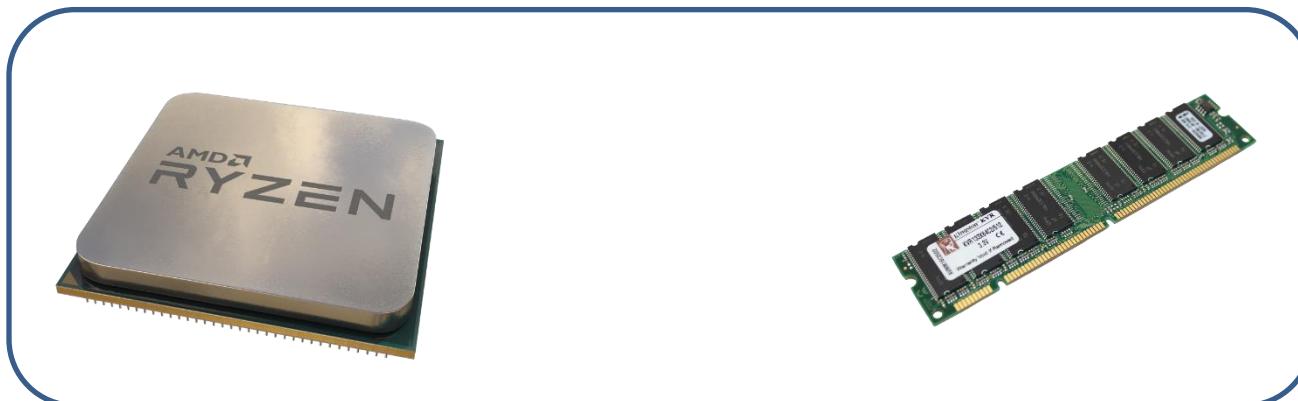
# PART I



# OpenMP™

Open Multi-Processing

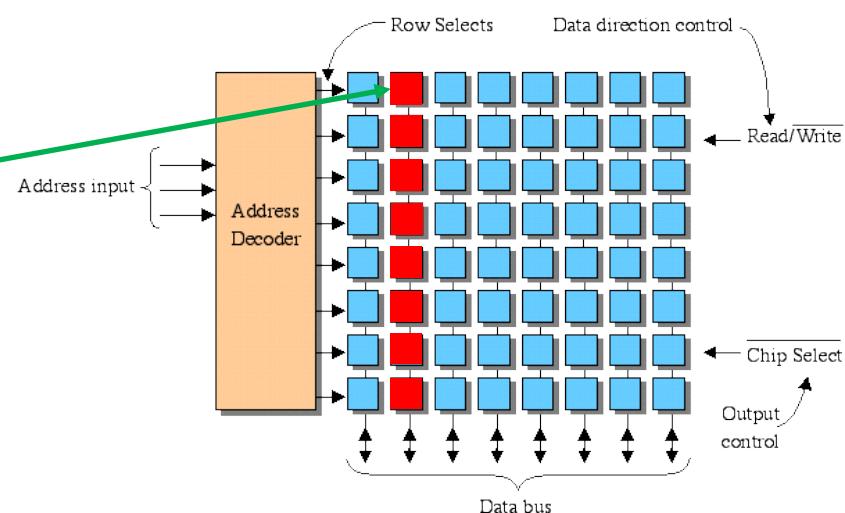
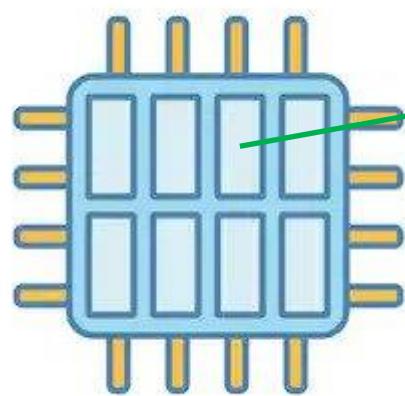
Works in **shared** memory systems  
Can use as many cores as there are in the **compute node**  
Uses **threads** to split the work  
Threads have both **private and shared** variables  
Uses **preprocessor directives** in commented lines



# OpenMP™

Open Multi-Processing

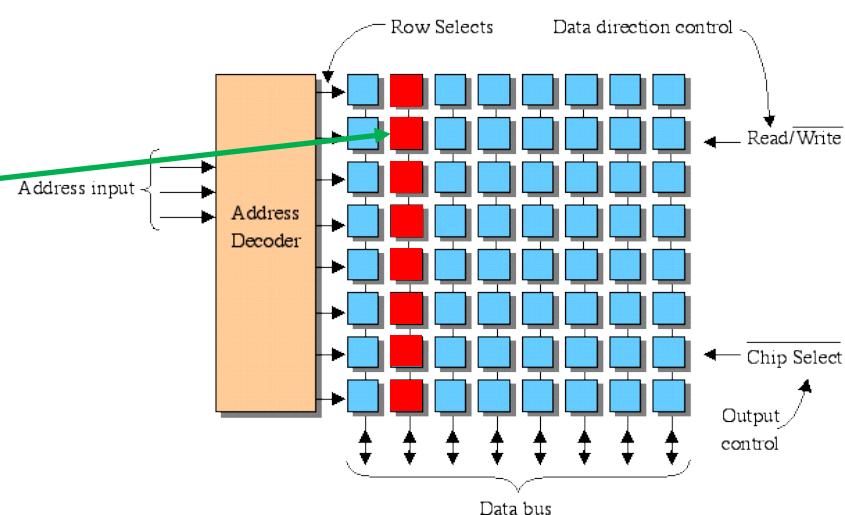
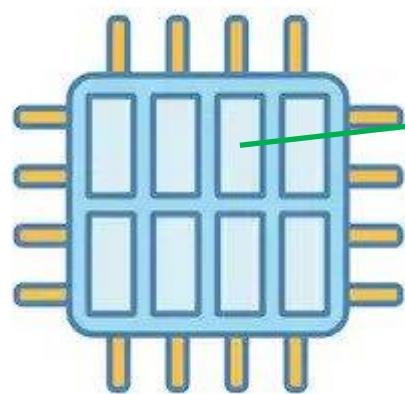
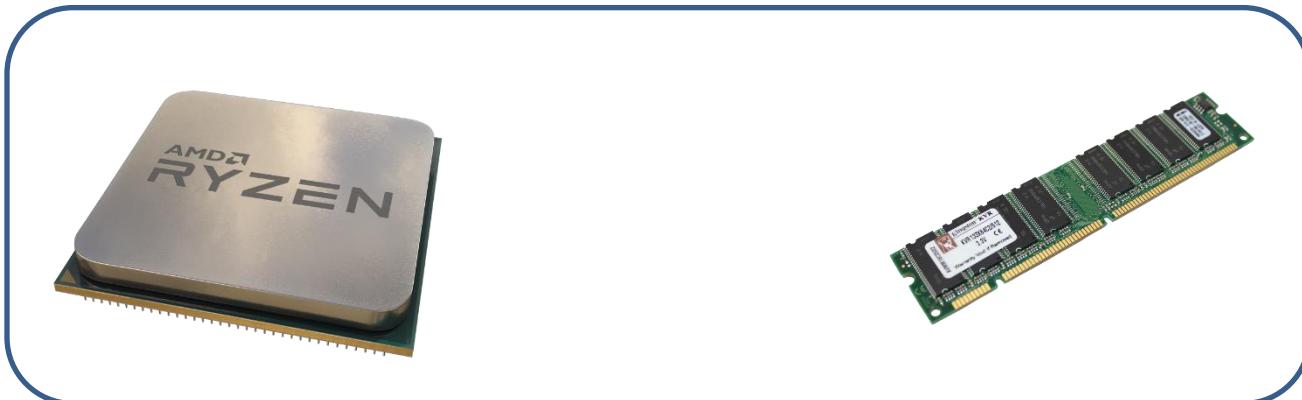
Works in **shared** memory systems  
Can use as many cores as there are in the **compute node**  
Uses **threads** to split the work  
Threads have both **private** and **shared** variables  
Uses **preprocessor directives** in commented lines



# OpenMP™

Open Multi-Processing

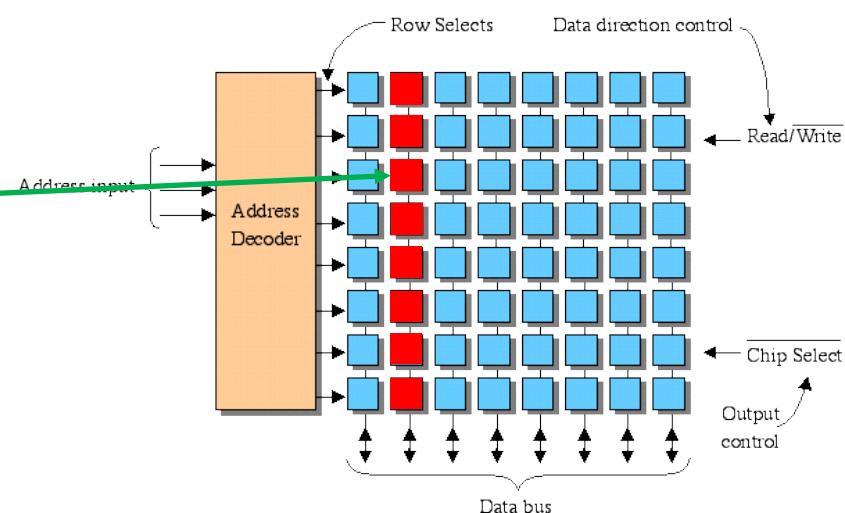
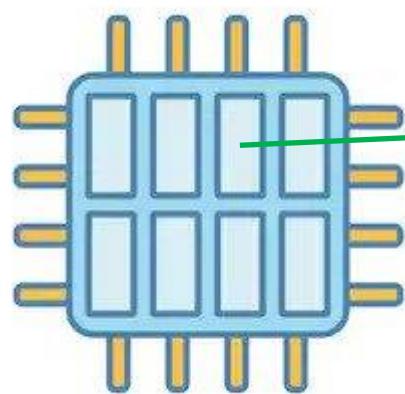
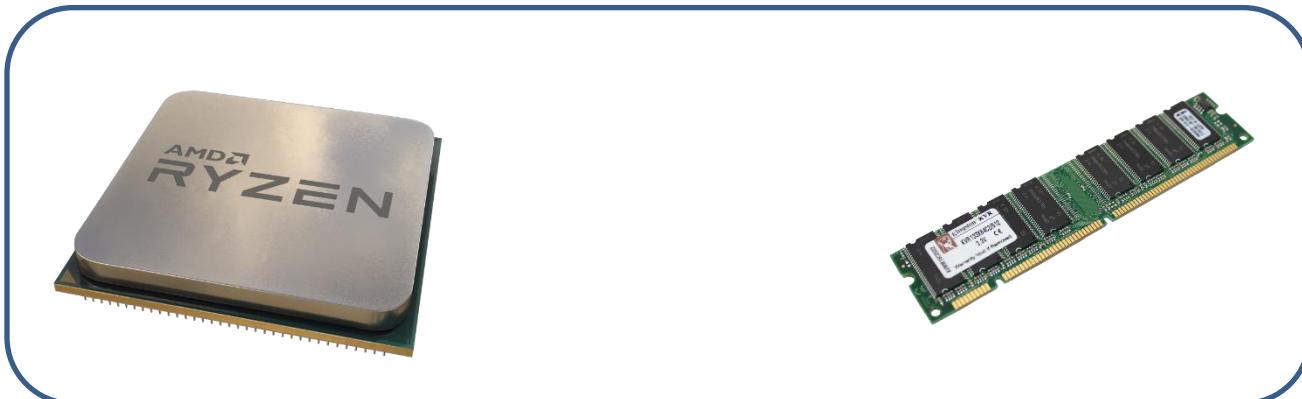
Works in **shared** memory systems  
Can use as many cores as there are in the **compute node**  
Uses **threads** to split the work  
Threads have both **private** and **shared** variables  
Uses **preprocessor directives** in commented lines



# OpenMP™

Open Multi-Processing

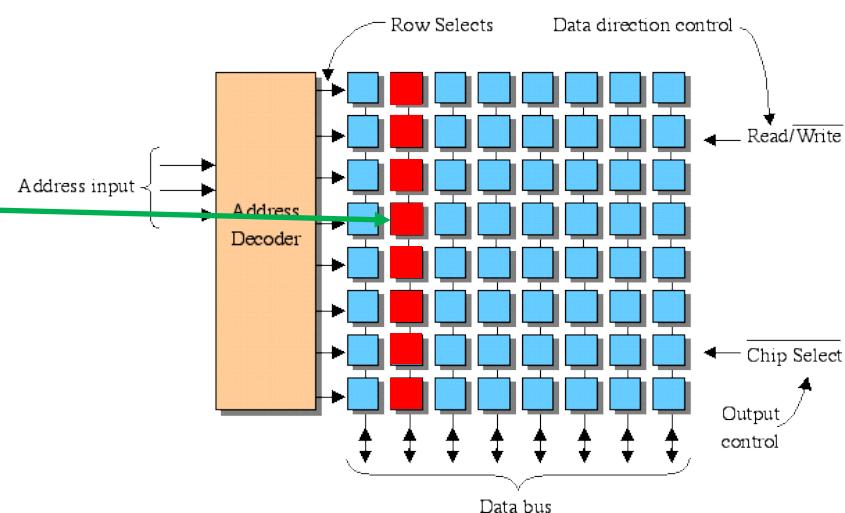
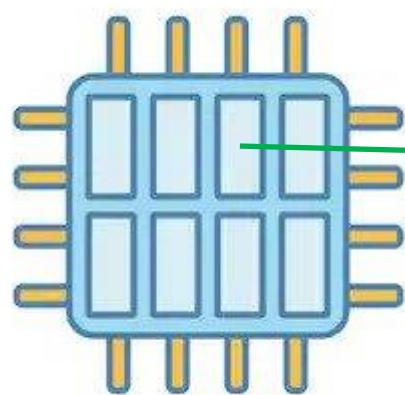
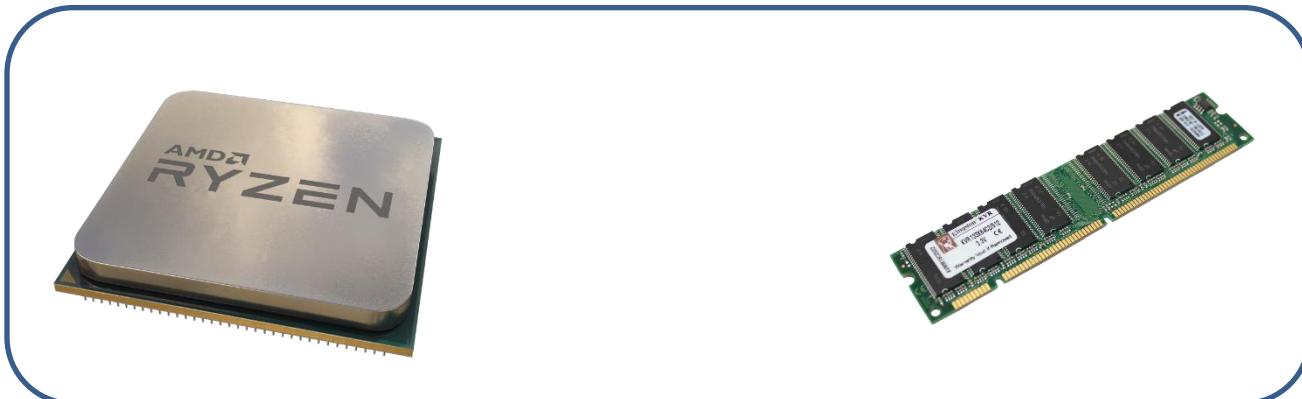
Works in **shared** memory systems  
Can use as many cores as there are in the **compute node**  
Uses **threads** to split the work  
Threads have both **private** and **shared** variables  
Uses **preprocessor directives** in commented lines



# OpenMP™

Open Multi-Processing

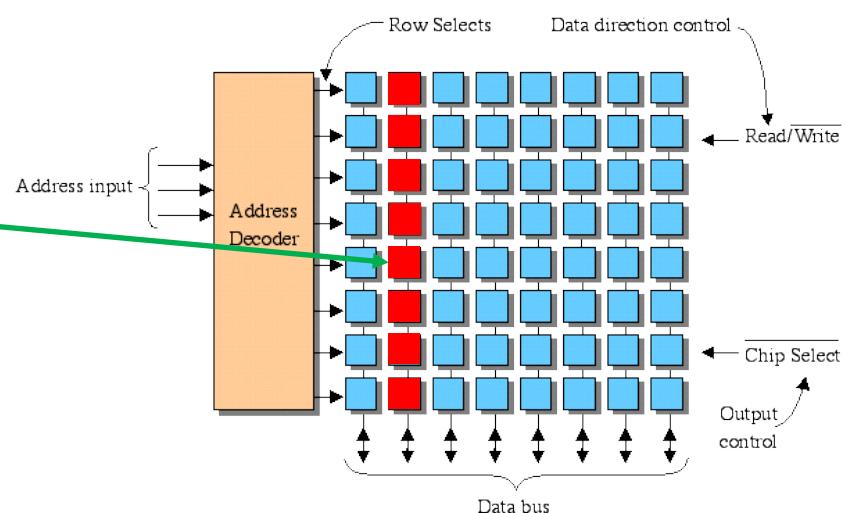
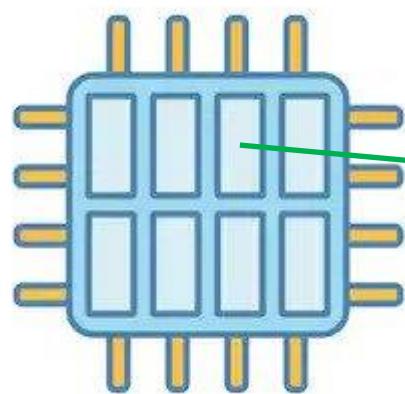
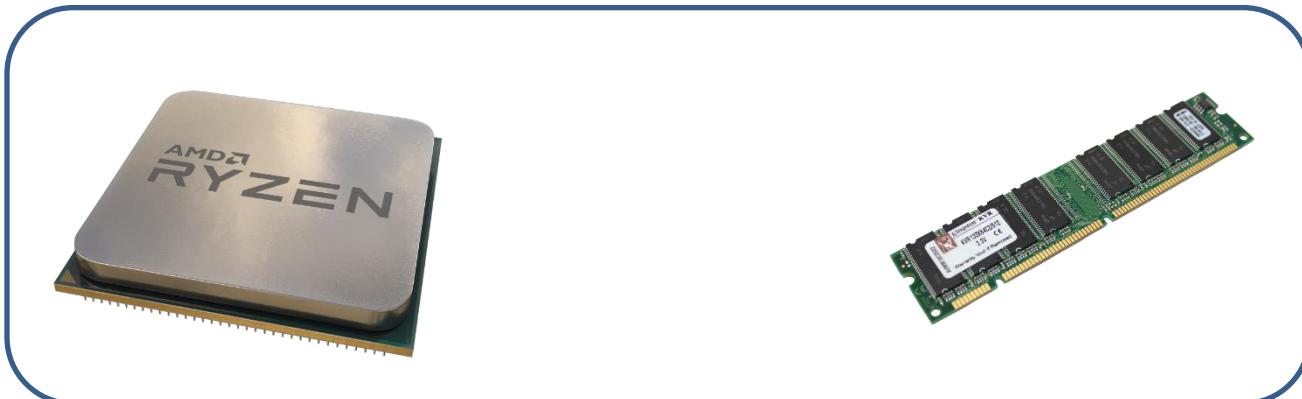
Works in **shared** memory systems  
Can use as many cores as there are in the **compute node**  
Uses **threads** to split the work  
Threads have both **private** and **shared** variables  
Uses **preprocessor directives** in commented lines



# OpenMP™

Open Multi-Processing

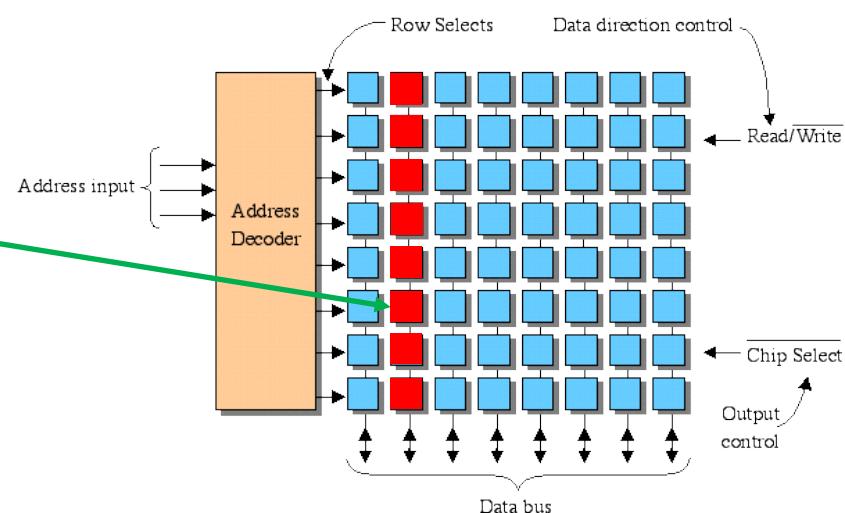
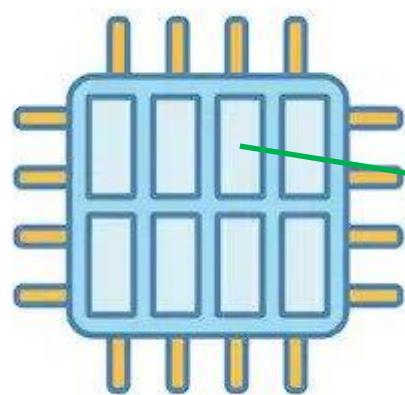
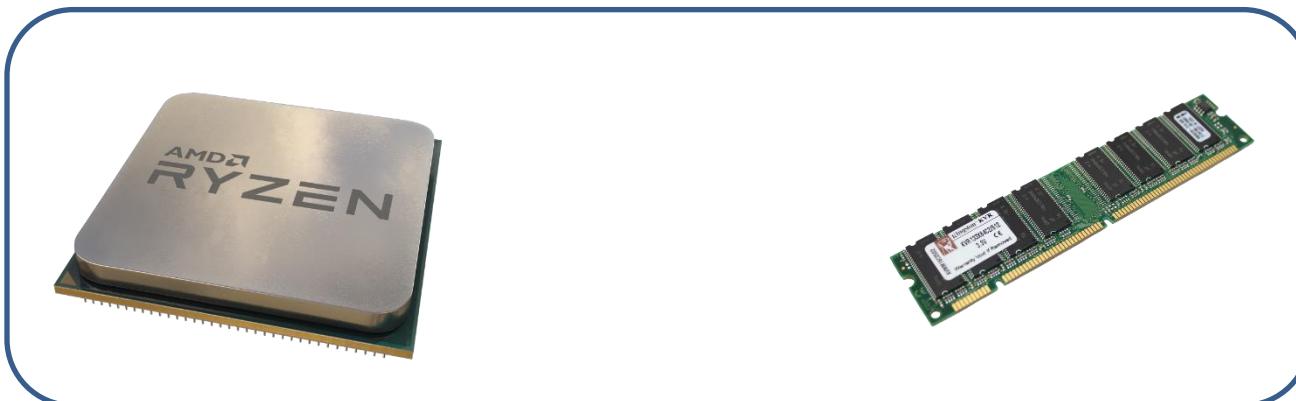
Works in **shared** memory systems  
Can use as many cores as there are in the **compute node**  
Uses **threads** to split the work  
Threads have both **private and shared** variables  
Uses **preprocessor directives** in commented lines



# OpenMP™

Open Multi-Processing

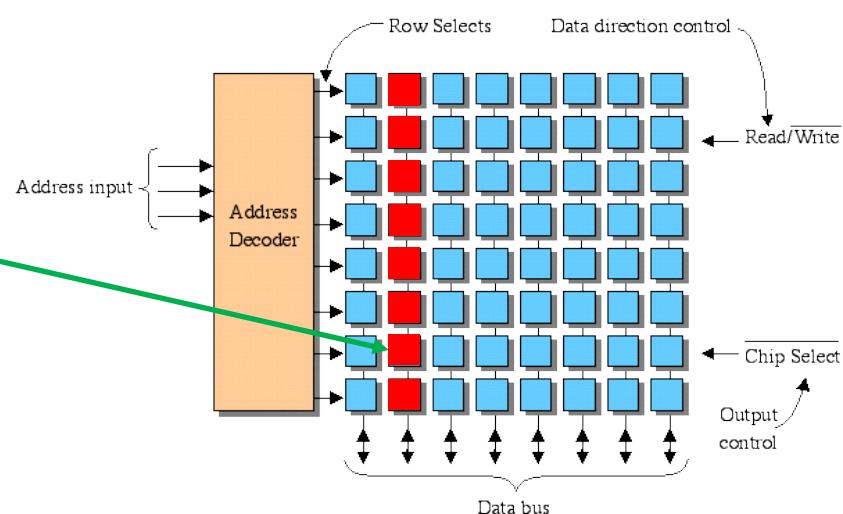
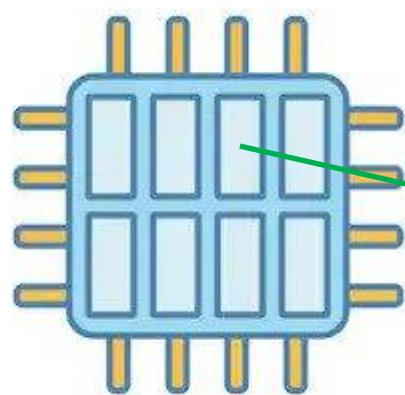
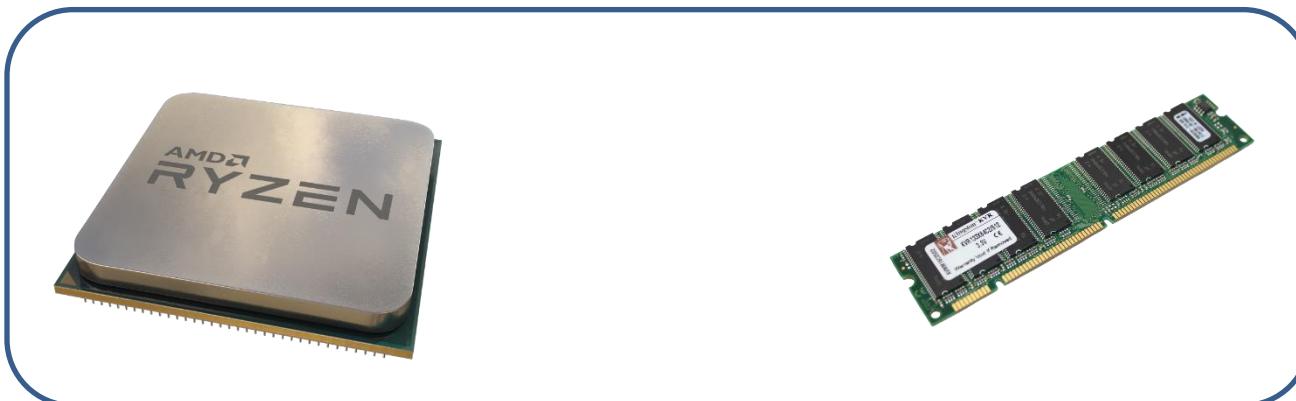
Works in **shared** memory systems  
Can use as many cores as there are in the **compute node**  
Uses **threads** to split the work  
Threads have both **private** and **shared** variables  
Uses **preprocessor directives** in commented lines



# OpenMP™

Open Multi-Processing

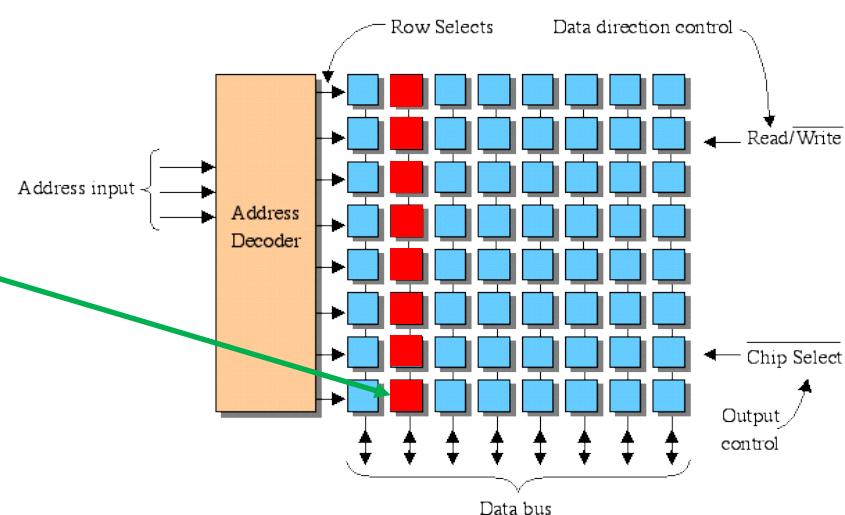
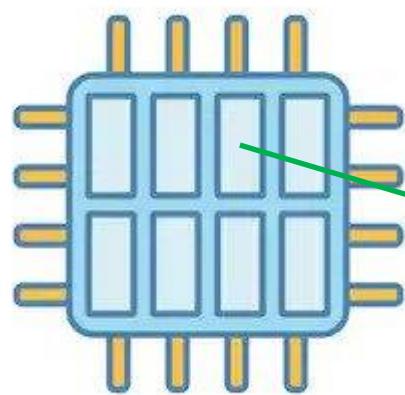
Works in **shared** memory systems  
Can use as many cores as there are in the **compute node**  
Uses **threads** to split the work  
Threads have both **private and shared** variables  
Uses **preprocessor directives** in commented lines



# OpenMP™

Open Multi-Processing

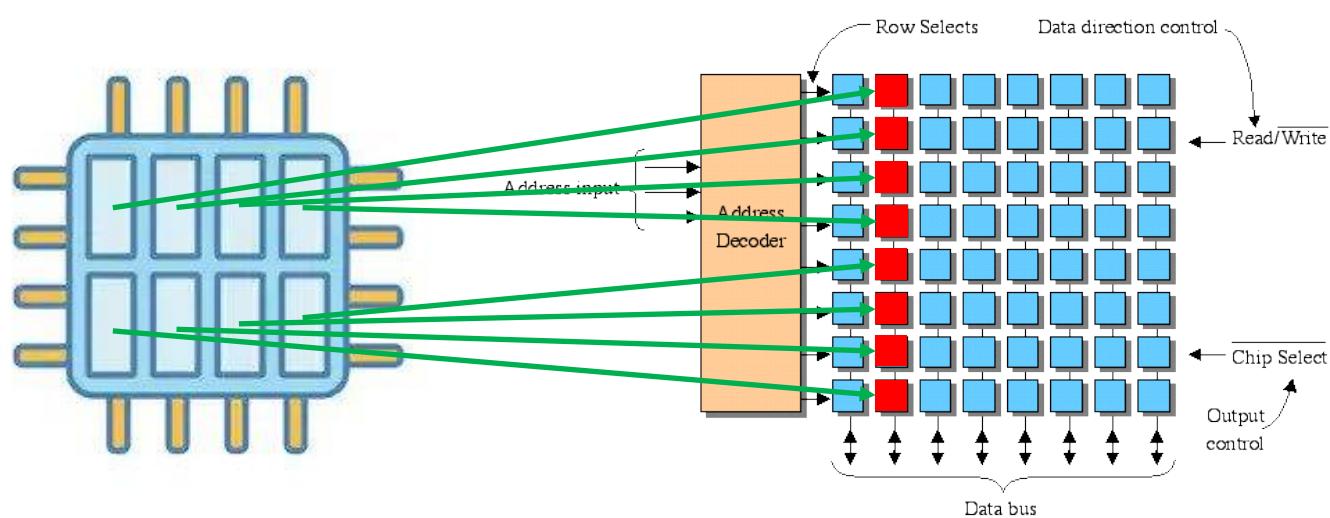
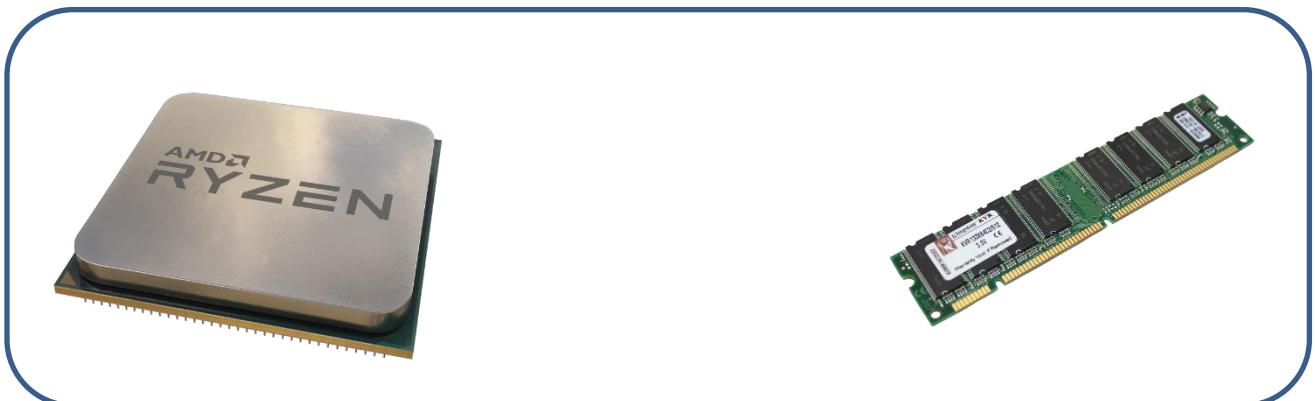
Works in **shared** memory systems  
Can use as many cores as there are in the **compute node**  
Uses **threads** to split the work  
Threads have both **private** and **shared** variables  
Uses **preprocessor directives** in commented lines



# OpenMP™

Open Multi-Processing

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.community.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)

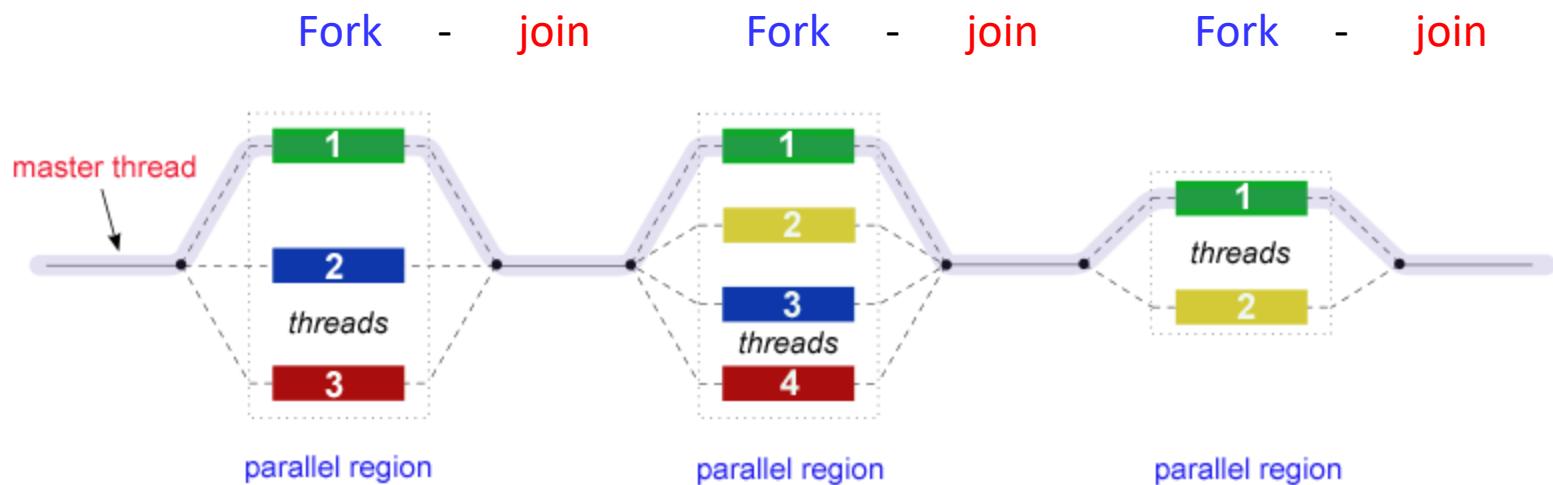
- In-person Courses:

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)



## 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 routine.

- `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 1

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
```

```
!$omp parallel  
 !$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)

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
```

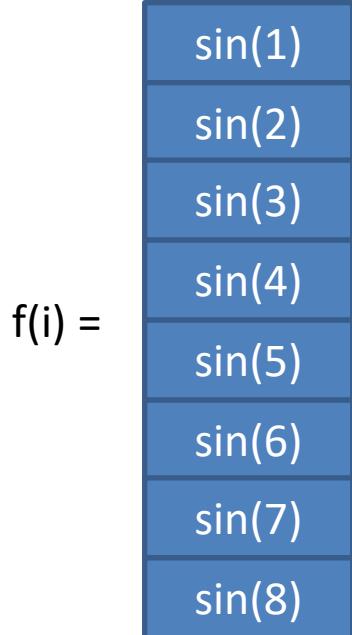
f(i) =	<table border="1"><tr><td>sin(1)</td></tr><tr><td>sin(2)</td></tr><tr><td>sin(3)</td></tr><tr><td>sin(4)</td></tr><tr><td>sin(5)</td></tr><tr><td>sin(6)</td></tr><tr><td>sin(7)</td></tr><tr><td>sin(8)</td></tr></table>	sin(1)	sin(2)	sin(3)	sin(4)	sin(5)	sin(6)	sin(7)	sin(8)
sin(1)									
sin(2)									
sin(3)									
sin(4)									
sin(5)									
sin(6)									
sin(7)									
sin(8)									

Check env. variable, f.e.:  
\$OMP\_NUM\_THREADS=4

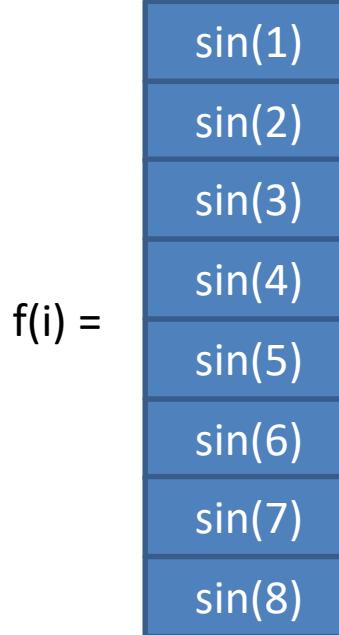
f(i) =	<table border="1"><tr><td>sin(1)</td></tr><tr><td>sin(2)</td></tr><tr><td>sin(3)</td></tr><tr><td>sin(4)</td></tr><tr><td>sin(5)</td></tr><tr><td>sin(6)</td></tr><tr><td>sin(7)</td></tr><tr><td>sin(8)</td></tr></table>	sin(1)	sin(2)	sin(3)	sin(4)	sin(5)	sin(6)	sin(7)	sin(8)
sin(1)									
sin(2)									
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
```



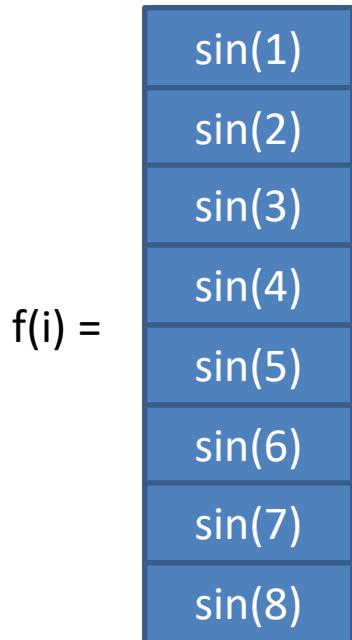
Check env. variable, f.e.:  
\$OMP\_NUM\_THREADS=4



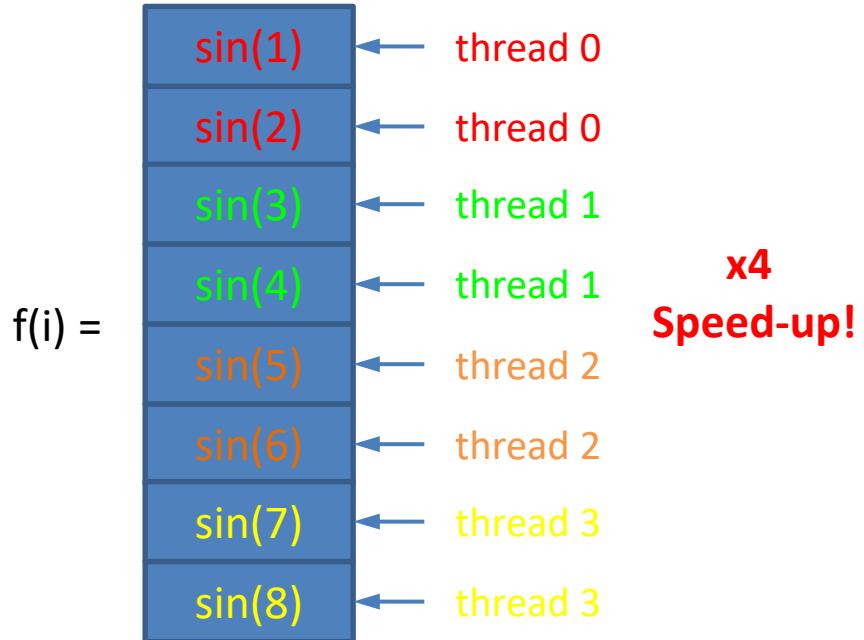
Creates 4 threads and distributes the work evenly

```
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
```



Check env. variable, f.e.:  
\$OMP\_NUM\_THREADS=4



Creates 4 threads and distributes the work evenly

```

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

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

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

```

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

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`)

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
23339	cabezon	20	0	89876	904	696	R	769.6	0.0	0:11.19	integral.out
31848	cabezon	20	0	1135m	151m	11m	S	17.8	2.0	318:52.56	chrome
21008	cabezon	20	0	3985m	2.1g	2.1g	S	5.9	27.7	10:53.79	VirtualBox
19624	cabezon	20	0	956m	118m	8828	S	2.0	1.5	241:53.73	chrome
23309	cabezon	20	0	30816	1876	1276	R	2.0	0.0	0:00.32	top
1	root	20	0	36924	588	332	S	0.0	0.0	0:01.17	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.07	kthreadd
3	root	RT	0	0	0	0	S	0.0	0.0	0:01.64	migration/0
4	root	20	0	0	0	0	S	0.0	0.0	0:05.95	ksoftirqd/0
5	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	stopper/0
6	root	RT	0	0	0	0	S	0.0	0.0	0:01.97	watchdog/0
7	root	RT	0	0	0	0	S	0.0	0.0	0:00.97	migration/1
8	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	stopper/1
9	root	20	0	0	0	0	S	0.0	0.0	0:04.68	ksoftirqd/1
10	root	RT	0	0	0	0	S	0.0	0.0	0:01.78	watchdog/1
11	root	RT	0	0	0	0	S	0.0	0.0	0:00.62	migration/2
12	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	stopper/2

```

!$omp parallel
!$omp do schedule(static)
do i = 1, n
    f(i) = sin(dble(i))
end do
 !$omp end do
 !$omp end parallel

```

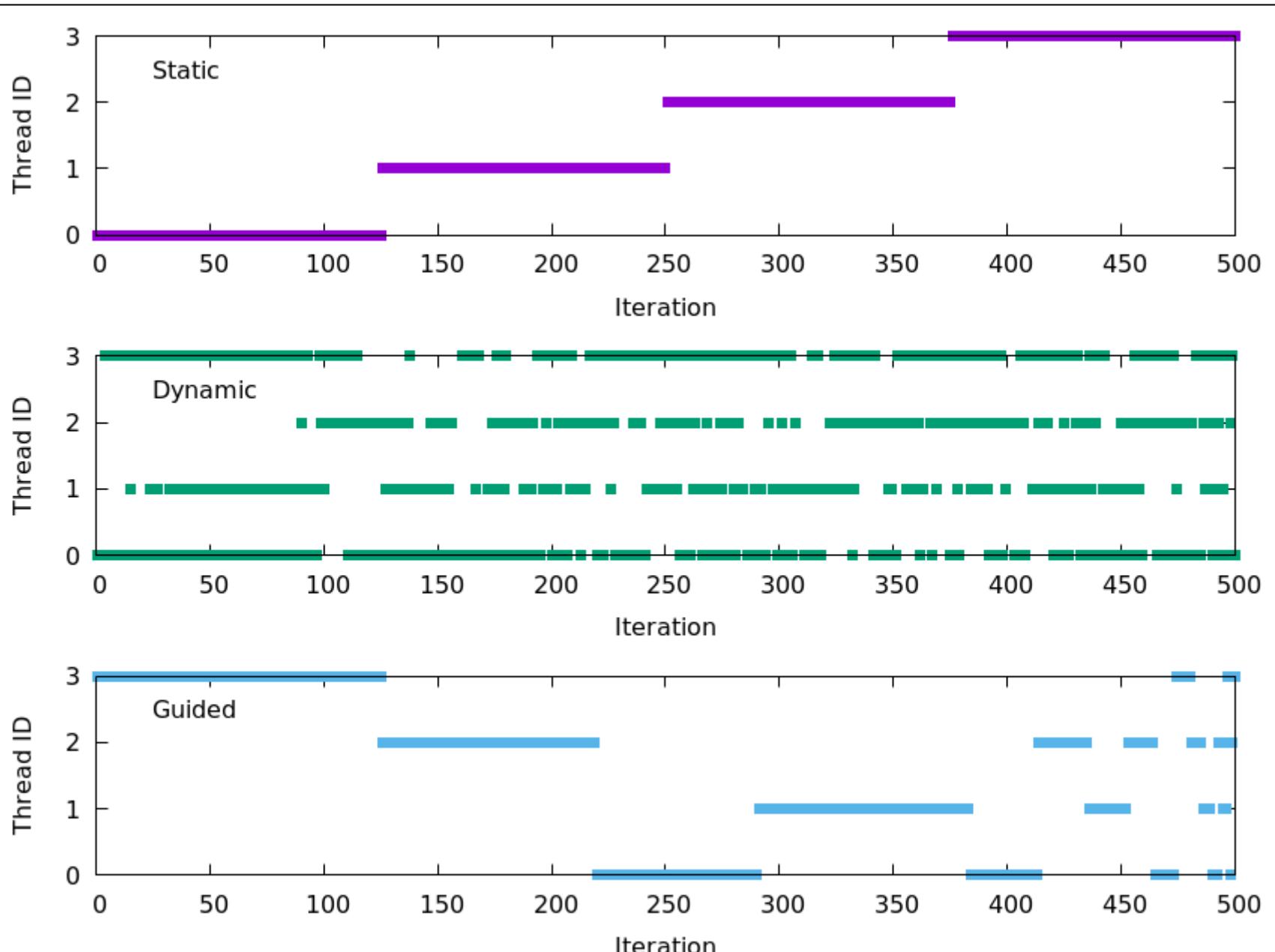
Parallel loop **evenly** divided among the threads

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

**static** schedule

Other options are available: **!\$omp do schedule(<kind> [,chunk\_size])**

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



```
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
```

factorial(i) =

1
2
6
24
120
720
5040

There is something wrong here!

We cannot ensure this!

factorial(i) =



1
2
6
24
120
720
5040

thread 0

thread 0

thread 1

thread 1

thread 2

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

factorial(i) =

1
2

Threads access data in a disordered way

← 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
```

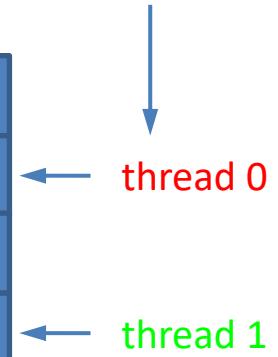
factorial(i) =

1
2
6
24
120
720
5040

factorial(i) =

1
2
0

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
```

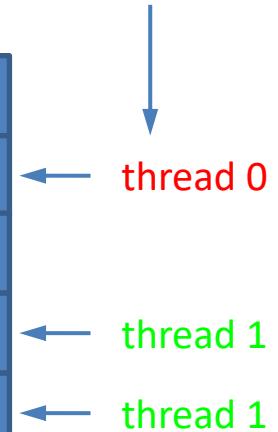
factorial(i) =

1
2
6
24
120
720
5040

factorial(i) =

1
2
0
0

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
```

factorial(i) =

1
2
6
24
120
720
5040

factorial(i) =

1
2
6
0
0

Threads access data in a disordered way



- ← thread 0
- ← thread 0
- ← thread 1
- ← thread 1

```

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

factorial(i) =

Threads access data in a disordered way

↓

1	
2	← thread 0
6	← thread 0
0	← thread 1
0	← thread 1
0	← thread 2
0	← 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

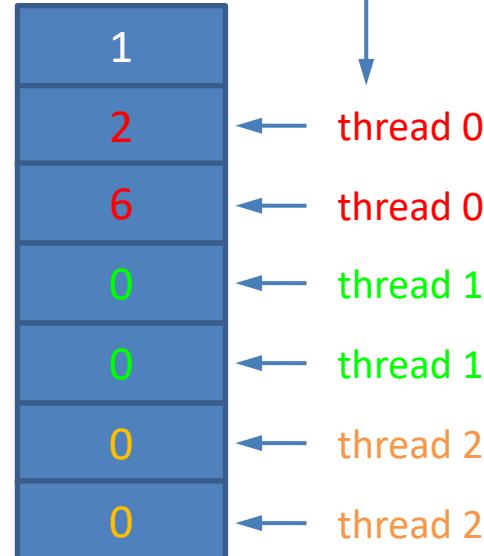
This is called **Race Condition**

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.

Threads access data in a disordered way

factorial(i) =



The diagram illustrates a race condition in a shared memory space. A vertical stack of eight cells represents memory. Thread 0 (blue arrow) writes values 2, 6, and 0 to the first three cells. Thread 1 (green arrow) writes values 0 and 0 to the fourth and fifth cells. Thread 2 (orange arrow) writes values 0 and 0 to the sixth and eighth cells. The final state shows the array containing [1, 2, 6, 0, 0, 0, 0, 0], which is incorrect for factorial(6).

1
2
6
0
0
0
0
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
```

$y(i) =$

2
4
6
8
10
12

$y(i) =$

x	$y(i)$
1	

← thread 0

Another example of **Race Condition**

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

    x = i
    y(i) = x * 2

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

$y(i) =$	2
	4
	6
	8
	10
	12

$y(i) =$

x	$y(i)$
1	2

← thread 0

Another example of **Race Condition**

```

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

   x = i
   y(i) = x * 2

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

```

$y(i) =$	2
	4
	6
	8
	10
	12

$y(i) =$

x	$y(i)$
1	2
2	

← thread 0  
← thread 0

Another example of **Race Condition**

```

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

   x = i
   y(i) = x * 2

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

```

$y(i) =$	2
	4
	6
	8
	10
	12

$y(i) =$

x	y(i)
1	2
2	
5	

← thread 0  
← thread 0  
← thread 2

Another example of **Race Condition**

```

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

   x = i
   y(i) = x * 2

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

```

$y(i) =$	2
	4
	6
	8
	10
	12

$y(i) =$

x	$y(i)$
1	2
2	10
5	

← thread 0  
← thread 0  
← thread 2

Another example of **Race Condition**

```

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

   x = i
   y(i) = x * 2

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

```

$y(i) =$	<table border="1"><tr><td>2</td></tr><tr><td>4</td></tr><tr><td>6</td></tr><tr><td>8</td></tr><tr><td>10</td></tr><tr><td>12</td></tr></table>	2	4	6	8	10	12
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

Another example of **Race Condition**

```

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

   x = i
   y(i) = x * 2

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

```

$y(i) =$	<table border="1"><tr><td>2</td></tr><tr><td>4</td></tr><tr><td>6</td></tr><tr><td>8</td></tr><tr><td>10</td></tr><tr><td>12</td></tr></table>	2	4	6	8	10	12
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

Another example of **Race Condition**

```

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

   x = i
   y(i) = x * 2

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

```

$y(i) =$	<table border="1"><tr><td>2</td></tr><tr><td>4</td></tr><tr><td>6</td></tr><tr><td>8</td></tr><tr><td>10</td></tr><tr><td>12</td></tr></table>	2	4	6	8	10	12
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

Another example of **Race Condition**

```

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

   x = i
   y(i) = x * 2

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

```

$y(i) =$	<table border="1"><tr><td>2</td></tr><tr><td>4</td></tr><tr><td>6</td></tr><tr><td>8</td></tr><tr><td>10</td></tr><tr><td>12</td></tr></table>	2	4	6	8	10	12
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

Another example of **Race Condition**

```

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

   x = i
   y(i) = x * 2

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

```

$y(i) =$	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

Another example of **Race Condition**

```

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

   x = i
   y(i) = x * 2

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

```

$y(i) =$	<table border="1"><tr><td>2</td></tr><tr><td>4</td></tr><tr><td>6</td></tr><tr><td>8</td></tr><tr><td>10</td></tr><tr><td>12</td></tr></table>	2	4	6	8	10	12
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

Another example of **Race Condition**

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

```

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

   x = i
   y(i) = x * 2

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

```

	2
	4
	6
	8
	10
	12

y(i) =

x <sub>0</sub>	x <sub>1</sub>	x <sub>2</sub>	y(i)
1			2
2			4
	3		6
	4		8
	5	10	
	6	12	

y(i) =

← thread 0  
← thread 0  
← thread 1  
← thread 1  
← thread 2  
← thread 2

## 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.

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.

```
!$omp parallel default(None) private(REDACTED) &
!$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
```

You don't need to remember this.  
Just declare everthing!

With more practice you can skip declaring  
shared variables.

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 construct

Consider now this code. Is it correct?

```
sum = 0

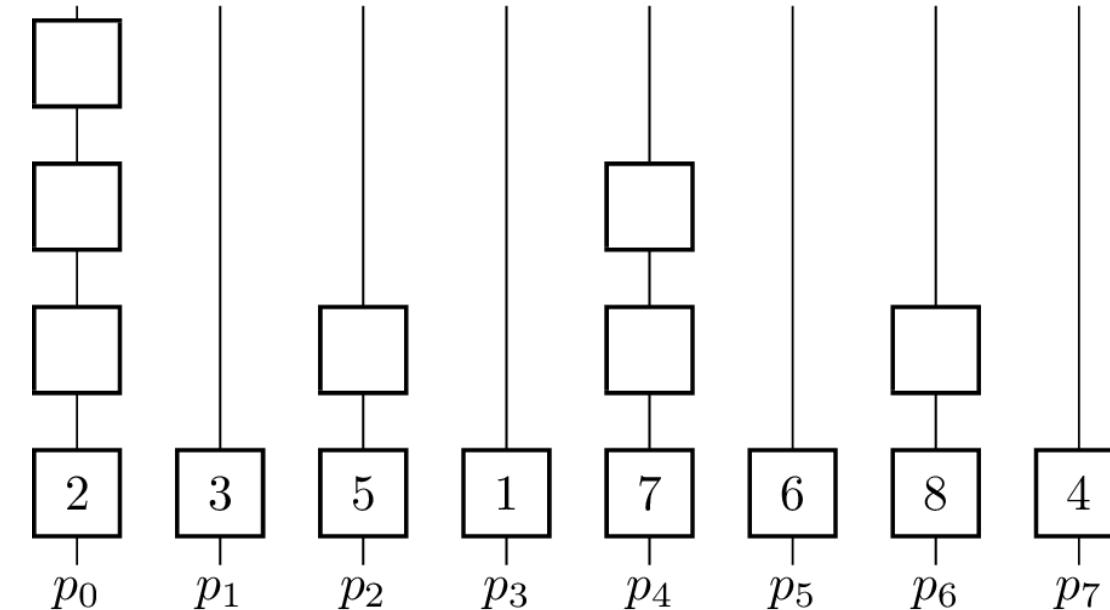
 !$omp parallel
 !$omp do reduction(+: sum)
 do i = 1, 6

     sum = sum

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

This allows each thread to store their partial sum. When they exit, it groups all partial sums and performs a reduction operation (+ in this case).

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



Source: wikipedia

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

# Connecting to the cloud cluster:

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

Connect to the cluster:

`ssh <username>@<cluster_name>`

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

The password is: openmp\_course

Clone the repository of the exercises:

`git clone https://github.com/acavelan/openmp_course_2021.git`

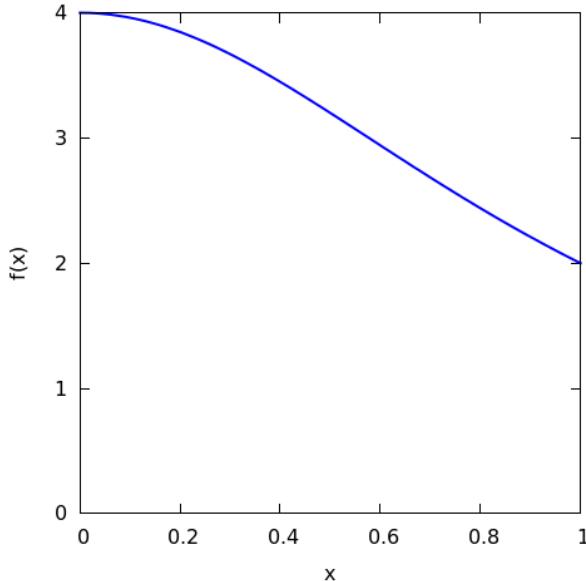
Enter in openmp\_course/:

`cd openmp_course`

## Exercise: Numerical integration

We know that:

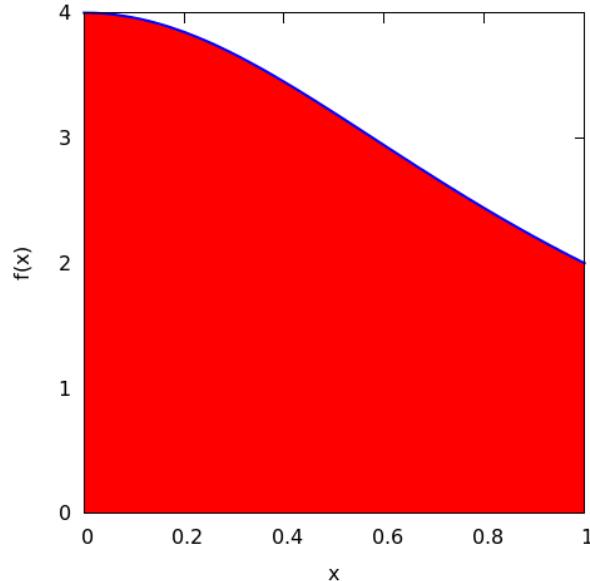
$$\int_0^1 \frac{4}{(1+x^2)} dx = \pi$$



## Exercise: Numerical integration

We know that:

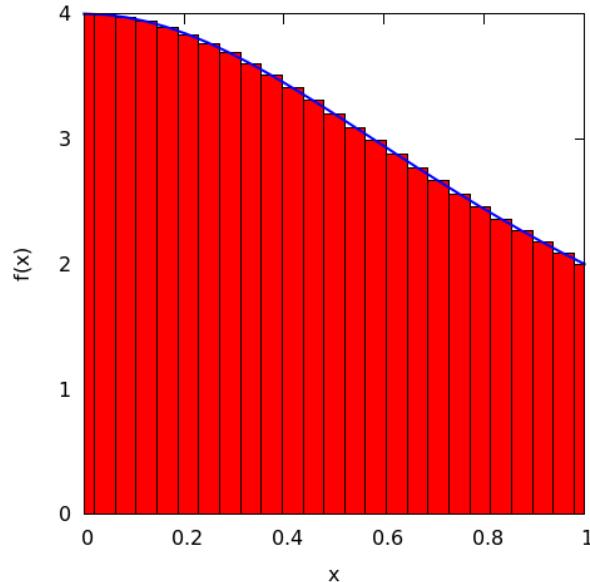
$$\int_0^1 \frac{4}{(1+x^2)} dx = \pi$$



## Exercise: Numerical integration

We know that:

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



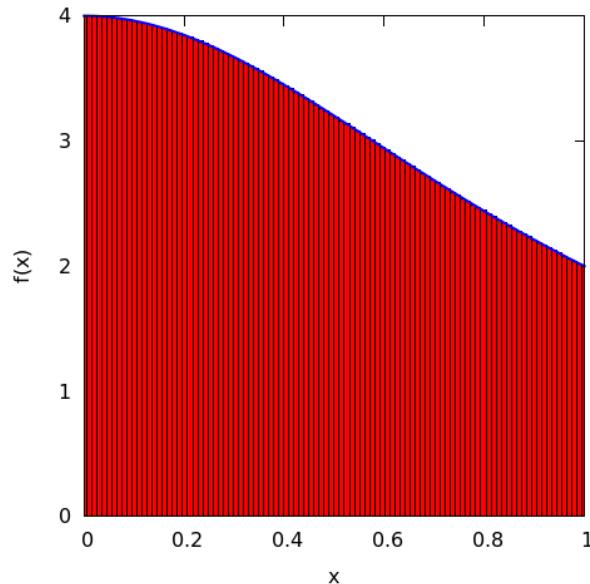
Using the composite formula we can approximate the integral as:

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

## 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
```

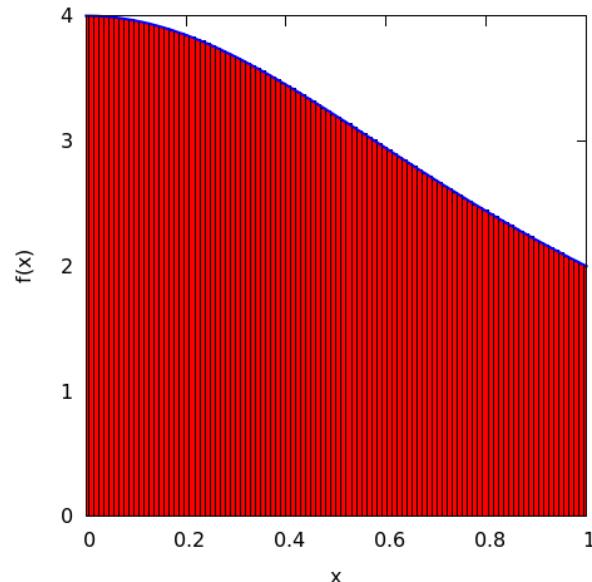
Compile with: **gfortran -fopenmp pi.f90**

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
```

Compile with: `gfortran -fopenmp a.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:

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

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
```

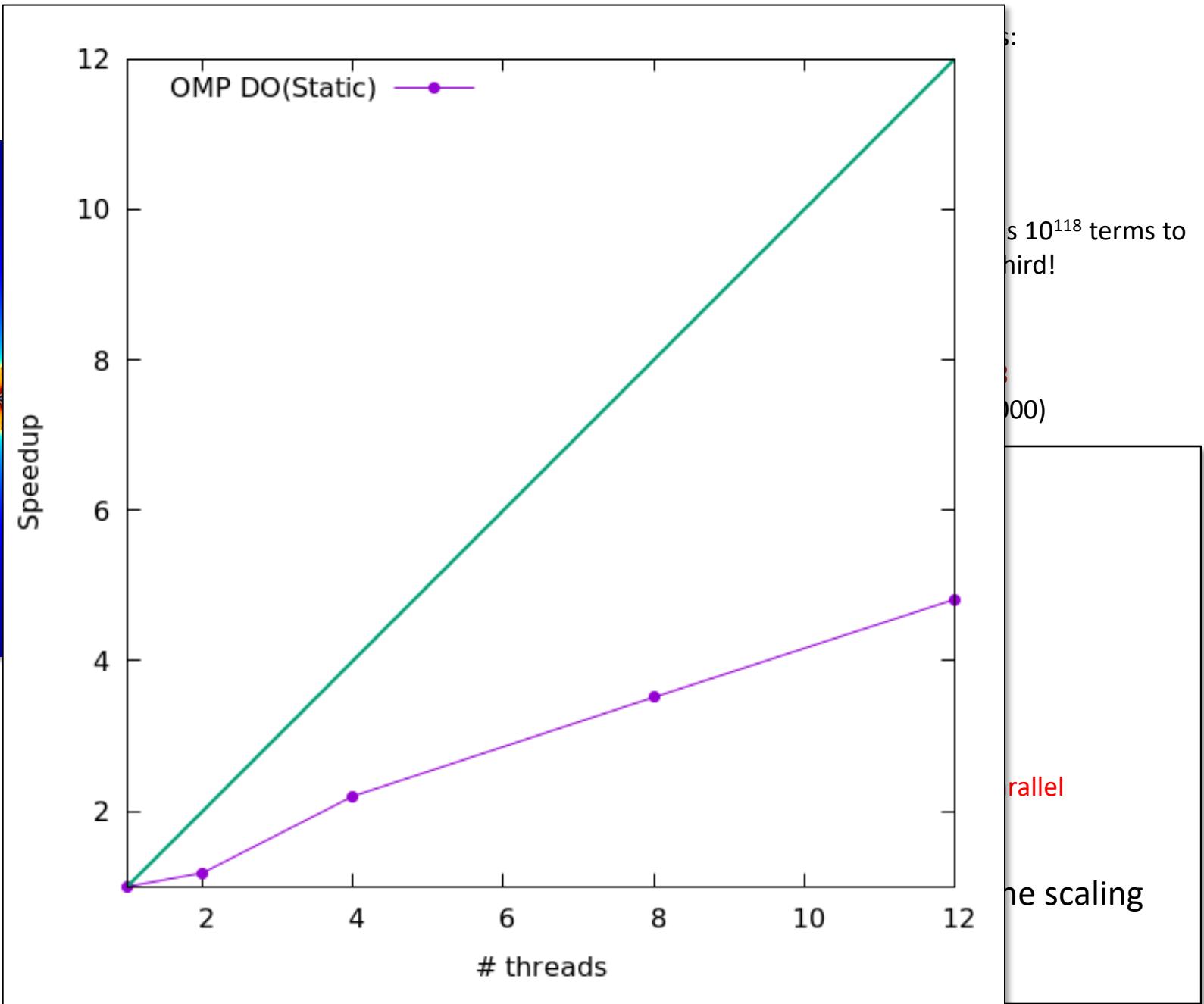
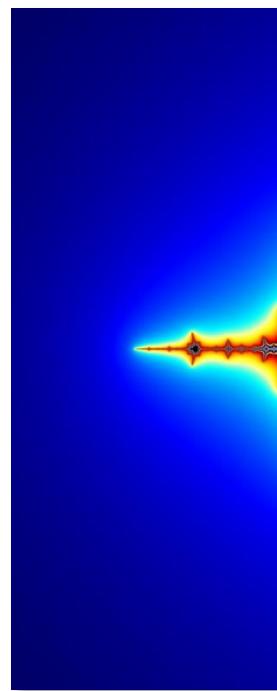
What about subroutines? We can use **orphaned directives**

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

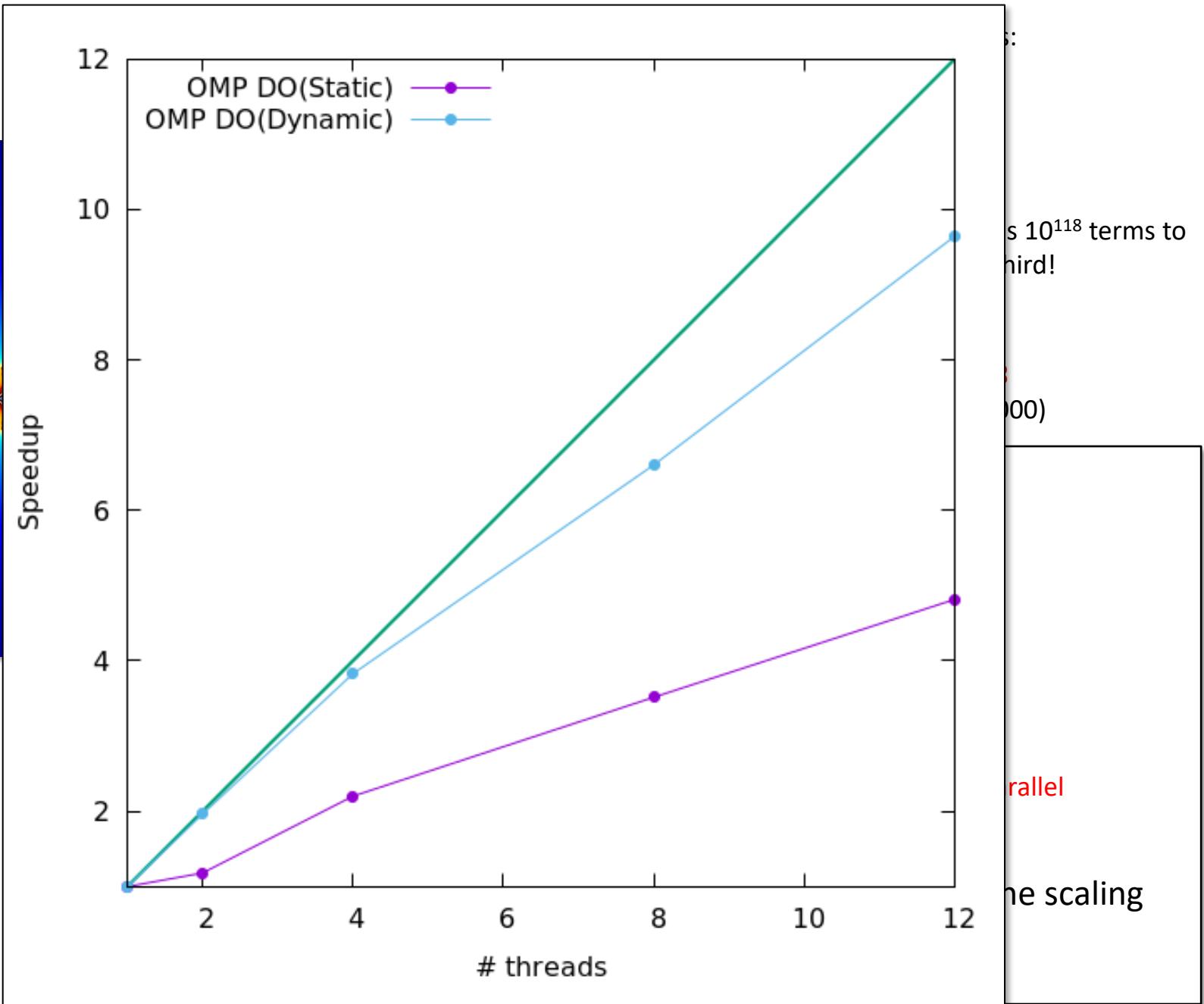
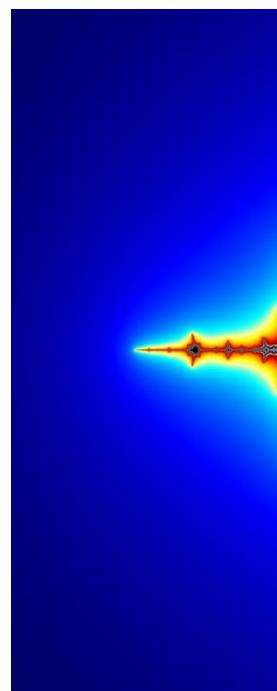


```
subroutine calculate_manythings()  
  
    !$omp do schedule(static)  
        do i=1,n  
            <calculations>  
        enddo  
    !$omp end do  
  
    return  
end subroutine
```

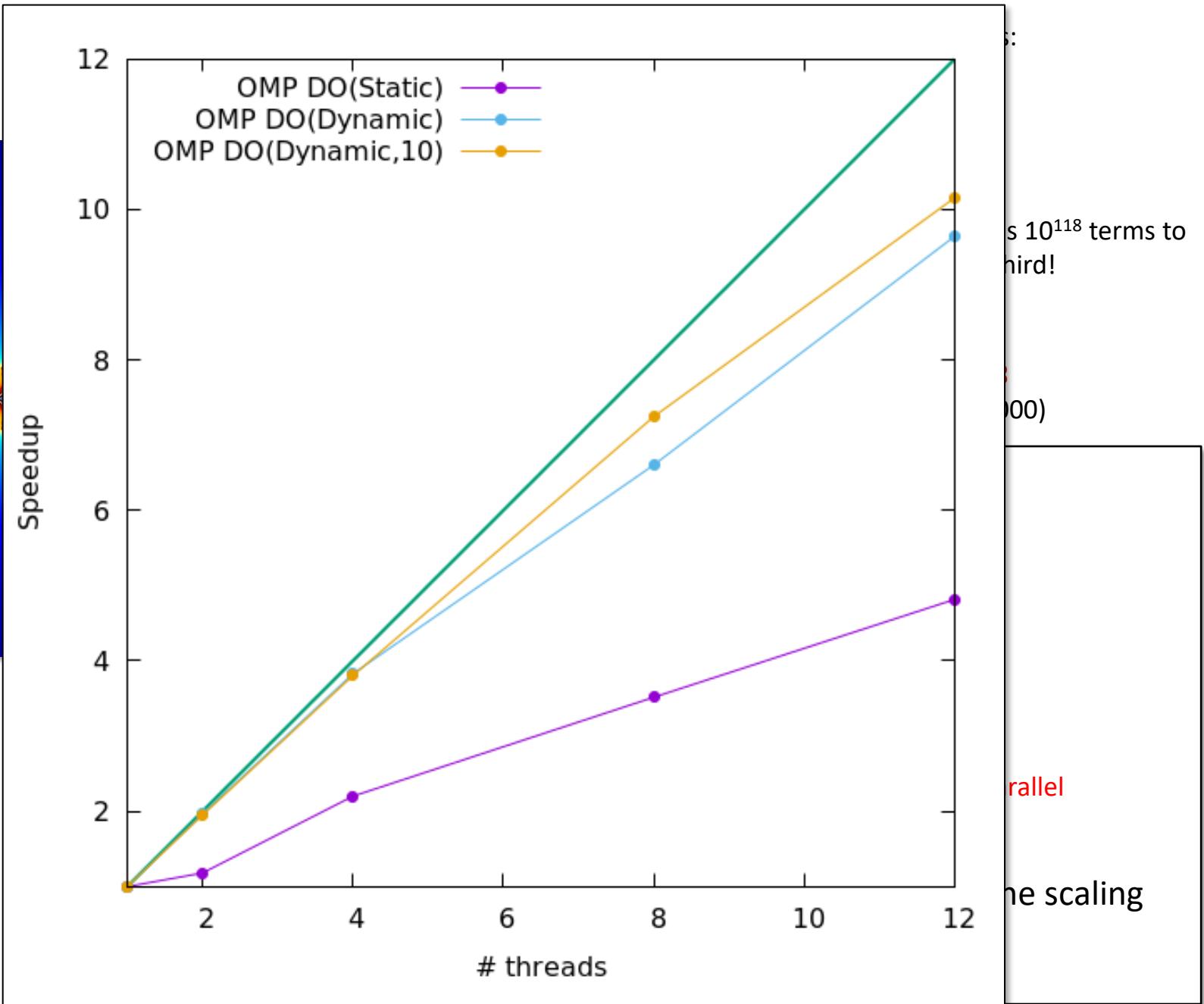
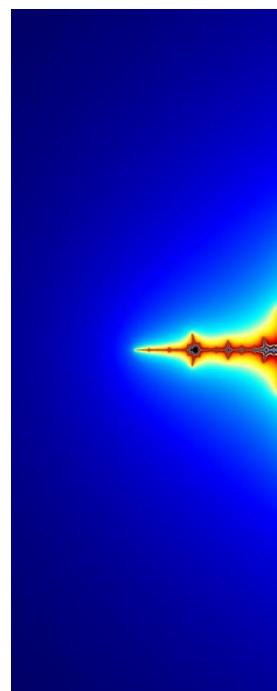
## Mandelbrot



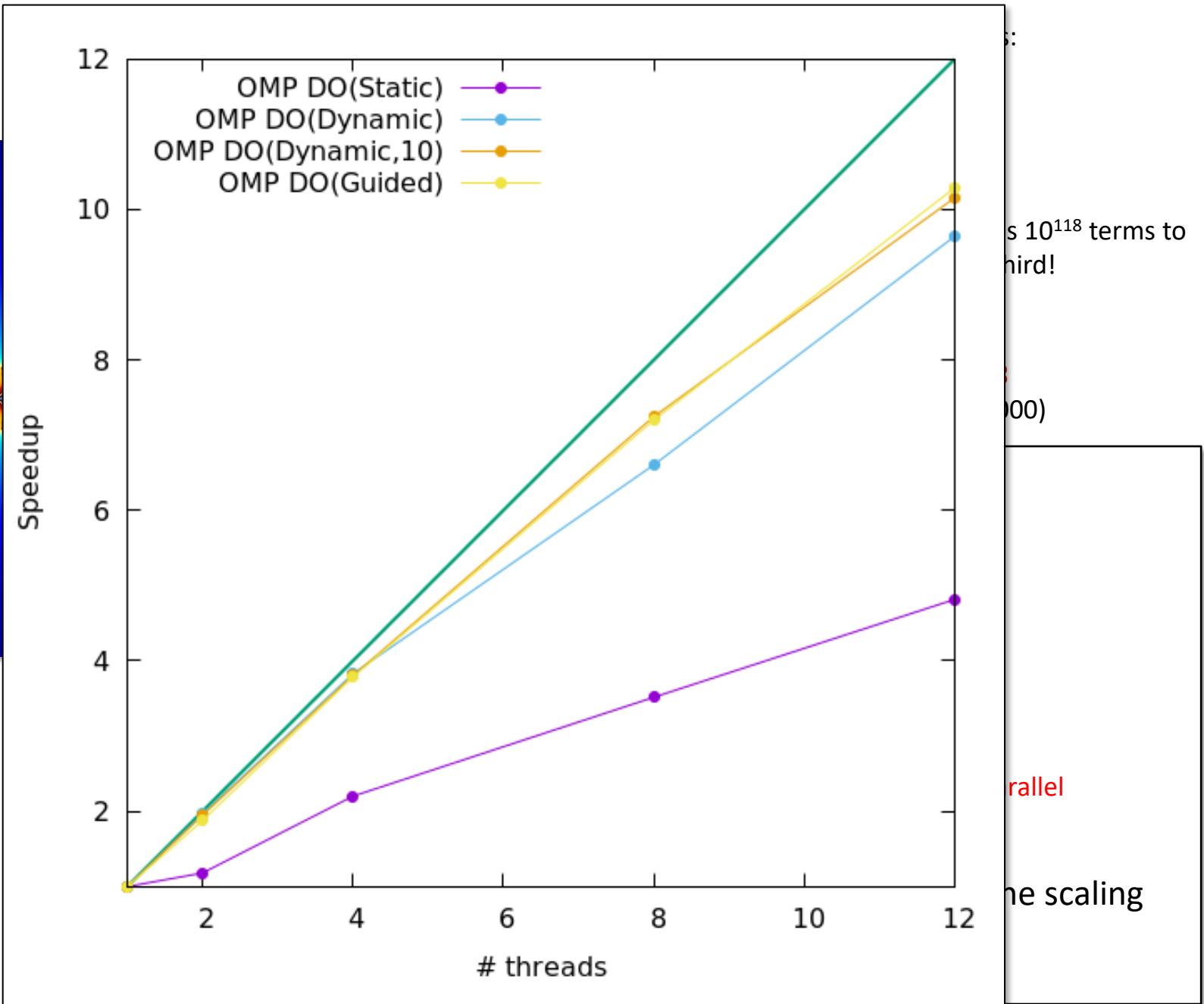
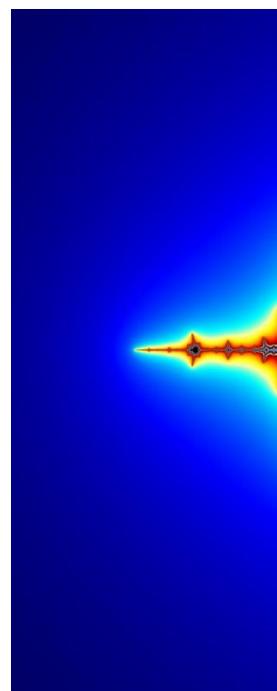
# Mandelbrot



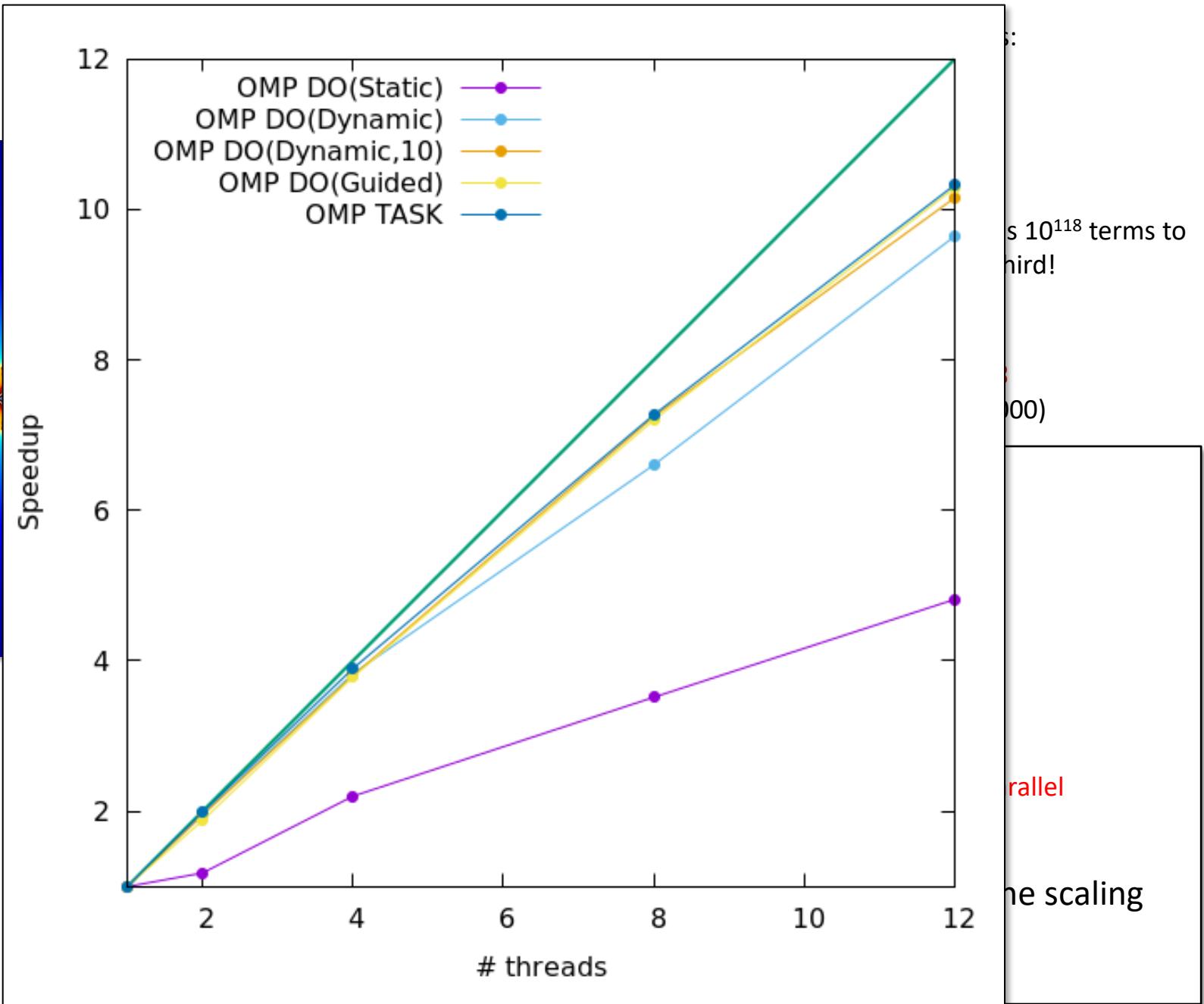
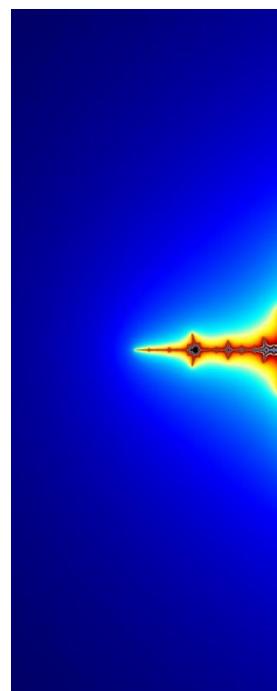
# Mandelbrot

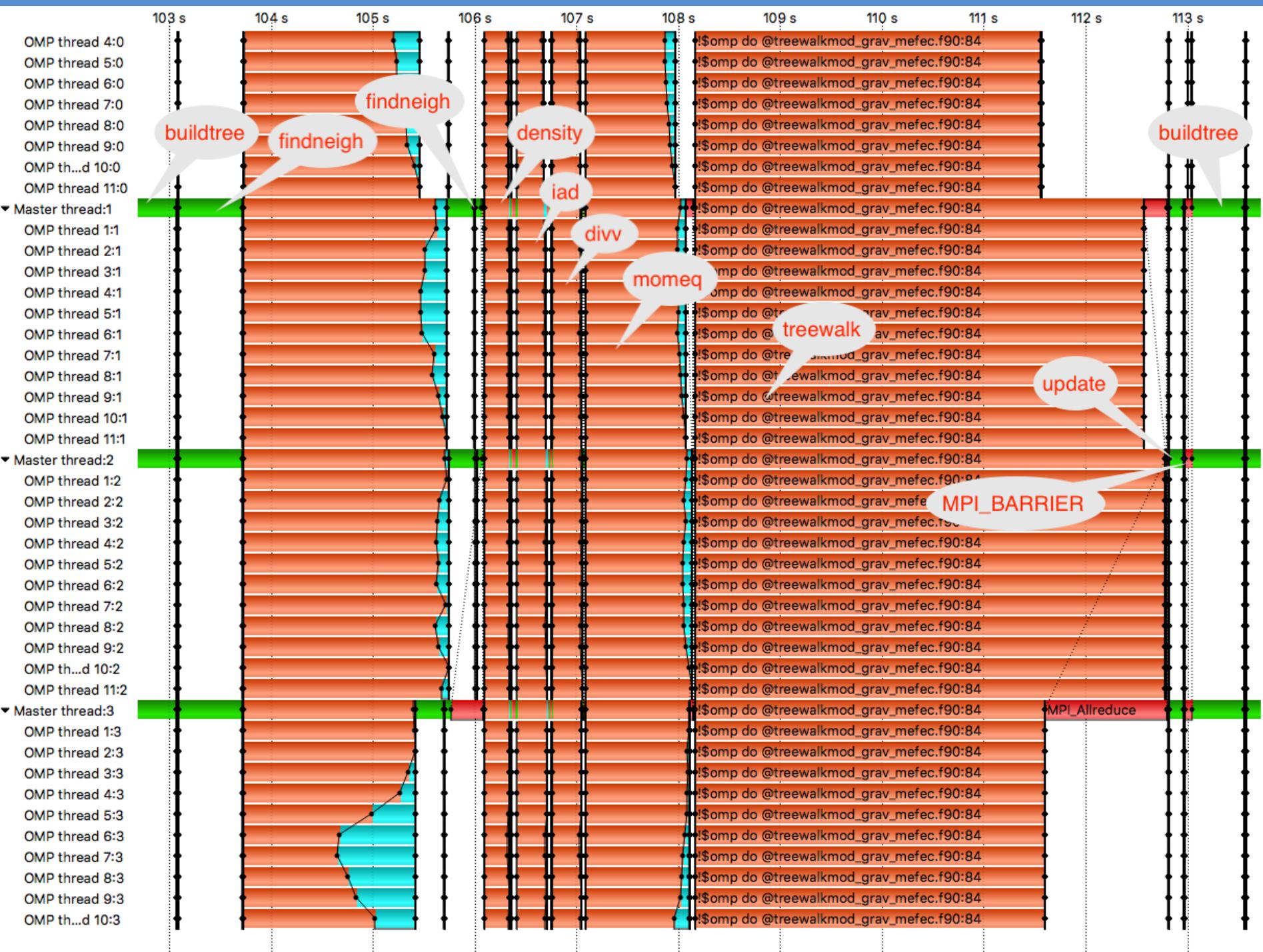


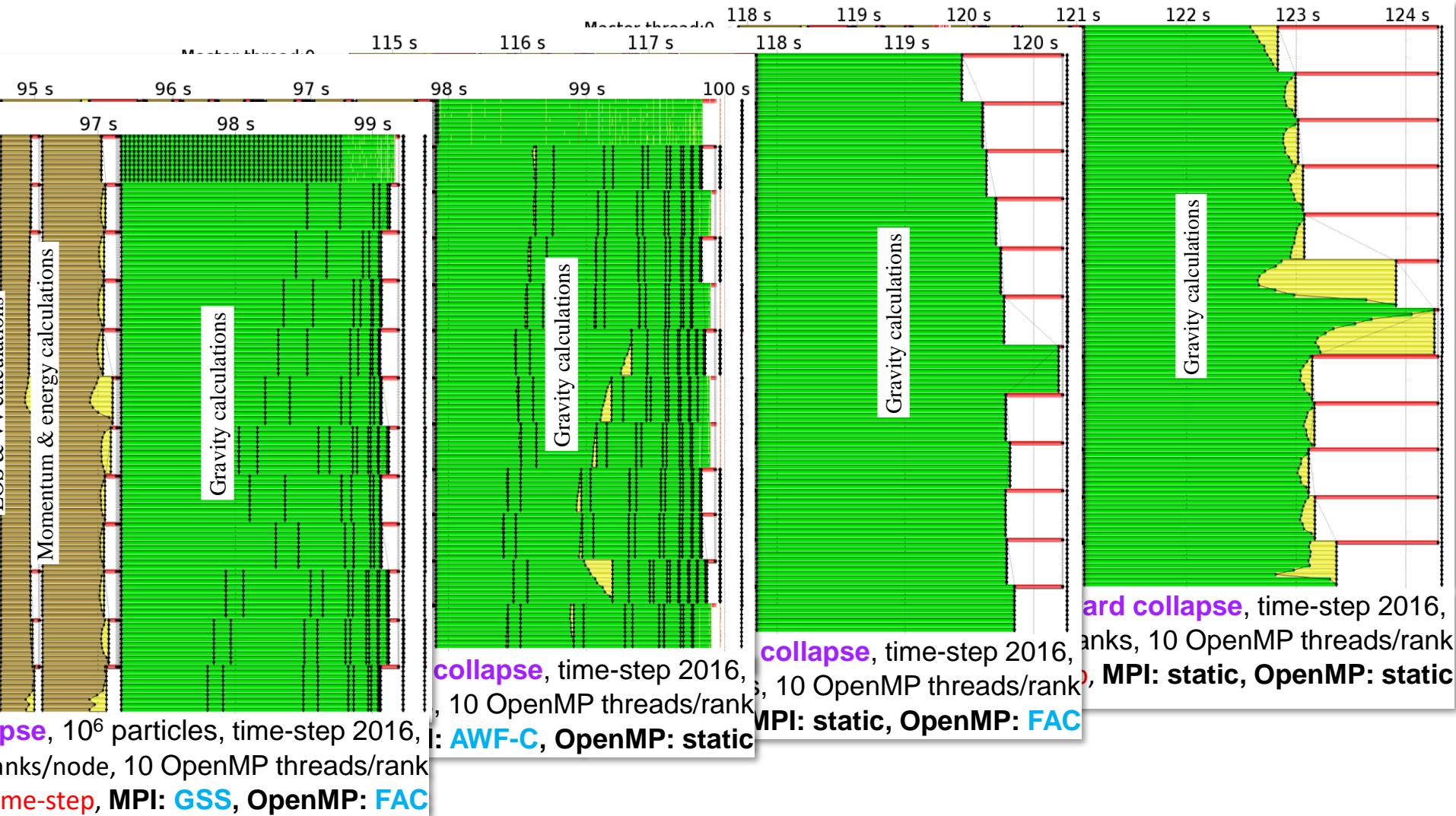
## Mandelbrot



## Mandelbrot

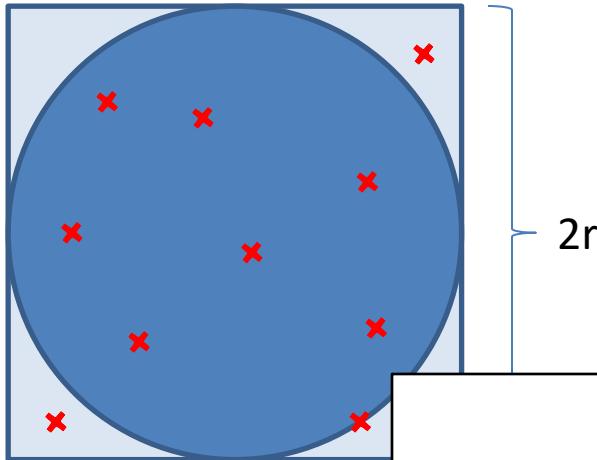






Two-level dynamic load balancing for high performance scientific applications  
 Mohammed, A. et al (Siam PP20)

## 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  $\pi$  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
```



random generation openmp fortran



All



Images



Videos



News



Shopping



More

Settings

Tools

About 356'000 results (0.40 seconds)

Search Site  Search  only in current section



# CMISS

Home CM openCMISS CMGUI Documentation Data

Log in Register

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

Navigation

## Random number generation with OpenMP

News

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.

[deleted wiki pages](#)

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.

- [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](#)

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** [<a href="#test\\_random\\_number.f90">test\\_random\\_number.f90</a>](#) 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.

candidate release  
2012-01-31  
  
Zinc 0.7.0.0 beta  
release  
2011-11-10  
  
[More news...](#)

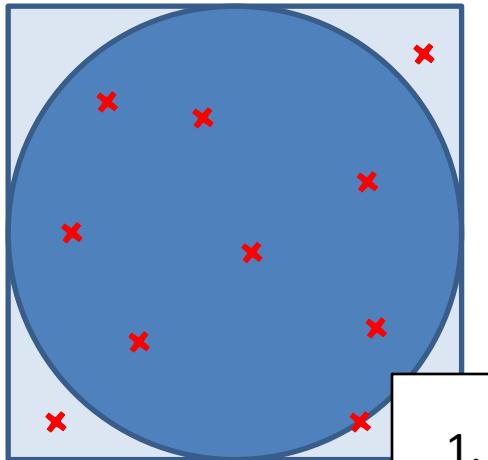
You visited this page on 5/2/20.

[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>.

## 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  $\pi$  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
```

# PART II



Numba makes Python code fast

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(10000000000):
        j=j+1
    print ('func1: finishing',j)

def func2():
    j=0
    print ('func2: starting')
    for i in range(10000000000):
        j=j+1
    print ('func2: finishing',j)

def func3():
    j=0
    print ('func3: starting')
    for i in range(10000000000):
        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	svstemd

```
ruben@jarvis:~/test/parallelpython$ time python test.py
func1: starting
func2: starting
func3: starting
func3: finishing 10000000000
func2: finishing 10000000000
func1: finishing 10000000000
[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	svstemd

```
ruben@jarvis:~/test/parallelpython$ time python test.py
func1: starting
func2: starting
func3: starting
func3: finishing 10000000000
func2: finishing 10000000000
func1: finishing 10000000000
[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.

```
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: 1249999985000000]
2249999700000000
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()
```

```
[cabezón@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: 1249999985000000]
2249999700000000

real 0m12.819s
user 0m27.130s
sys 0m8.865s
[cabezón@login10 openmp]$
```

```
[cabezón@login10 openmp]$ source numbatest/bin/activate
(numbatest) [cabezón@login10 openmp]$ time python python_numba.py
func1: starting
func1: finishing 2.24999988355443e+16

real 0m1.142s
user 0m1.205s
sys 0m1.294s
(numbatest) [cabezón@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

```
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()
```

```
[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: 1249999985000000]
2249999700000000
real 0m12.819s
user 0m27.130s
sys 0m8.865s
```

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
```

```
saktho00@worker04:~/temp/numbatest
File Edit View Search Terminal Help
genotypes[cellID,2+targetoffset]=maxsnps#genotypes[c
ellID,3+targetoffset] + 1
#genotypes[cellThreadID,1+targetoffset]=gen
#genotypes[cellID,2+targetoffset]=genotypes[cellID,2+targetoffse
t] + 1
    cuda.syncthreads()
    #cuda.atomic.add(syncObj,0,1)
    #while(syncObj[0]<N*(numbIt+1+gen)):
    #    1+1
@jit
def pairwisedist(genotypes):
    difflist = []
    N=int(genotypes.shape[0])
    L=int(genotypes.shape[1]/2)
    for i in range(N-1):
        for j in range(i+1,N):
            delta=0
            diff = 0
            for k in range(L):
                delta = genotypes[i,k] ^ genotypes[j,k]
                for m in range(64):
                    if int64(delta) & 1 == 1:
                        diff = diff + 1
                delta = math.floor(delta / 2)
                if delta == 0:
                    break
            difflist.append(diff/(L*64))
    return(difflist)

@jit
def closedist(genotypes):
    "recombSim.py" 329L, 11141C written
```

Thomas Sakoparnig

```
saktho00@worker04:~/temp/numbatest
File Edit View Search Terminal Help
[saktho00@sgi27 numbatest]$ vi recombSim.py
[saktho00@sgi27 numbatest]$ python recombSim.py
ttime: 3.4206590335816145 ←
0.0170744024235
[saktho00@sgi27 numbatest]$ vi recombSim.py
[saktho00@sgi27 numbatest]$ python recombSim.py
ttime: 2637.8457502331585 ←
0.0178537920918
[saktho00@sgi27 numbatest]$
```

## Exercise: Numerical integration

Parallelize the following program with openMP

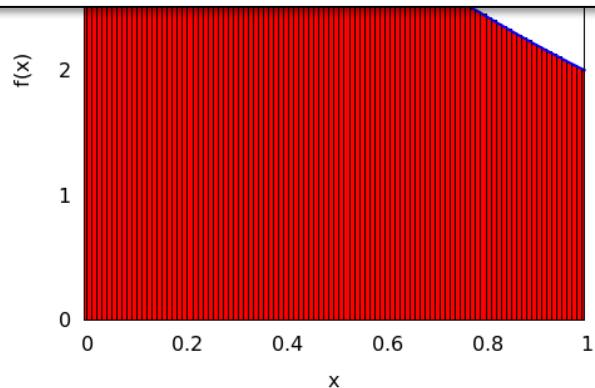
```
steps = 1000000000
dx = 1./dble(steps)
sum = 0.

!$omp parallel private(i,x)
!$omp do reduction(+:sum)
do i = 1, steps
    x = (dble(i)-0.5)*dx ←
    sum = sum + 4./(1. + x*x)
end do
!$omp end do
!$omp end parallel

sum = sum * dx
```



What about `dx`? Shouldn't be `firstprivate`?  
Otherwise is not initialized!



# of threads	Wall-clock time (s)	Result
1	12.169	3.1415926535899708
2	6.217	3.1415926535899010
4	3.212	3.1415926535898211
8	1.769	3.1415926535897687
Real value		3.1415926535897932

Using the composite formula we can approximate the integral as:

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

`@jit` tells numba to **compile** the function (just-in-time)

`@jit(nopython=True)` tells numba to fail if this cannot be done,  
otherwise numba will fallback to python.

`@njit` is equivalent to `@jit(nopython=True)`

```
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

@jit(nopython=True) # Set "nopython" mode for best performance, equivalent to @njit
def go_fast(a): # Function is compiled to machine code when called the first time
    trace = 0.0
    for i in range(a.shape[0]):    # Numba Likes Loops
        trace += np.tanh(a[i, i]) # Numba Likes NumPy functions
    return a + trace            # Numba Likes NumPy broadcasting

print(go_fast(x))
```

```
from numba import jit
import pandas as pd

x = {'a': [1, 2, 3], 'b': [20, 30, 40]}

@jit
def use_pandas(a): # Function will not benefit from Numba jit
    df = pd.DataFrame.from_dict(a) # Numba doesn't know about pd.DataFrame
    df += 1                  # Numba doesn't understand what this is
    return df.cov()           # or this!

print(use_pandas(x))
```

`@njit(parallel=True)` tells numba to parallelize the loops it will find inside the function. **Automatically.**

```
from numba import njit, prange
@njit(parallel=True)
def prange_test(A):
    s = 0
    # Without "parallel=True" in the
    # the prange statement is equivalent
    for i in prange(A.shape[0]):
        s += A[i]
    return s
```

What about private and shared variables?  
What about reductions?

A reduction is inferred automatically for binary function/operator

1. Parallelize and run the code in pi.py  
`export NUMBA_NUM_THREADS=4  
python pi.py`

2. Compare the execution time to Fortran/C++  
`time python pi.py`

Fortran

```
steps = 1000000000
dx = 1./dble(steps)
sum = 0.

!$omp parallel private(i,x)
!$omp do reduction(+:sum)
do i = 1, steps
    x = (dble(i)-0.5)*dx
    sum = sum + 4.0/(1.0 + x*x)
end do
!$omp end do
!$omp end parallel

sum = sum * dx
```

```
1 #include <cstdio>
2 #include <math.h>
3
4 int main()
5 {
6     const int steps = 1000000000;
7     const double dx = 1.0 / steps;
8     const double pi = 3.141592653589793;
9
10    double sum = 0.0;
11
12    #pragma omp parallel for reduction(+:sum)
13    for(int i=0; i<steps; i++)
14    {
15        double x = (i - 0.5) * dx;
16        sum += 4.0 / (1.0 + x*x);
17    }
18
19    sum *= dx;
20
21    printf("%.15f %.15e\n", sum, abs(pi-sum));
22
23    return 0;
}
```

C/C++

```
1 import math
2 from numba import njit, prange
3
4 @njit(parallel = True)
5 def pi(steps):
6     dx = 1./steps
7
8     lsum = 0.0
9     for i in prange(0, steps):
10         x = (i - 0.5) * dx
11         lsum = lsum + 4.0 / (1.0 + x*x)
12
13     lsum = lsum * dx
14
15     return lsum
16
17 lsum = pi(1000000000)
18 pi = math.acos(-1.0)
19
20 print(lsum, abs(pi - lsum)/pi)
```

Python (don't forget prange)

# PART III

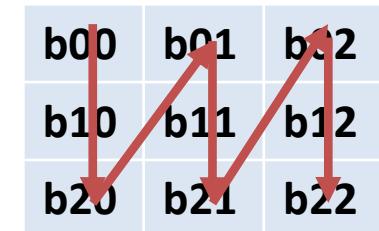
Optimizing the Matrix Multiplication  
+  
GPU Offloading

## 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];
```

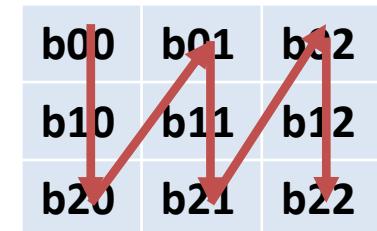


## Full example: matrix multiplication

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

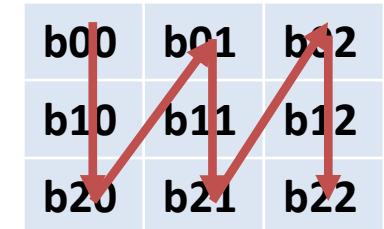
i	j	k
0	0	0

## 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

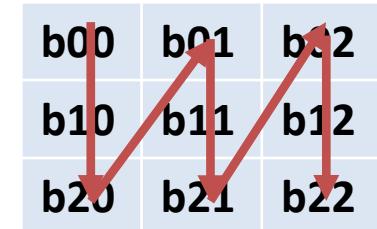
i	j	k
0	0	1

## 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

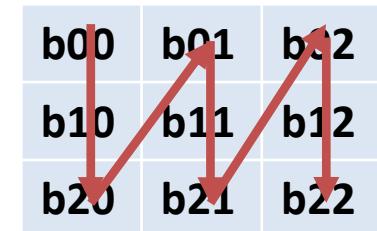
i	j	k
0	0	2

## 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										

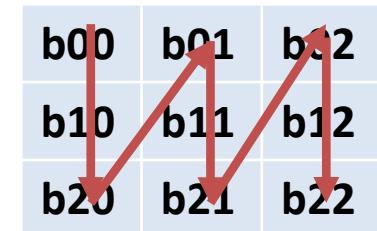
i	j	k
0	1	0

## 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

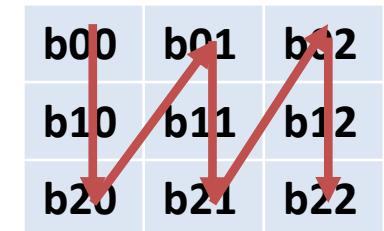
i	j	k
0	1	1

## 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

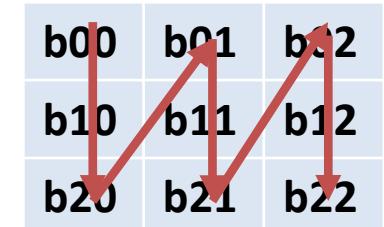
i	j	k
0	1	2

## 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

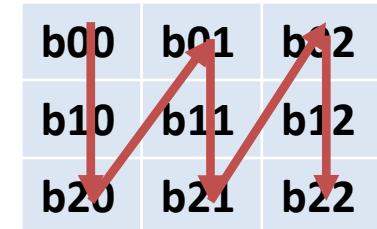
i	j	k
0	2	0

## 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

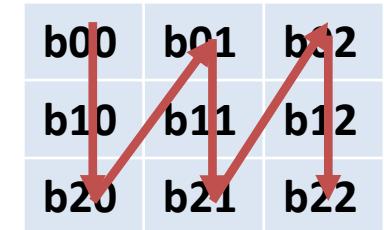
i	j	k
0	2	1

## 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

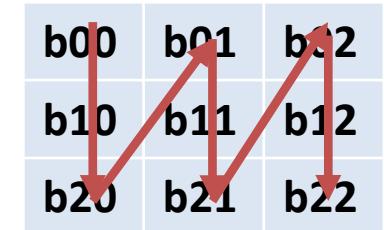
i	j	k
0	2	2

## 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

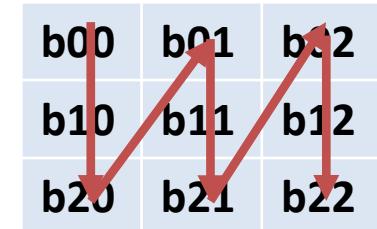
i	k	j
0	0	0

## Full example: matrix multiplication

Sub-optimal memory access will kill the performance...

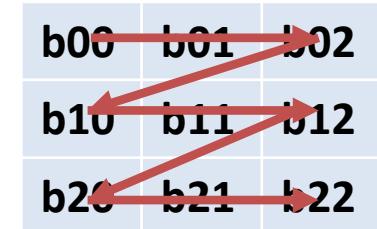
The less efficient way (column-major):

```
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

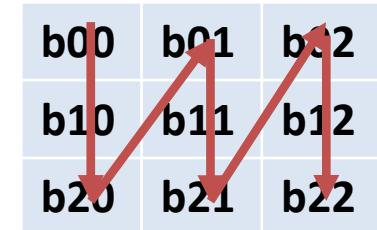
i	k	j
0	0	1

## Full example: matrix multiplication

Sub-optimal memory access will kill the performance...

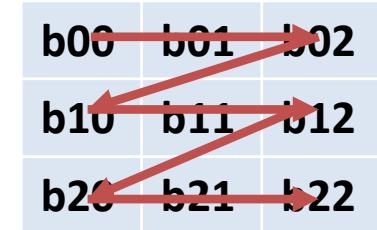
The less efficient way (column-major):

```
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

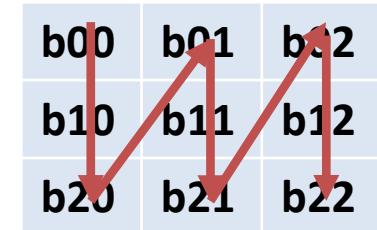
i	k	j
0	0	2

## Full example: matrix multiplication

Sub-optimal memory access will kill the performance...

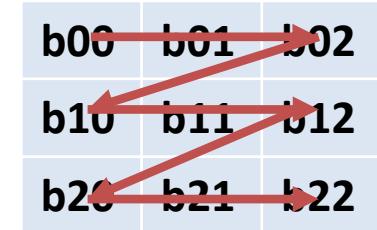
The less efficient way (column-major):

```
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];
```



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

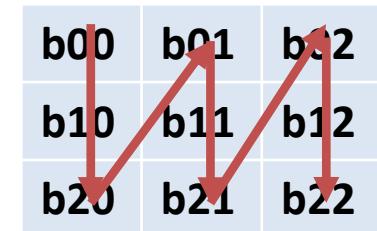
i	k	j
0	1	0

## Full example: matrix multiplication

Sub-optimal memory access will kill the performance...

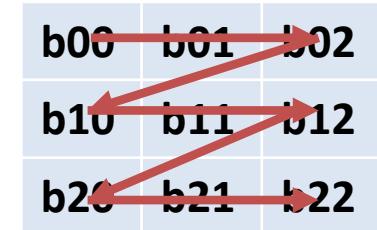
The less efficient way (column-major):

```
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

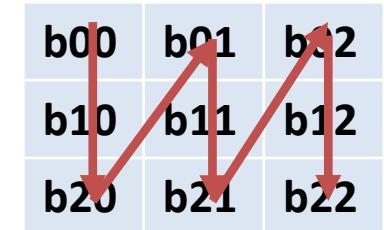
i	k	j
0	1	1

## Full example: matrix multiplication

Sub-optimal memory access will kill the performance...

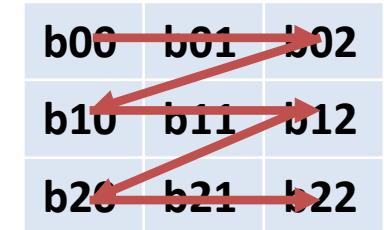
The less efficient way (column-major):

```
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

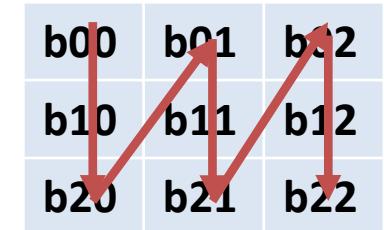
i	k	j
0	1	2

## 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

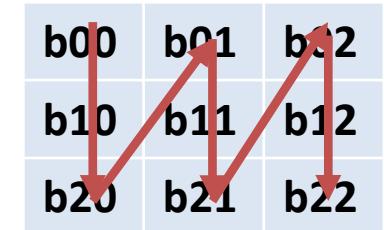
i	k	j
0	2	0

## 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];
```

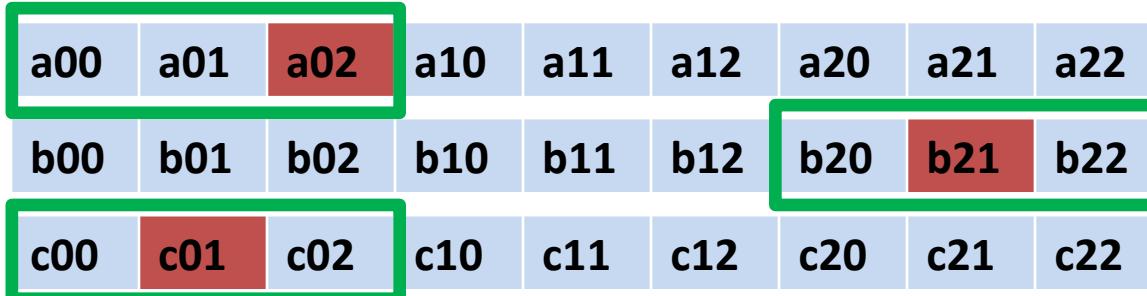


The better way (row-major):

```
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



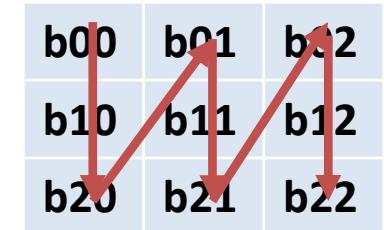
i	k	j
0	2	1

## Full example: matrix multiplication

Sub-optimal memory access will kill the performance...

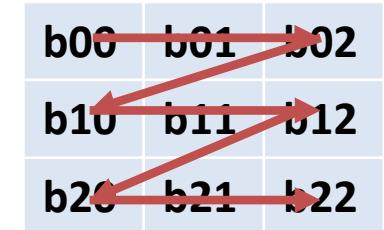
The less efficient way (column-major):

```
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
```

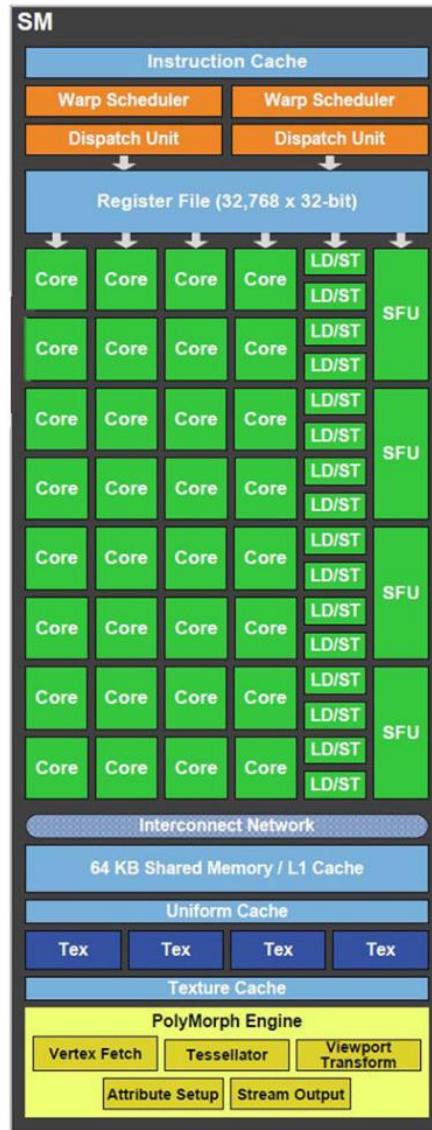
FORTRAN

```
#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];
```

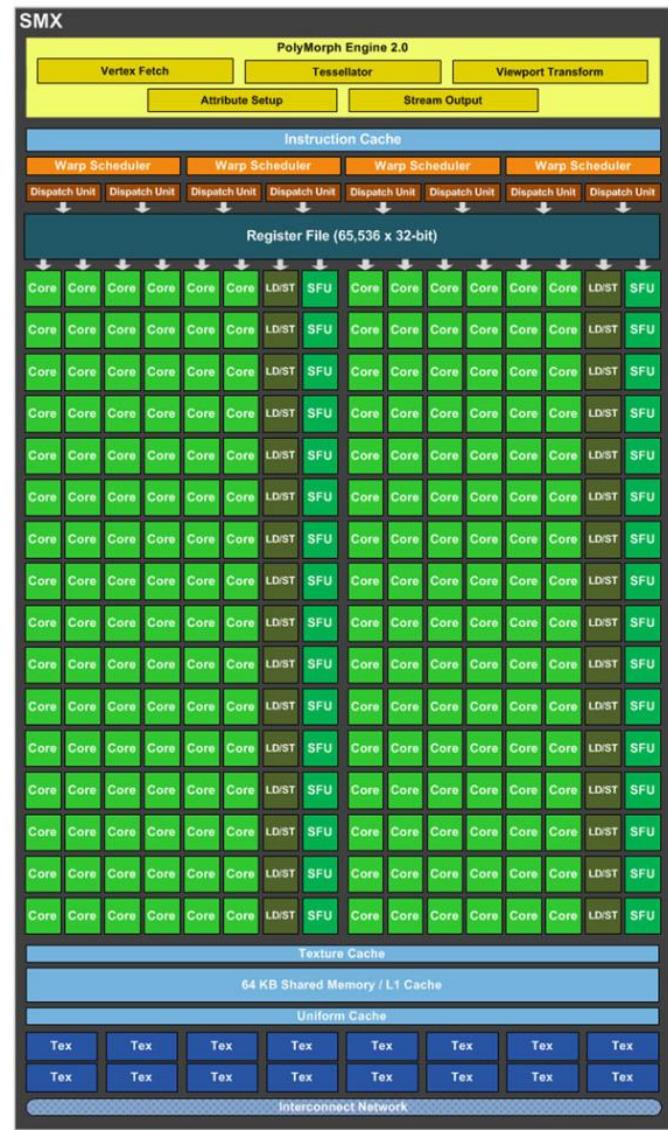
C++

# What about GPU?

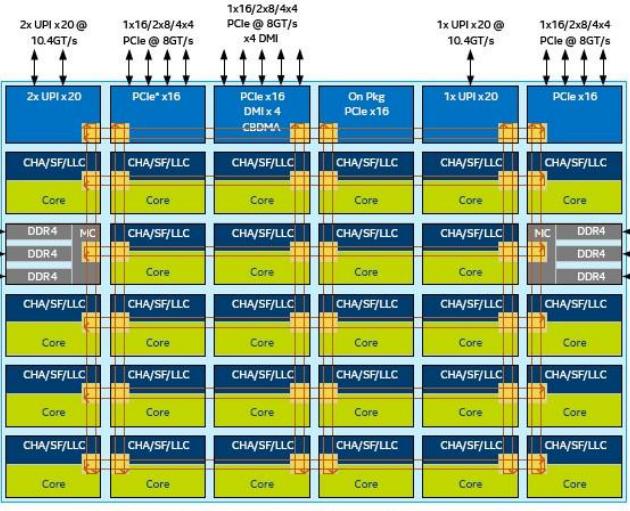
Tesla SM unit (2007)



Fermi SMX unit (2011)



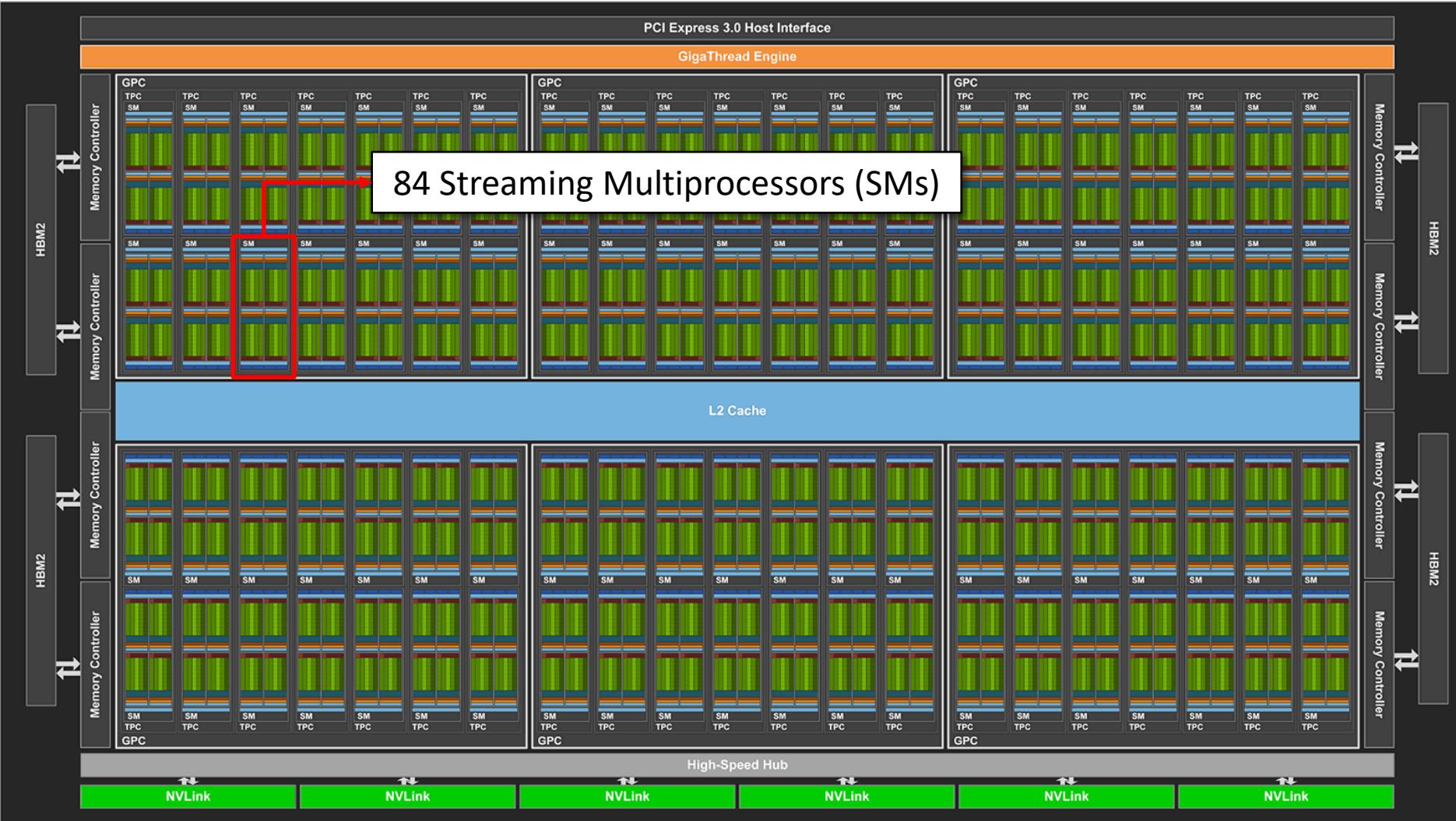
Intel Skylake (2017)



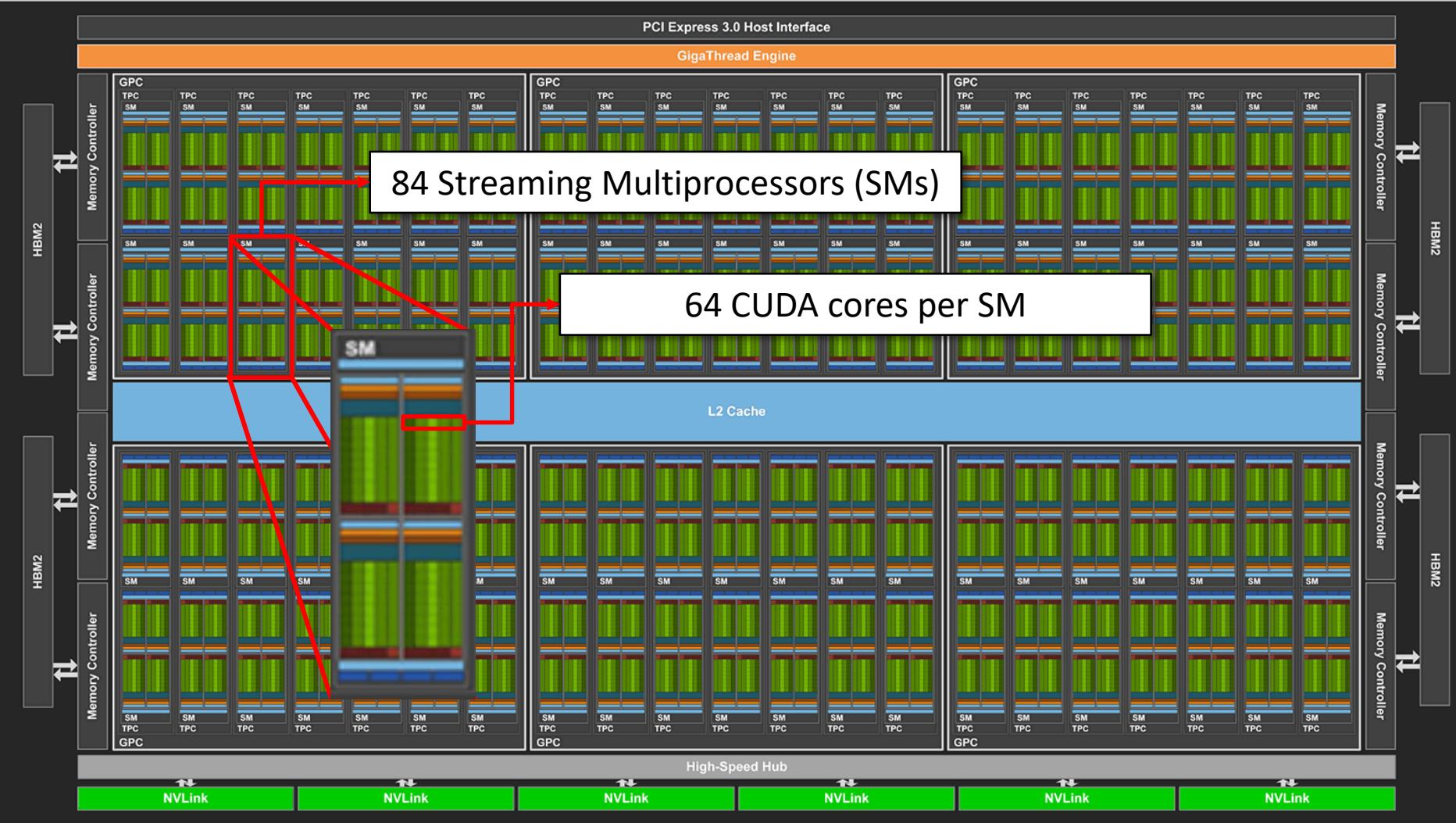
# Volta (2017)



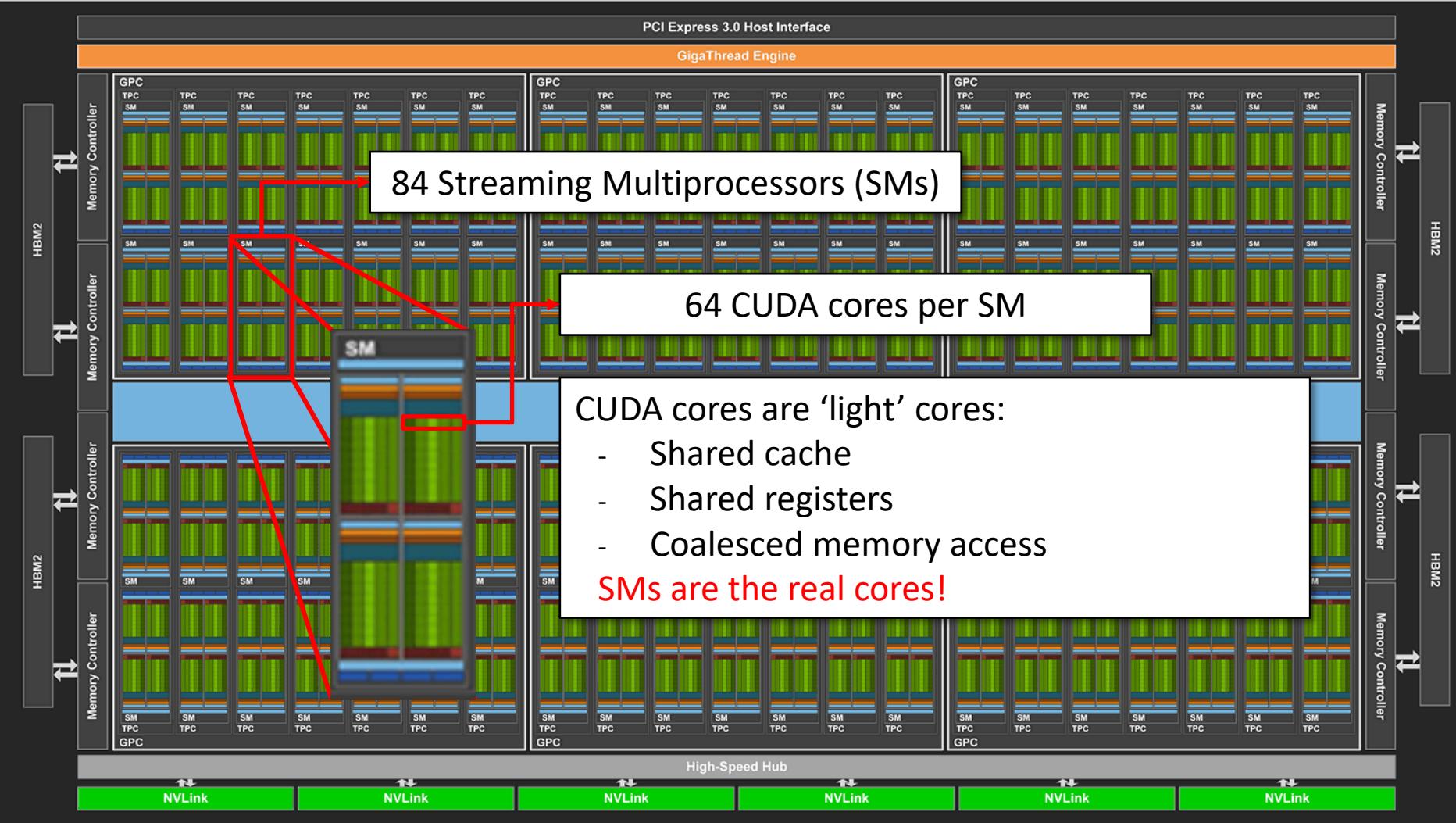
## Volta (2017)



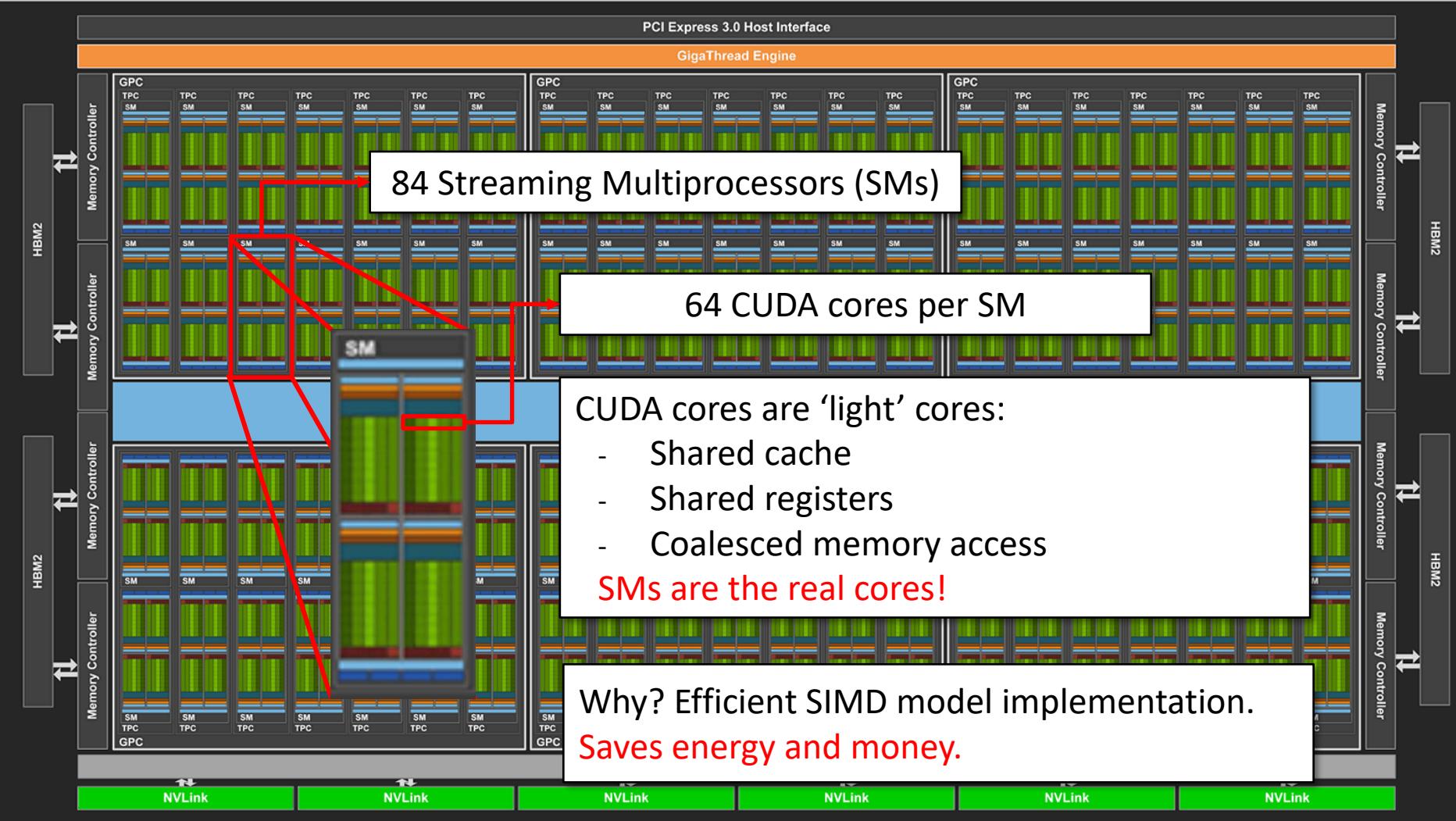
## Volta (2017)



## Volta (2017)



## Volta (2017)



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");  
}
```

You can control the transfer of data

omp\_is\_initial\_device() returns 0 when called from the accelerator

```
#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**.

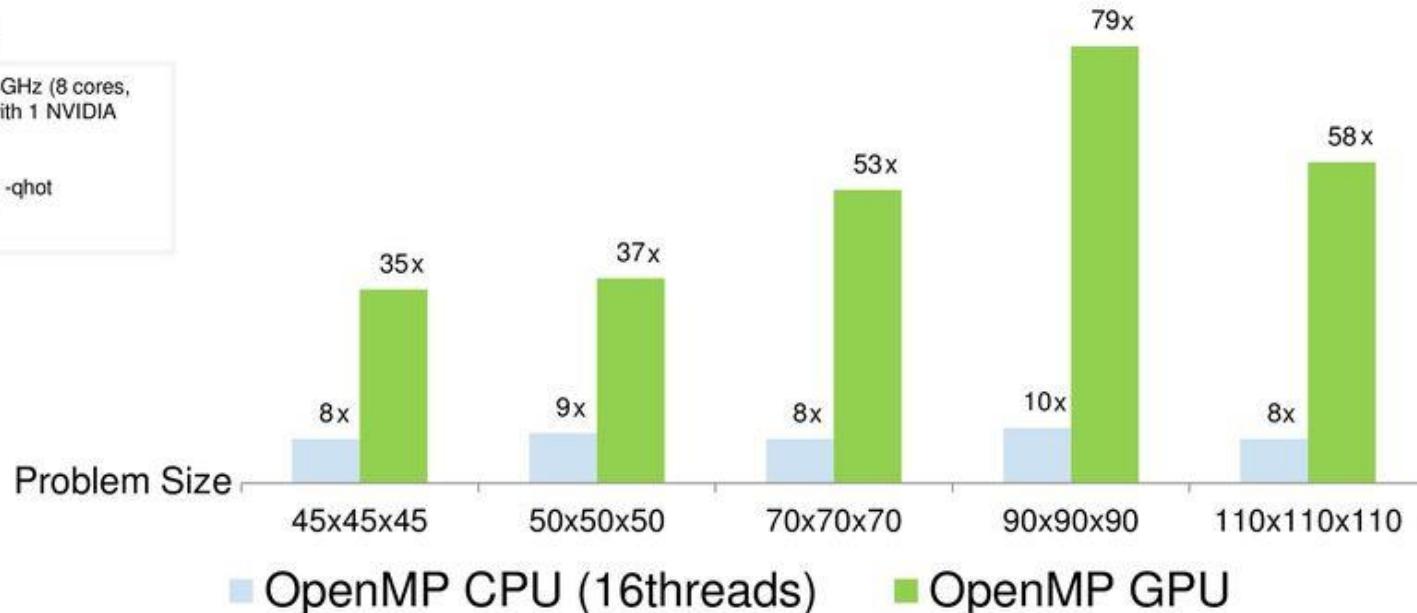
## Performance Results – End to End

**LULESH – Speedup Over Serial  
(Higher is Better)**

### Test Specs

2 Power8 sockets @ 4GHz (8 cores, with 8 threads each) with 1 NVIDIA Pascal P100 GPU.

Compiler Options: -O3 -qhot  
-qsmp=omp -qoffload\*  
\* Where applicable



## 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];
}
```

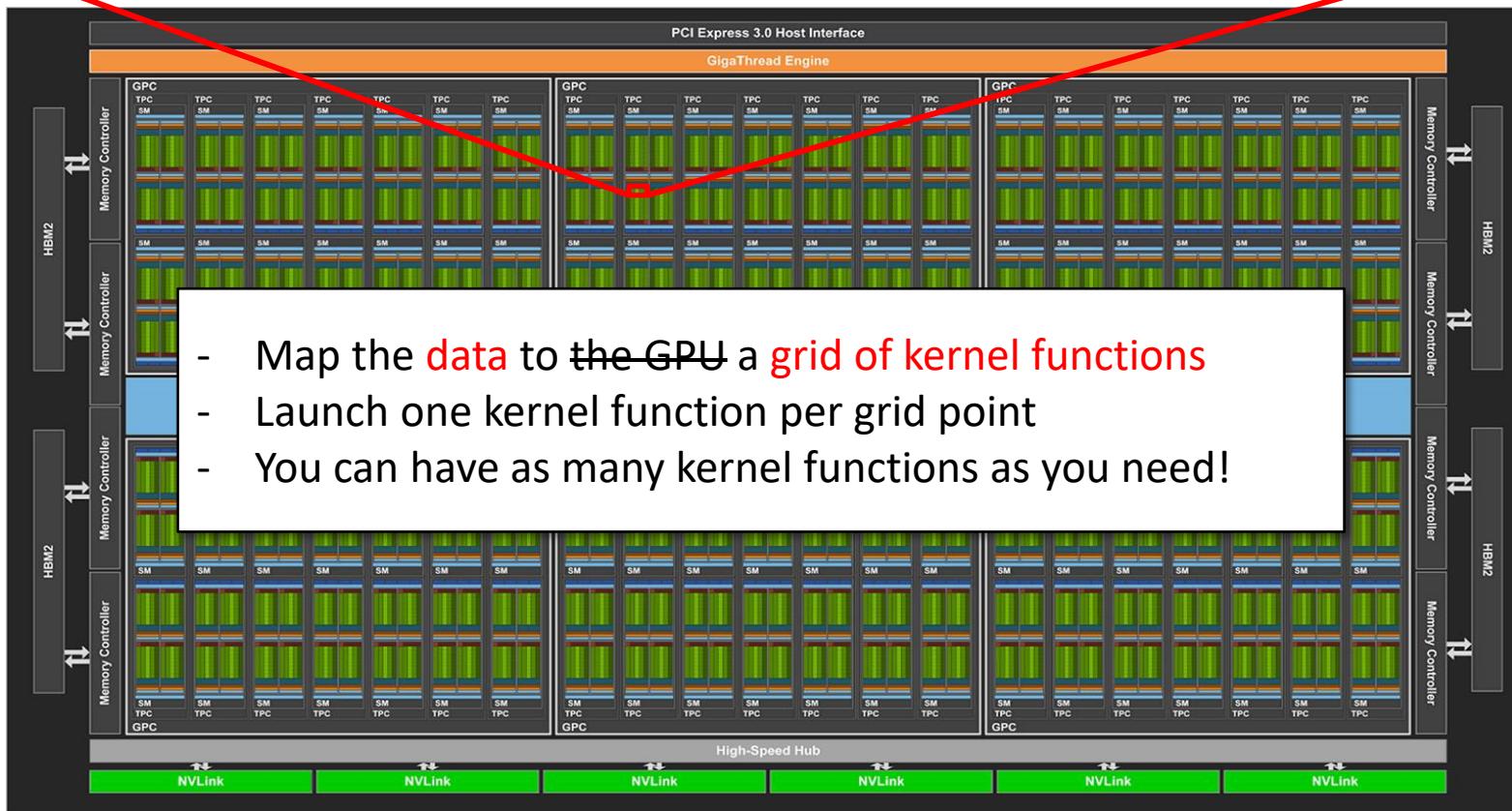
CUDA Kernel



## 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



## 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

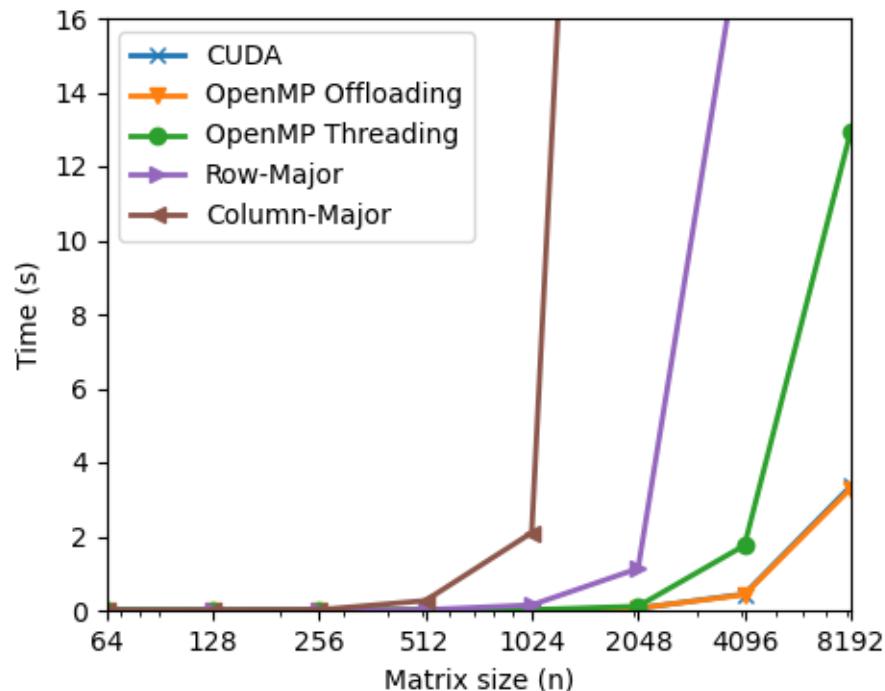
```
threadsperblock = (16,16)
blockspergrid_x = int(np.ceil(size_array / threadsperblock[0]))
blockspergrid_y = int(np.ceil(size_array / threadsperblock[1]))
blockspergrid = (blockspergrid_x, blockspergrid_y)

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

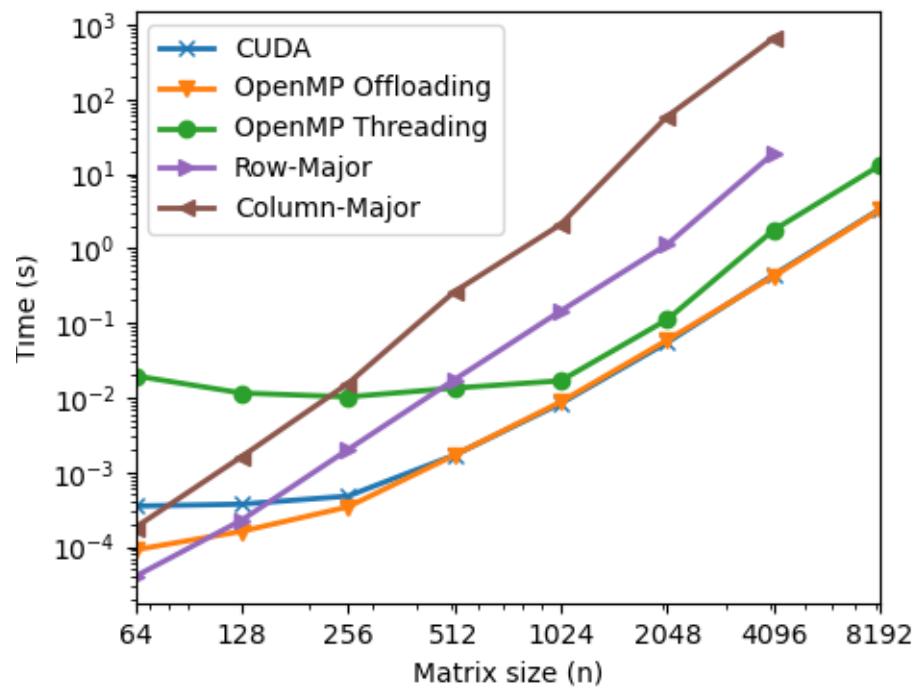
Kernel Call

## 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!

# PART IV



**Open Multi-Processing**



**Message Passing Interface**

There are two main ways of parallelizing a code



Open Multi-Processing

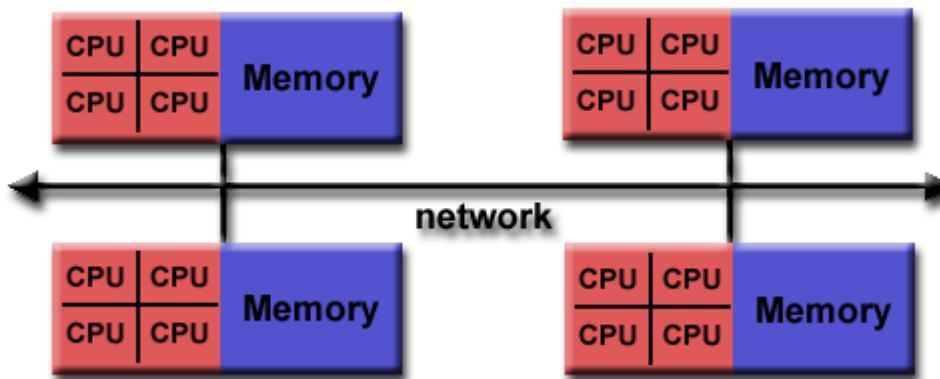


Message Passing Interface

Both are APIs for C/C++ and Fortran

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

Works in **shared and distributed** memory systems  
Can use as many cores as there are in the **cluster**  
Uses **processes** to split the work  
Processes have only **private** variables  
Loads a library and uses **specific commands**



There are two main ways of parallelizing a code



Open Multi-Processing

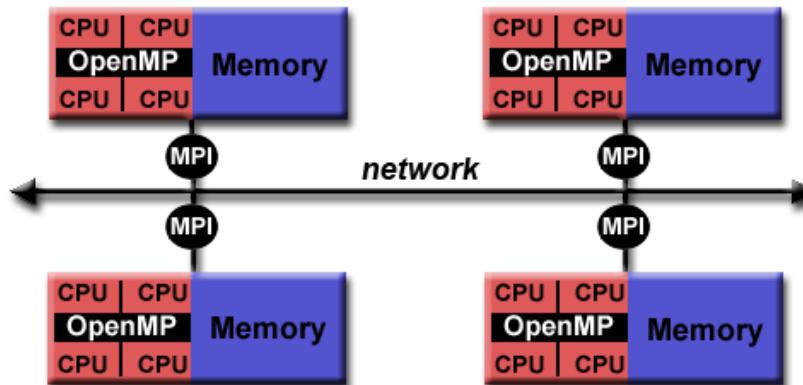


Message Passing Interface

Both are APIs for C/C++ and Fortran

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

Works in **shared and distributed** memory systems  
Can use as many cores as there are in the **cluster**  
Uses **processes** to split the work  
Processes have only **private** variables  
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>

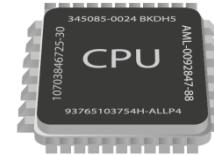
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,id,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)

do i=1,N
    <calculations>
enddo

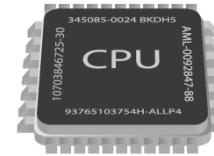
call MPI_FINALIZE(ierr)
```

N iterations

N/4



N/4



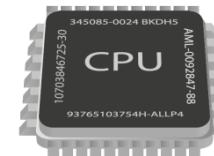
N/4



N/4



...



## What about MPI?

Loads a library and uses **specific commands**

```
include 'mpif.h'

<variable declarations>

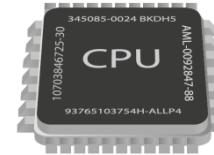
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,id,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)

do i=ini,fin
  <calculations>
enddo

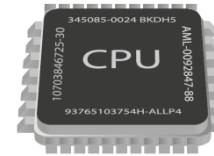
call MPI_FINALIZE(ierr)
```

N iterations

$N/nproc$  →



$N/nproc$  →



$N/nproc$  →



$N/nproc$  →



...

$N/nproc$  →



## 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)

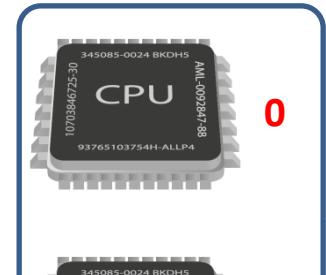
ini=id*(N/nproc)+1
fin=(id+1)*(N/nproc)

do i=ini,fin
    <calculations>
enddo

call MPI_FINALIZE(ierr)
```

N iterations

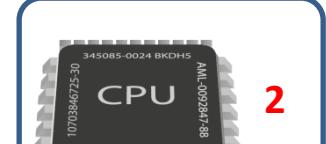
N/nproc →



N/nproc →



N/nproc →



N/nproc →

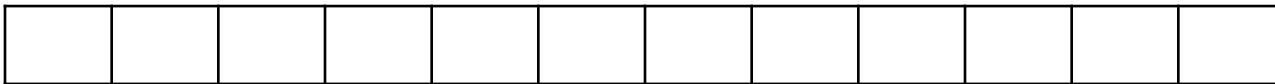
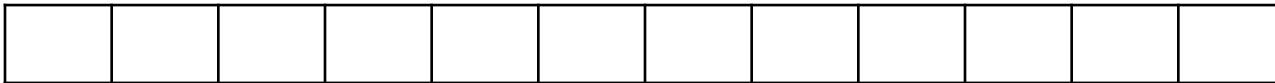


...

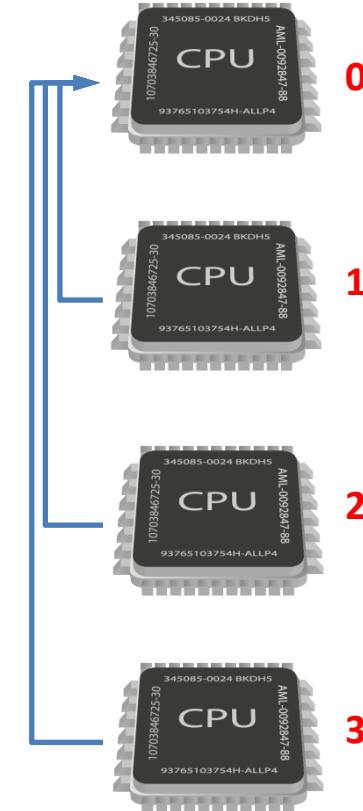
N/nproc →



nproc · 1

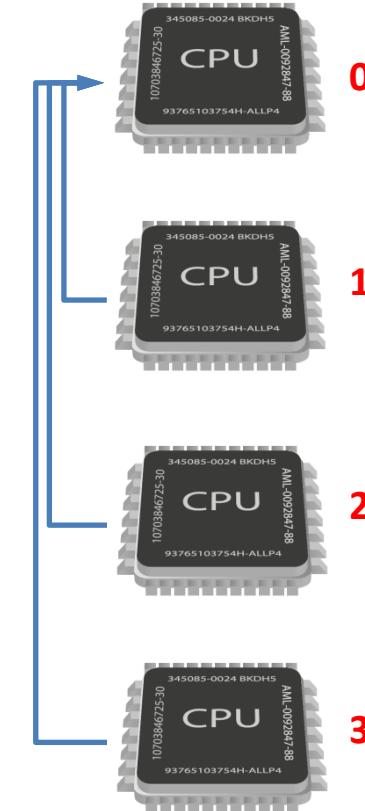
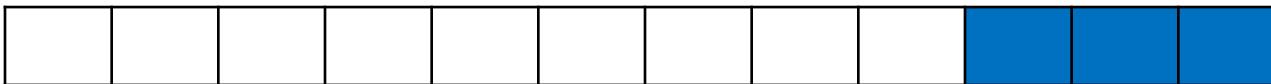


OPEN MPI



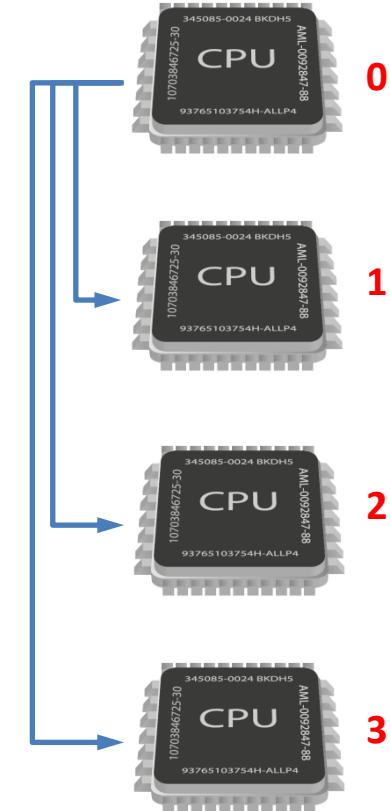
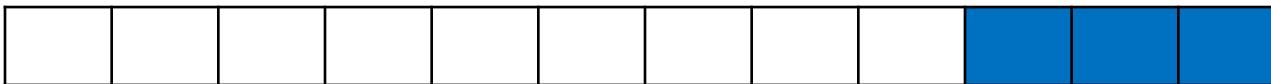
OPEN MPI

Only process 0 has the correct values!



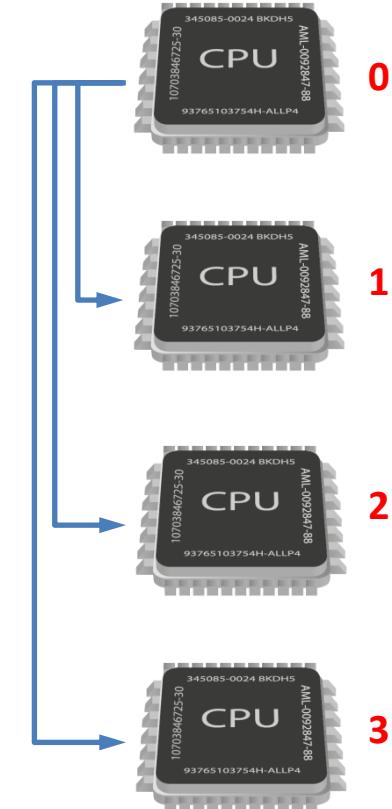
```
call MPI_REDUCE(arr,result,arraysize,<type>,<operation>,<dest_id>,MPI_WORLD_COMM,ierr)
call MPI_REDUCE(arr,result,12,MPI_INT,MPI_SUM,0,MPI_WORLD_COMM,ierr)
```

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)
```

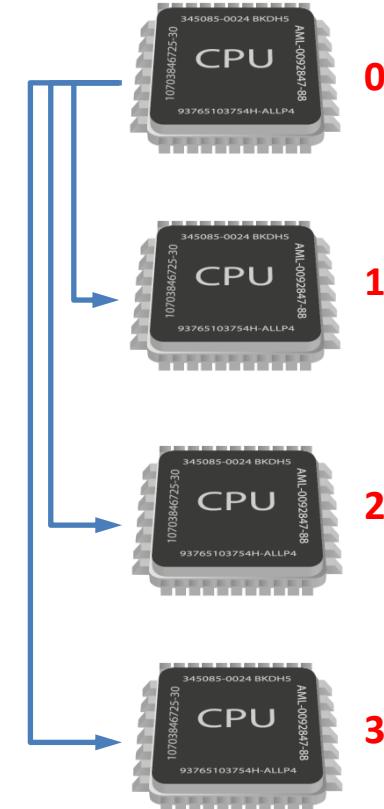
Only process 0 has the correct values!



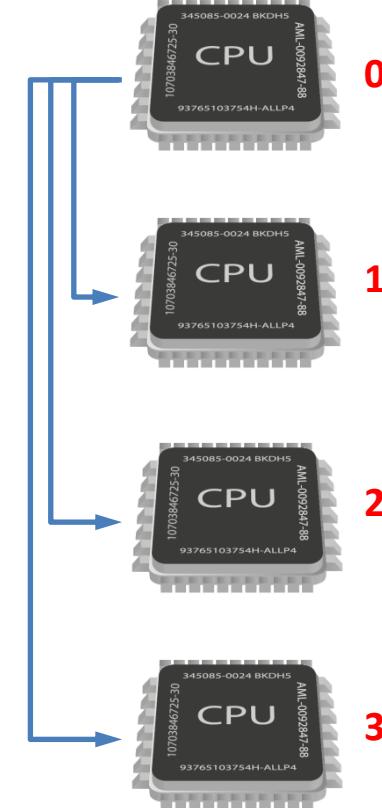
```
call MPI_REDUCE(arr,result,arraysize,<type>,<operation>,<dest_id>,MPI_WORLD_COMM,ierr)
call MPI_REDUCE(arr,result,12,MPI_INT,MPI_SUM,0,MPI_WORLD_COMM,ierr)
call MPI_BCAST(result,12,MPI_INT,0,MPI_WORLD,COMM,ierr)
```



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



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)

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**



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)

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

**FORTRAN**

Do the computation

Sum the data across nodes

## 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

```
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

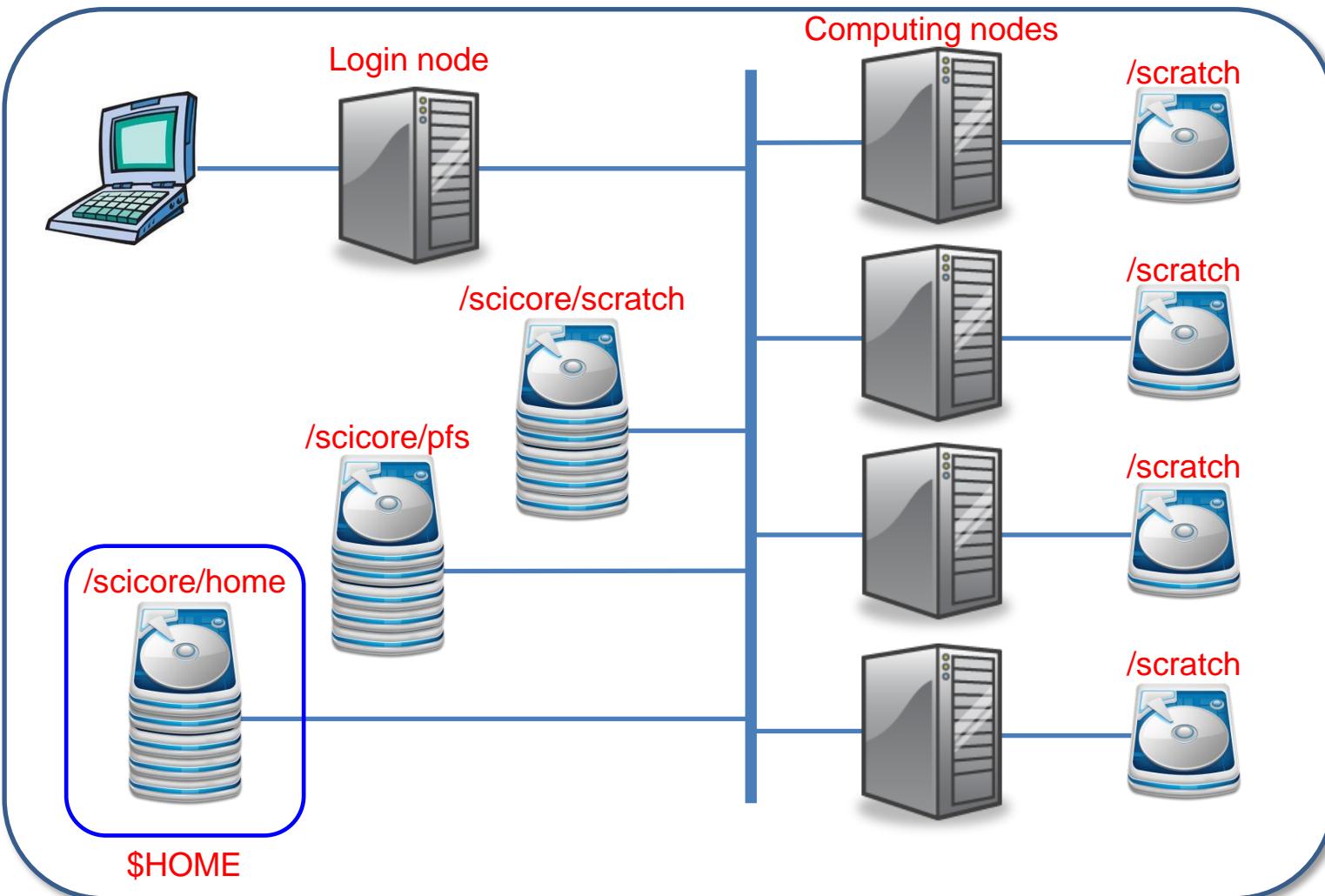
Python

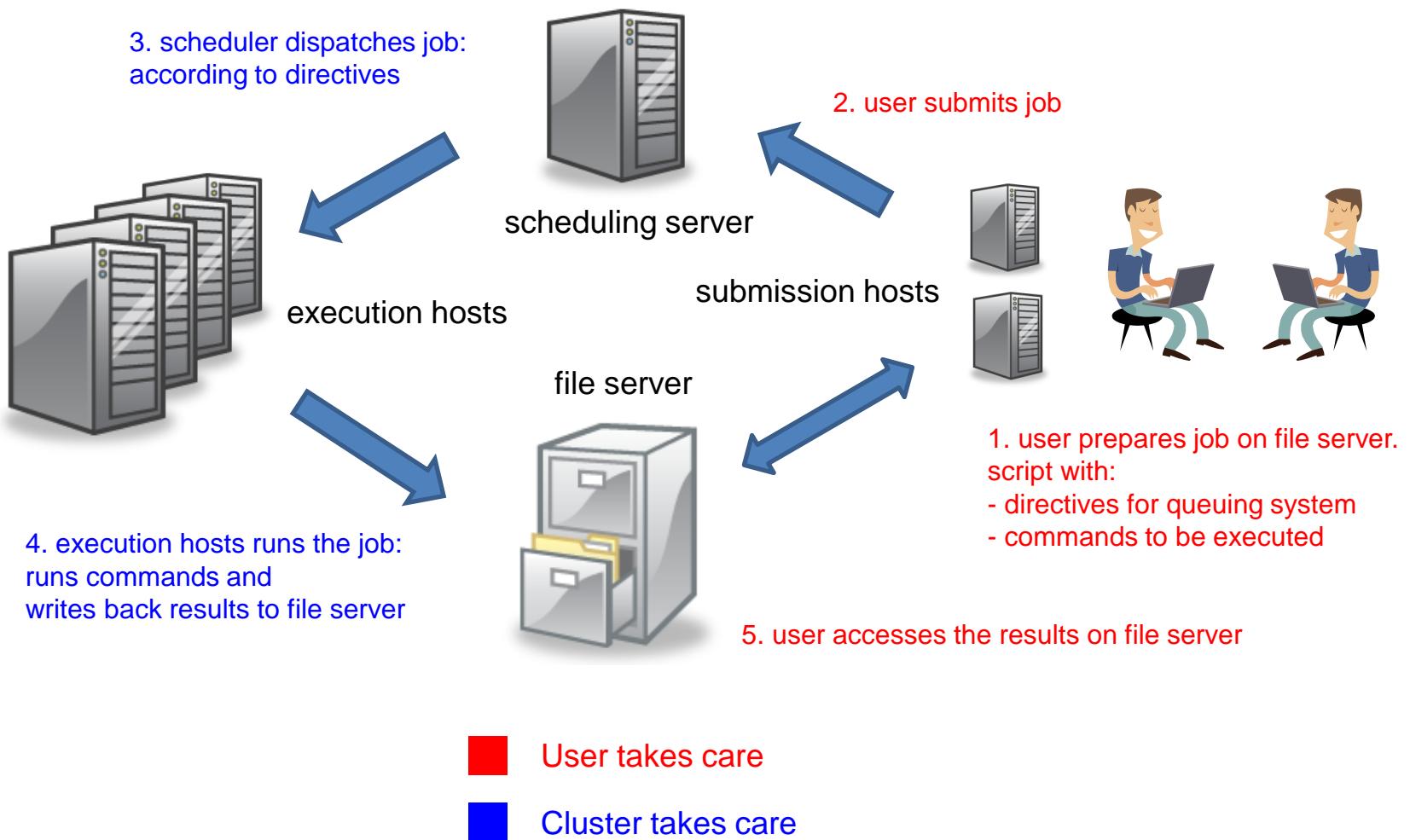
Do the computation

Sum the data across nodes



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





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
```

```

#!/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

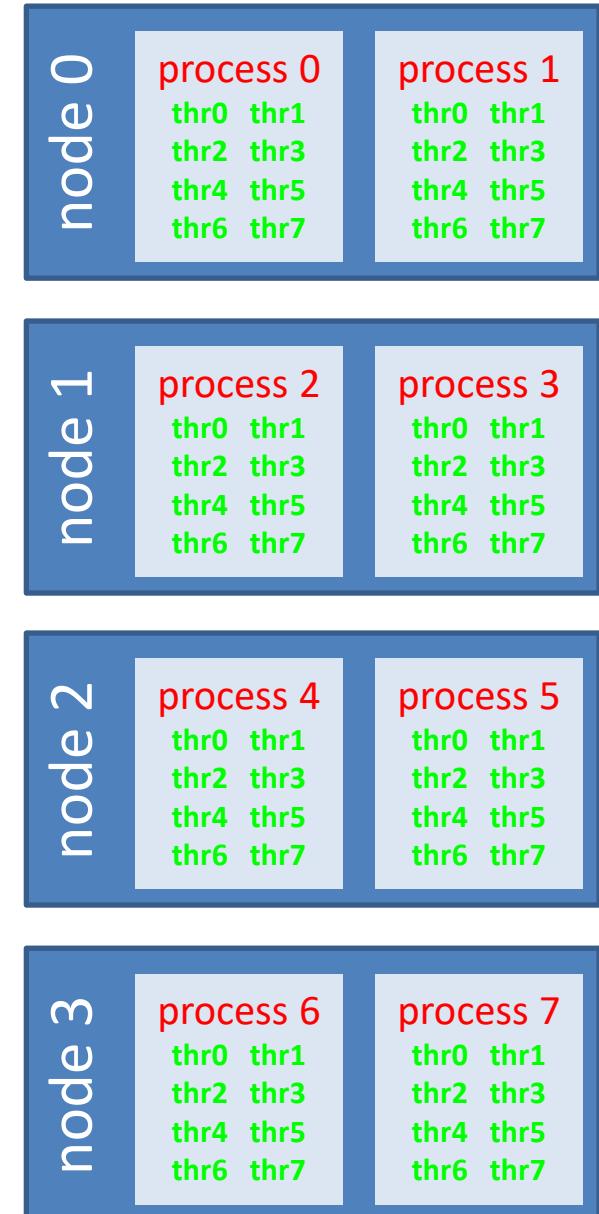
# load your required modules
#####
module load <module name>

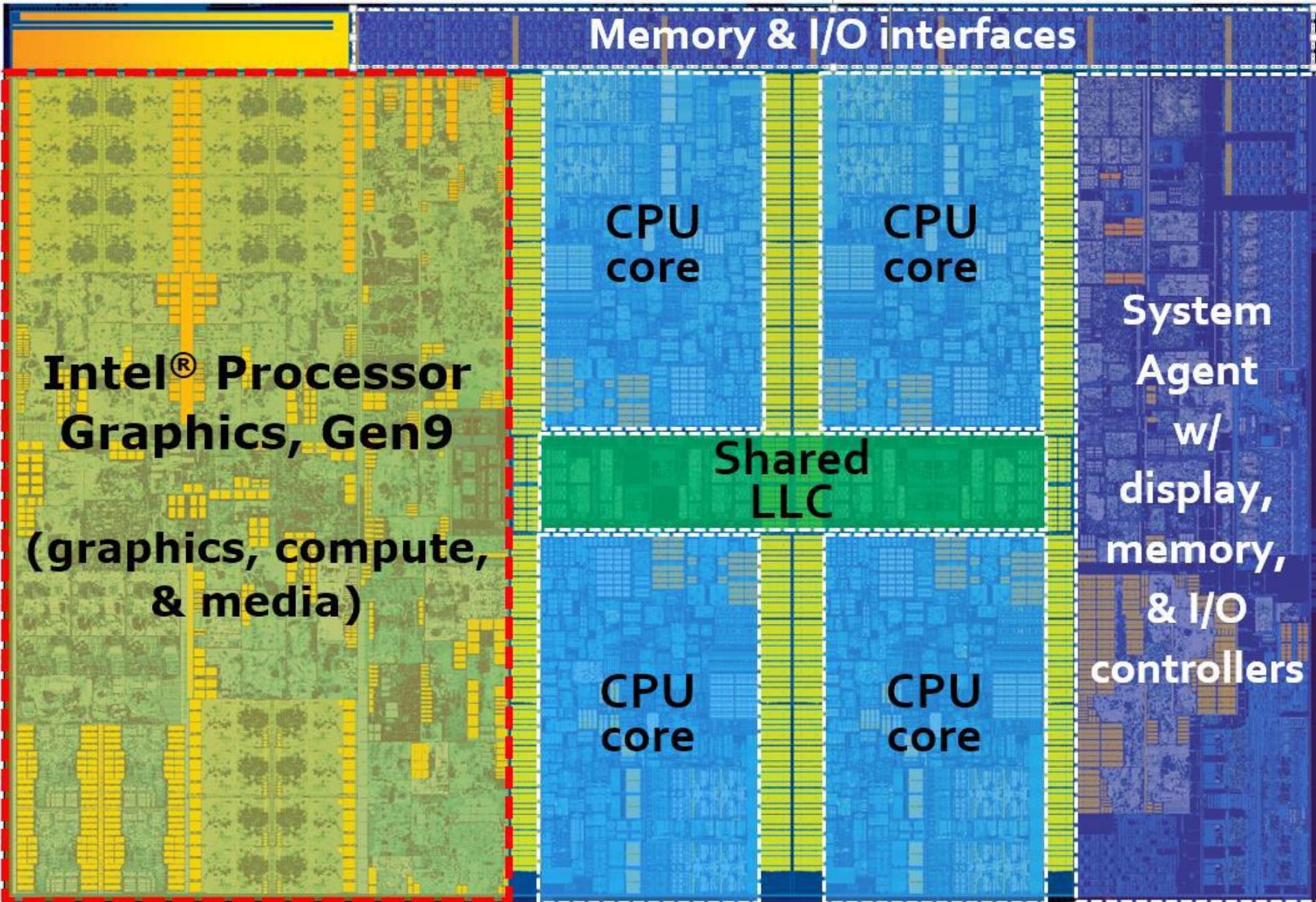
...
# and here goes your command line
srun my_openmp_program

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

```

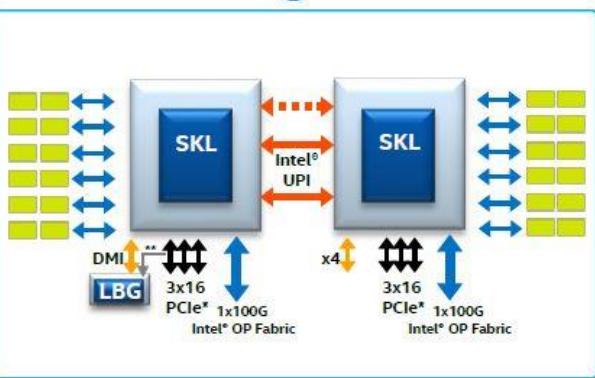
## How do I launch an hybrid job?





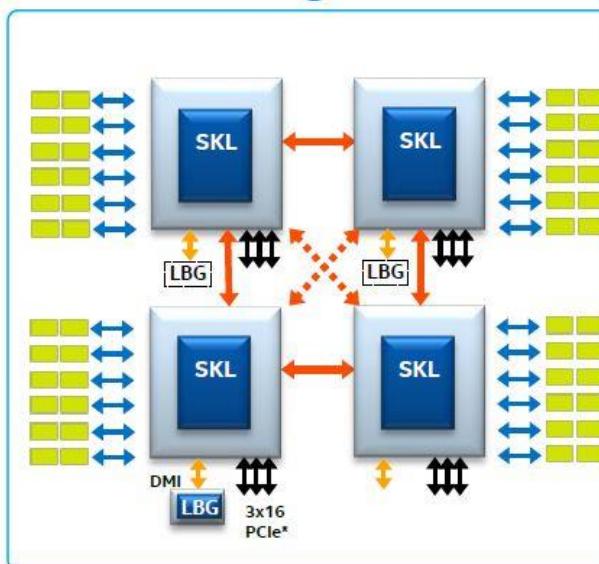
# Platform Topologies

2S Configurations



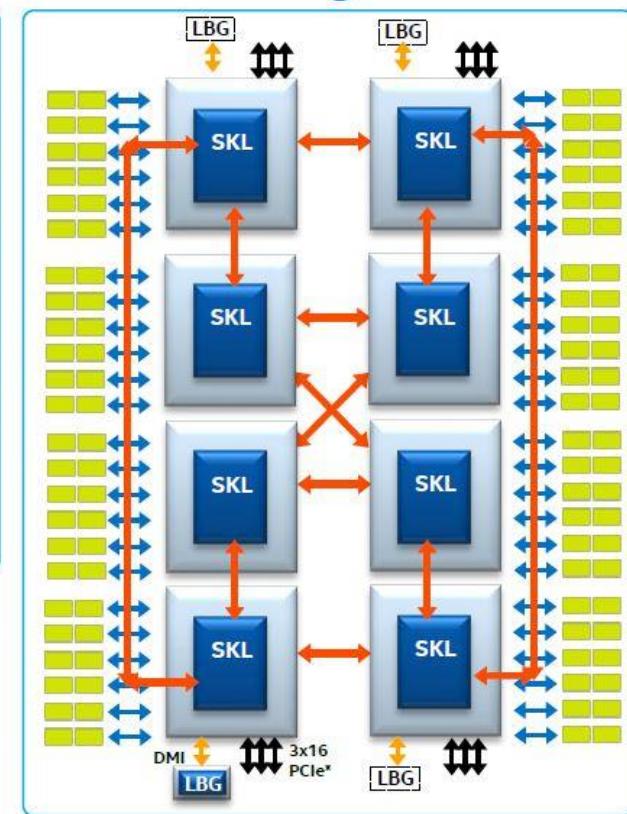
(2S-2UPI & 2S-3UPI shown)

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

```
[cabezón@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 ← Relative distances
  1: 11 10
```

## lscpu

```
[cabezón@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
```

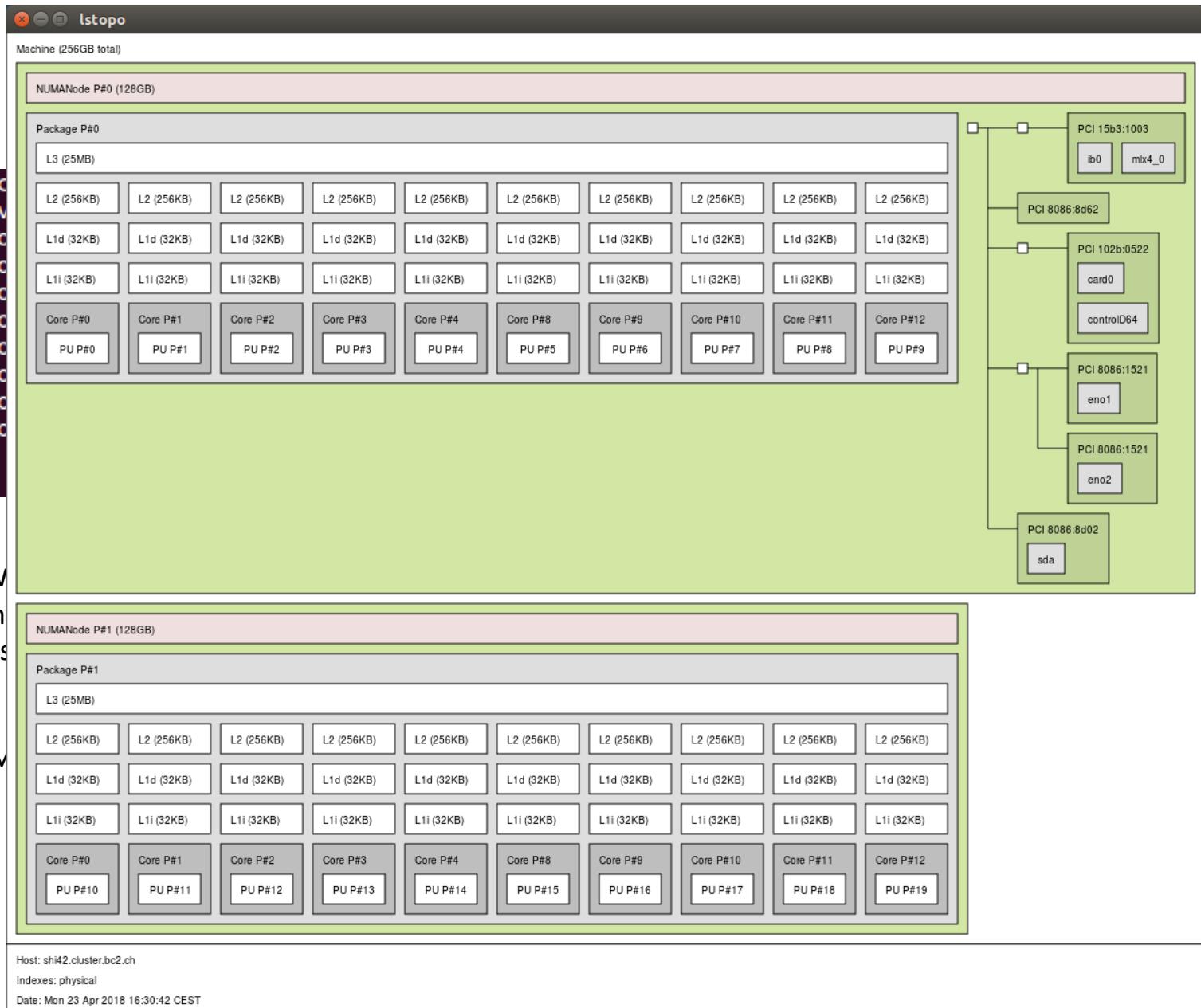
Two sockets!

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

vs.

SMP (symmetric multi-processor):

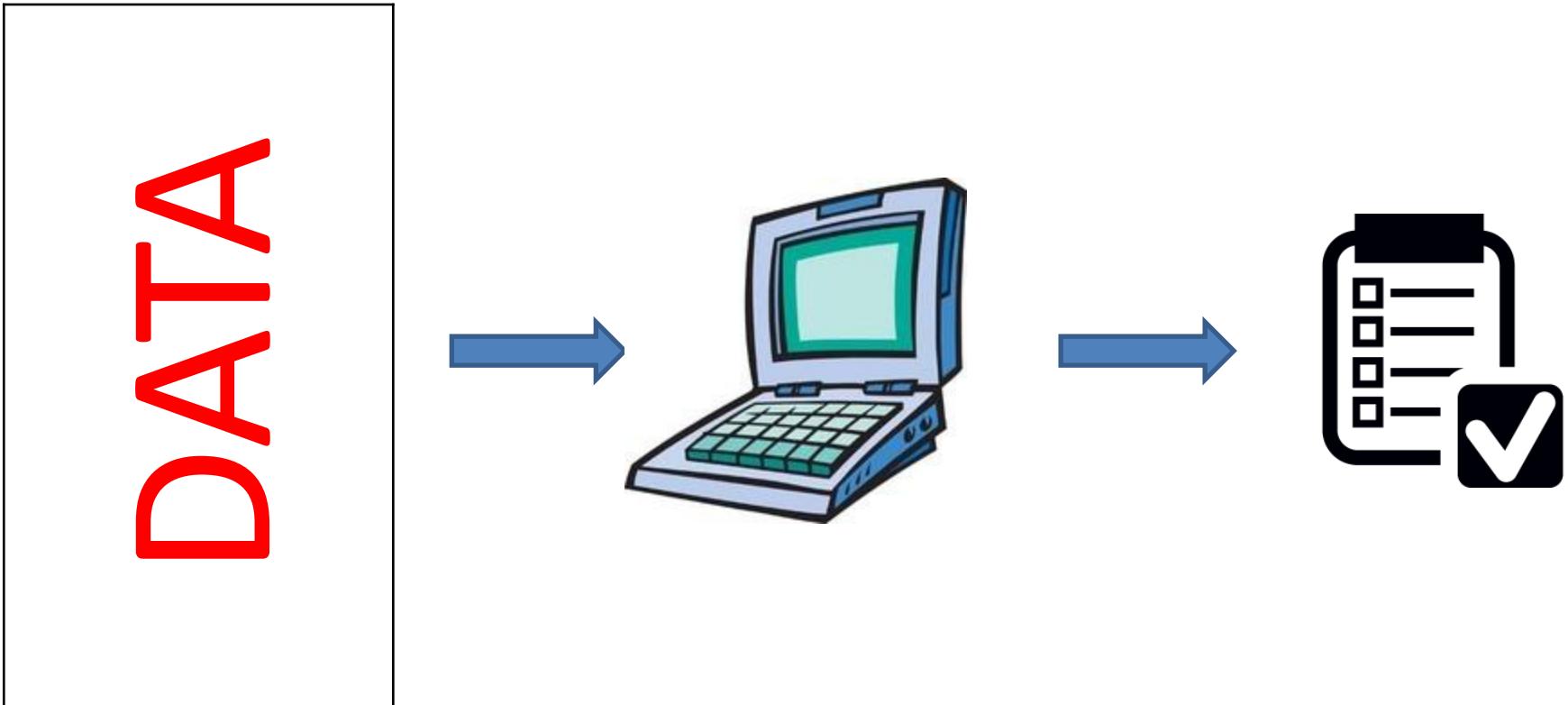
```
ruben@jarvis:~$ numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7
node 0 size: 7885 MB
node 0 free: 252 MB
node distances:
node    0
  0: 10
```



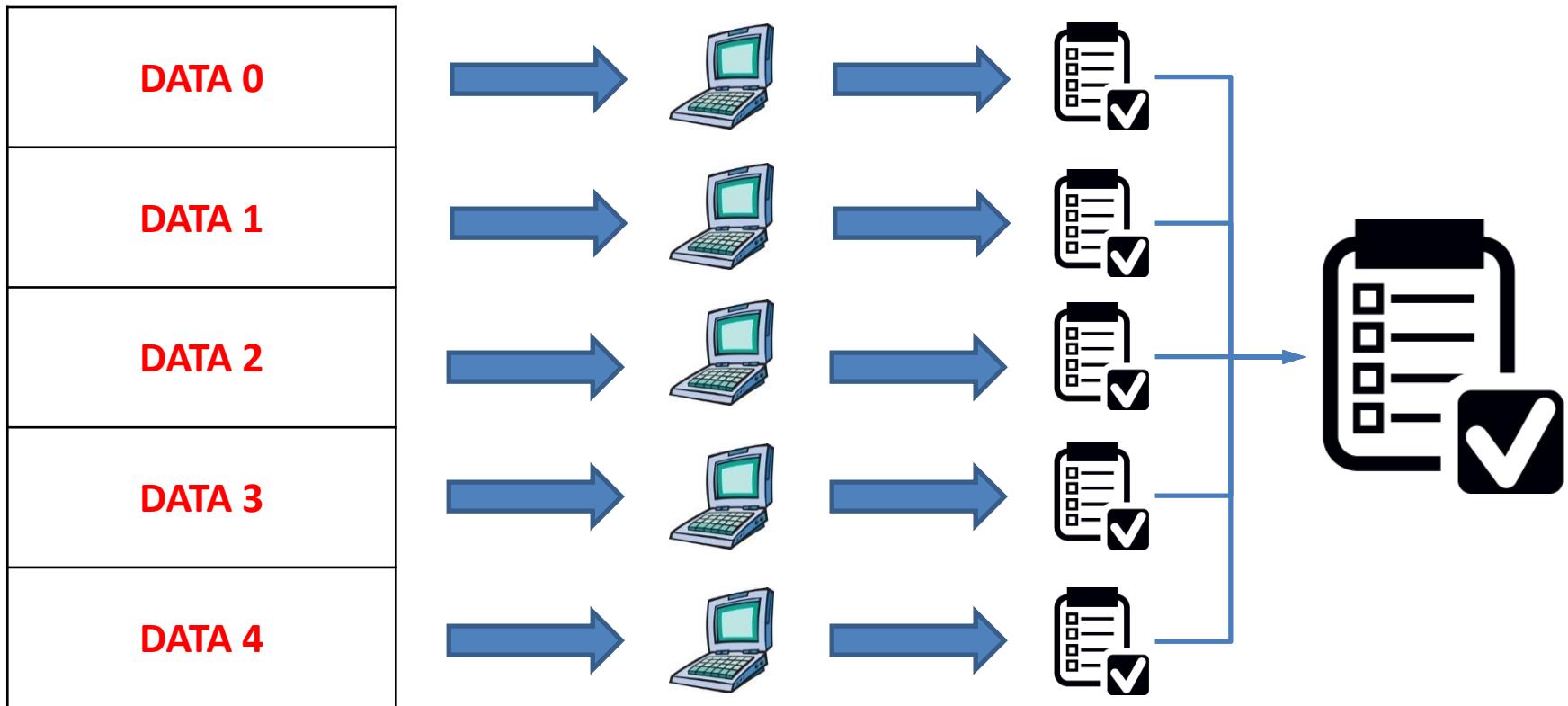
Istopo  
(from hwloc package)

E5-2670 0 @ 2.60GHz

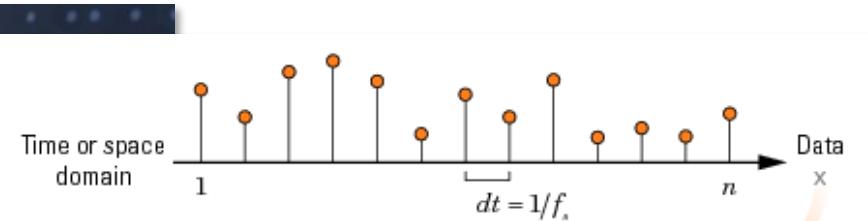
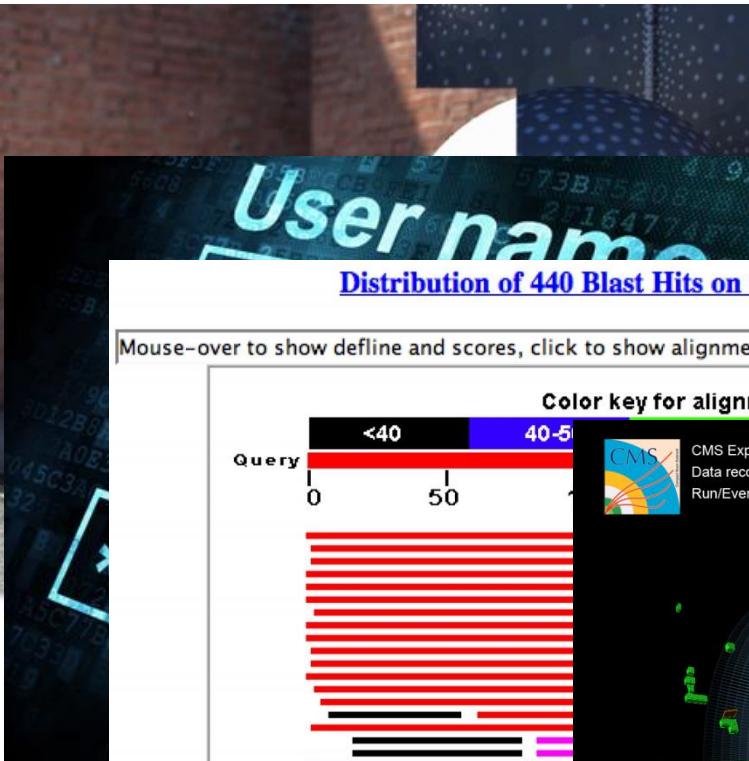
## Array jobs (embarrassingly parallel calculations)



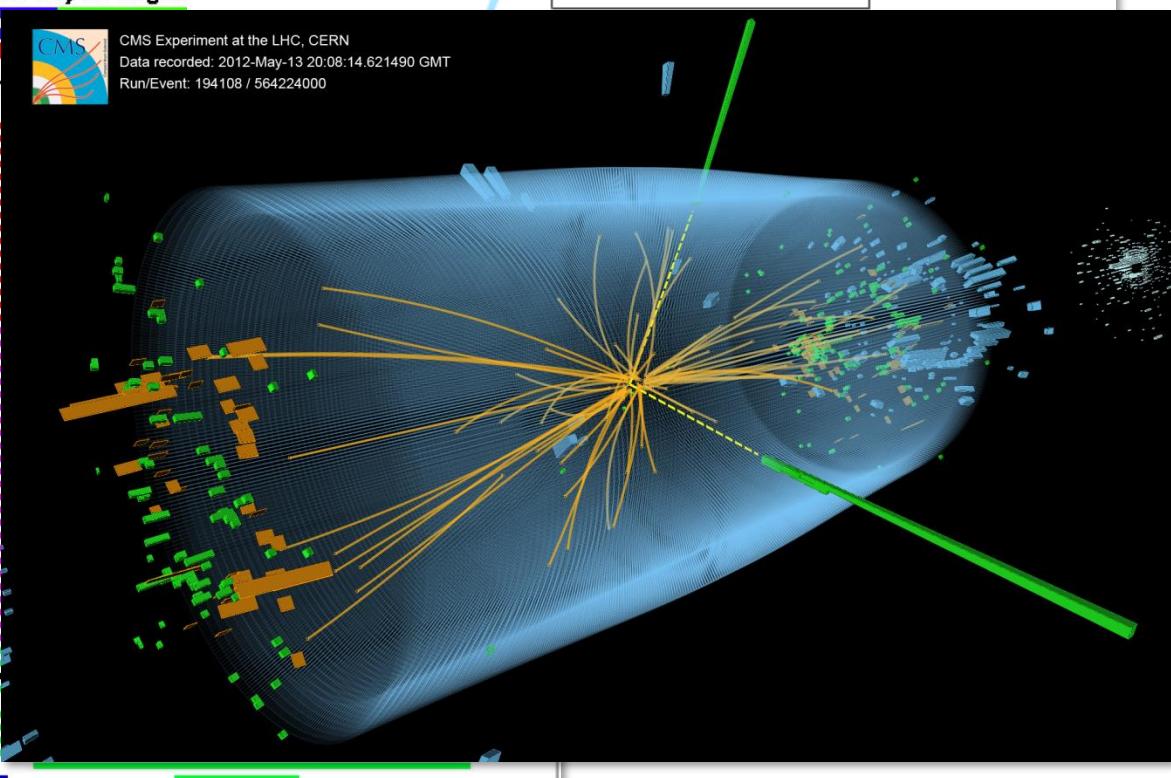
## Array jobs (embarrassingly parallel calculations)



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



$$\mathbf{y}_{p+1} = \sum_{j=0}^{n-1} \omega^{jp} \mathbf{x}_{j+1}$$



# Array jobs (embarrassingly 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 an array job, and realize that you made a mistake, you only have one job 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 (embarrassingly 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 an array job, and realize that you made a mistake, you only have one job 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.

commands.cmd

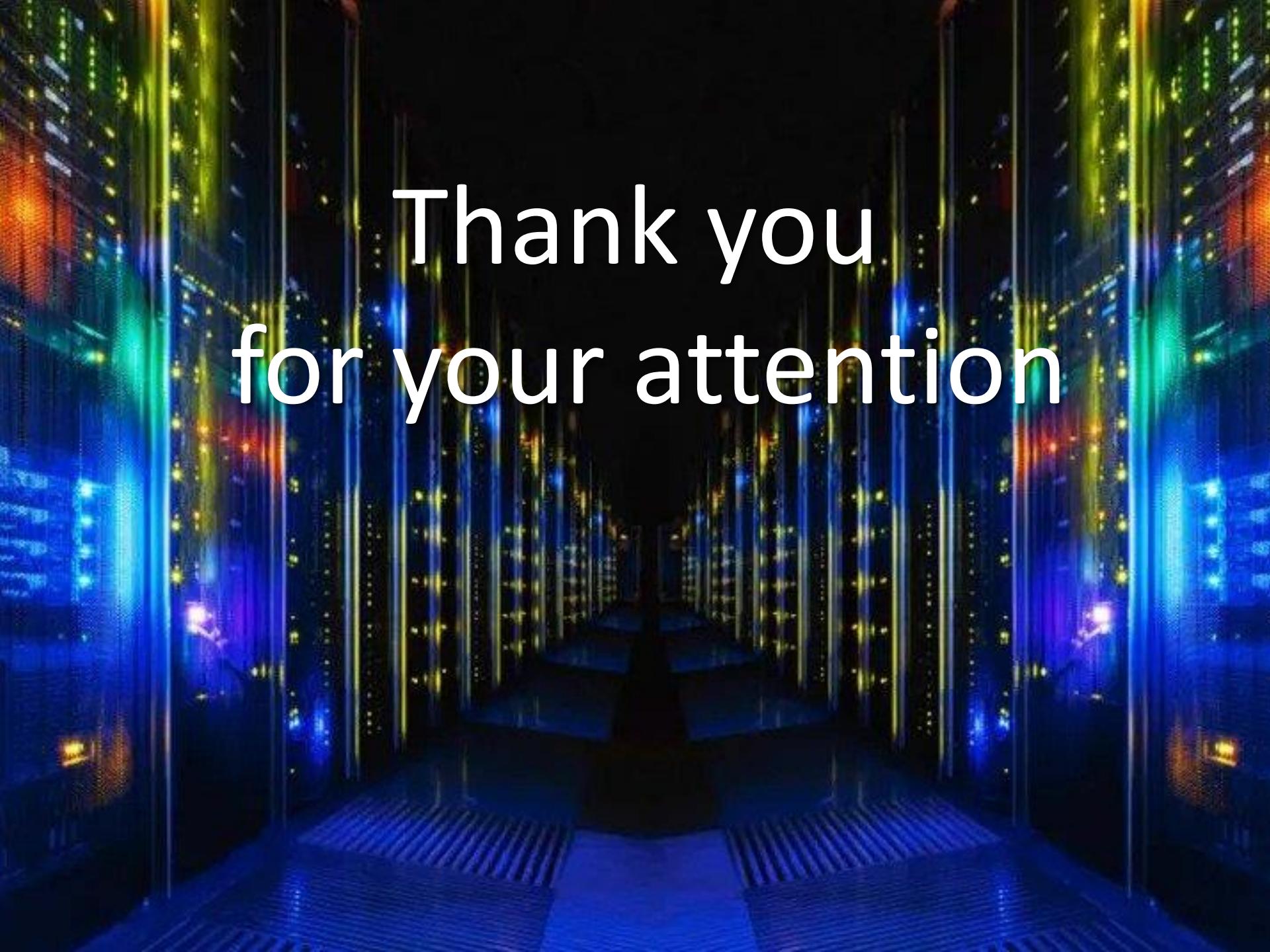
command1

command2

command3

...

commandN



Thank you  
for your attention