Week 7: Parallelisation

Shared memory versus memory passing, race conditions and load balancing, jargon buster!

Dr K Clough, Topics in Scientific computing, Autumn term 2023

Plan for today

1. What is a supercomputer? (very basic definition!)
2. What is parallelisation? Human computer demo!
3. Paradigm 1: Shared memory parallelisation, “threads”, e.g. OpenMP
4. Paradigm 2: Message passing, “ranks/processes”, e.g. MPI, MPI4py
5. What problems can be parallelised?

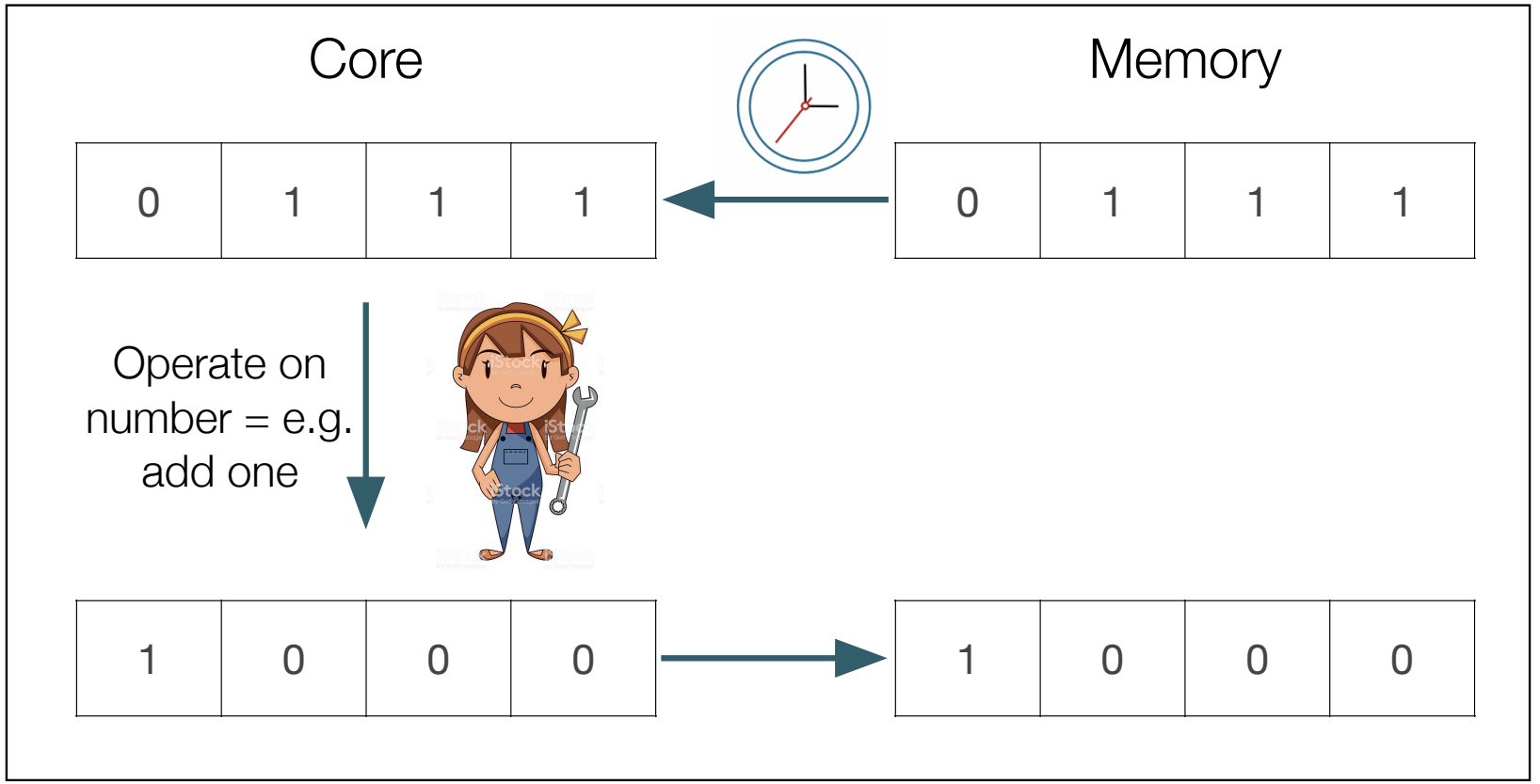
Do not let people confuse you with their fancy computing words, parallel computing is (at least in principle) not complicated

Reasonable questions to ask

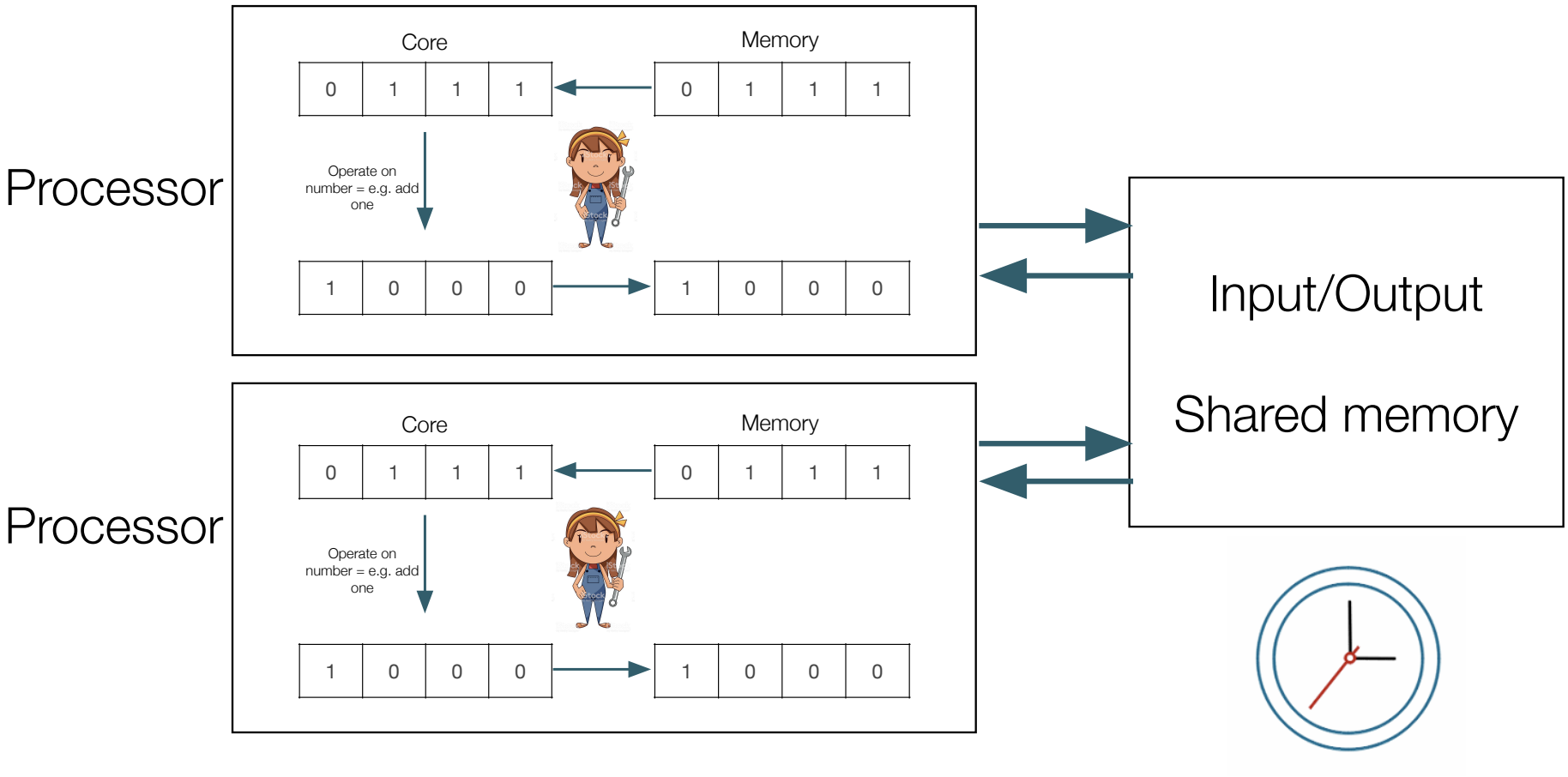
1. Can you draw out approximately the hardware design of the system, in terms of they layout of cores and how they are connected?
2. How many cores does this system have?
3. What is the memory? Is that per core, per processor or per node?
4. What parallelisation paradigm to you use? Why?
5. What is the main bottleneck in the calculation? Why?
6. Can you explain a bit more what you mean?

First, what is a computer?

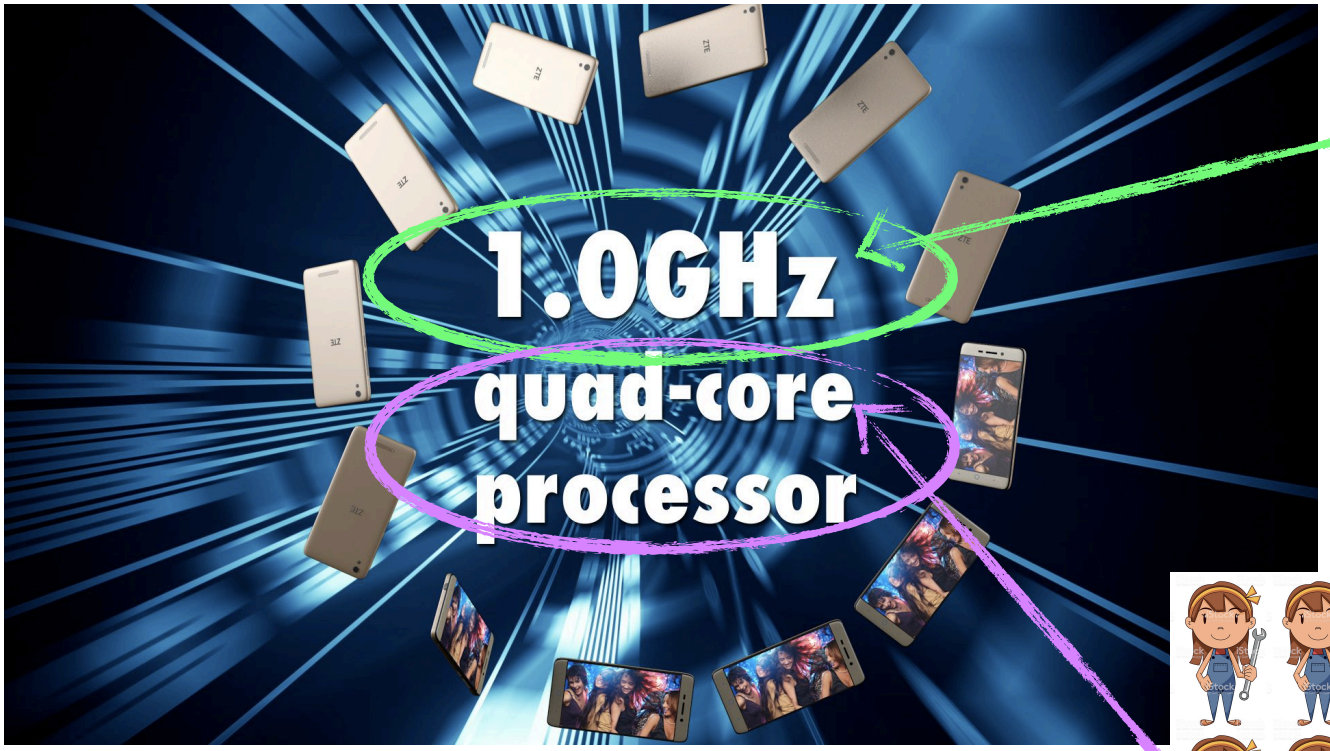
Computers do operations on numbers



Computers with multiple cores can do operations on multiple numbers at the same time



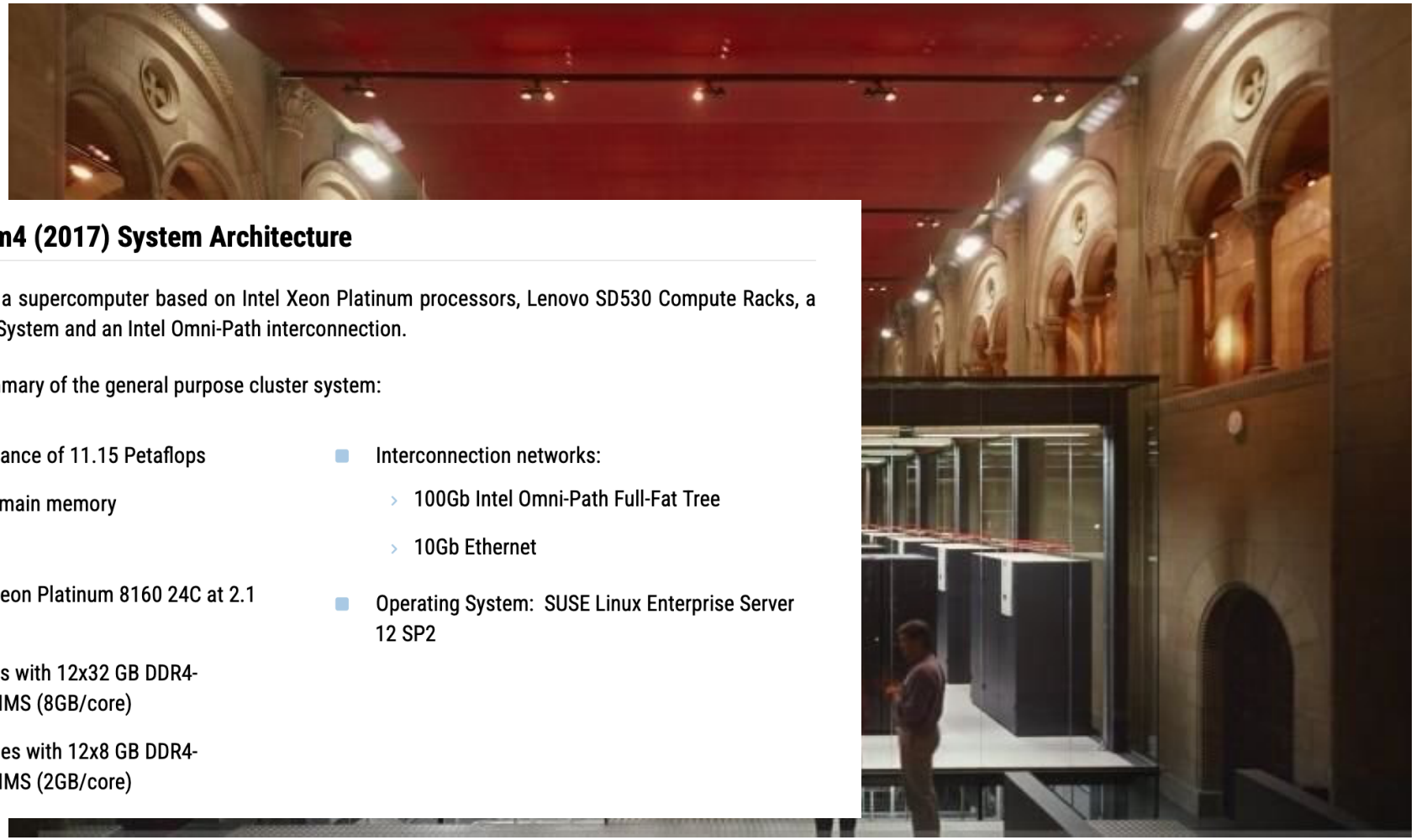
The computers you use every day!



So what is a supercomputer?

One of the supercomputers I use - MareNostrum in Barcelona

One of the supercomputers I use - MareNostrum in Barcelona



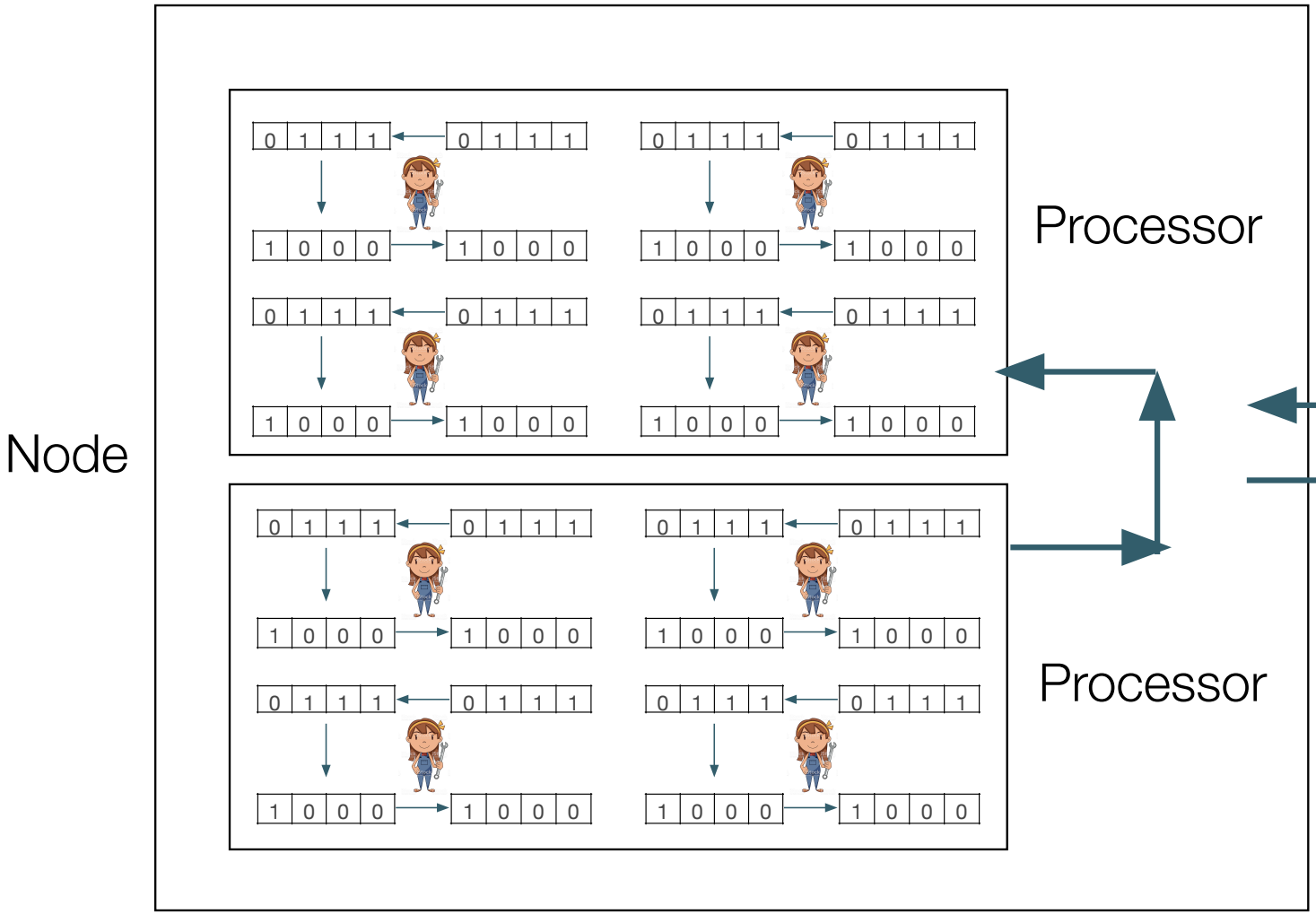
MareNostrum4 (2017) System Architecture

MareNostrum is a supercomputer based on Intel Xeon Platinum processors, Lenovo SD530 Compute Racks, a Linux Operating System and an Intel Omni-Path interconnection.

See below a summary of the general purpose cluster system:

- Peak Performance of 11.15 Petaflops
- 384.75 TB of main memory
- 3,456 nodes:
  - 2x Intel Xeon Platinum 8160 24C at 2.1 GHz
  - 216 nodes with 12x32 GB DDR4-2667 DIMMS (8GB/core)
  - 3240 nodes with 12x8 GB DDR4-2667 DIMMS (2GB/core)
- Interconnection networks:
  - 100Gb Intel Omni-Path Full-Fat Tree
  - 10Gb Ethernet
- Operating System: SUSE Linux Enterprise Server 12 SP2

Many complicated sounding words for simple things

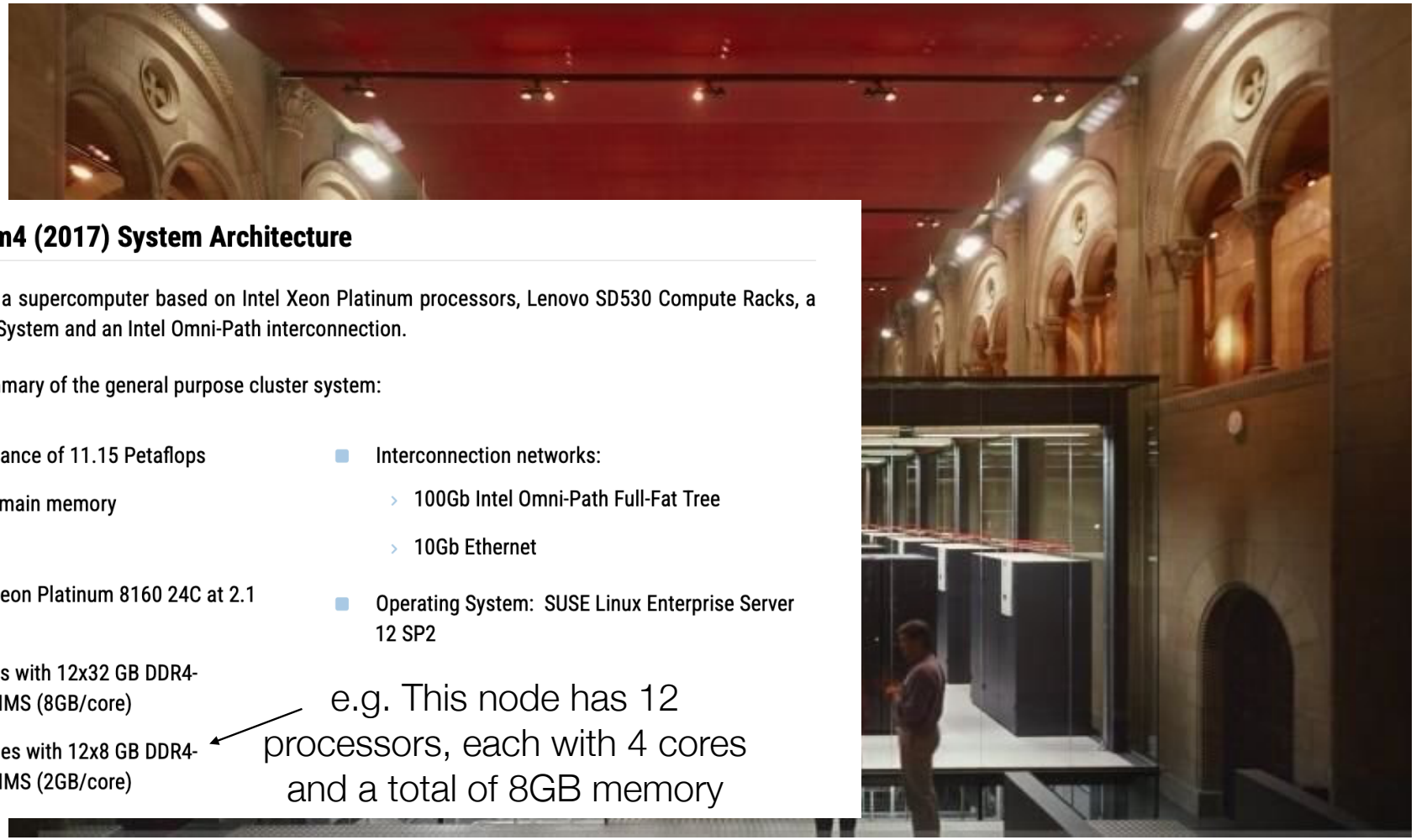


A **node** is a collection of **processors,** which contain one or more **cores.**

GPUs (graphics processing units) have many cores per processor, whereas CPUs (central processing units) typically have 1-8).

A supercomputer has many nodes that communicate with each other with **low latency** (ie, fast).

One of the supercomputers I use - MareNostrum in Barcelona



MareNostrum4 (2017) System Architecture

MareNostrum is a supercomputer based on Intel Xeon Platinum processors, Lenovo SD530 Compute Racks, a Linux Operating System and an Intel Omni-Path interconnection.

See below a summary of the general purpose cluster system:

- Peak Performance of 11.15 Petaflops
- 384.75 TB of main memory
- 3,456 nodes:
  - 2x Intel Xeon Platinum 8160 24C at 2.1 GHz
  - 216 nodes with 12x32 GB DDR4-2667 DIMMS (8GB/core)
  - 3240 nodes with 12x8 GB DDR4-2667 DIMMS (2GB/core)
- Interconnection networks:
  - 100Gb Intel Omni-Path Full-Fat Tree
  - 10Gb Ethernet
- Operating System: SUSE Linux Enterprise Server 12 SP2

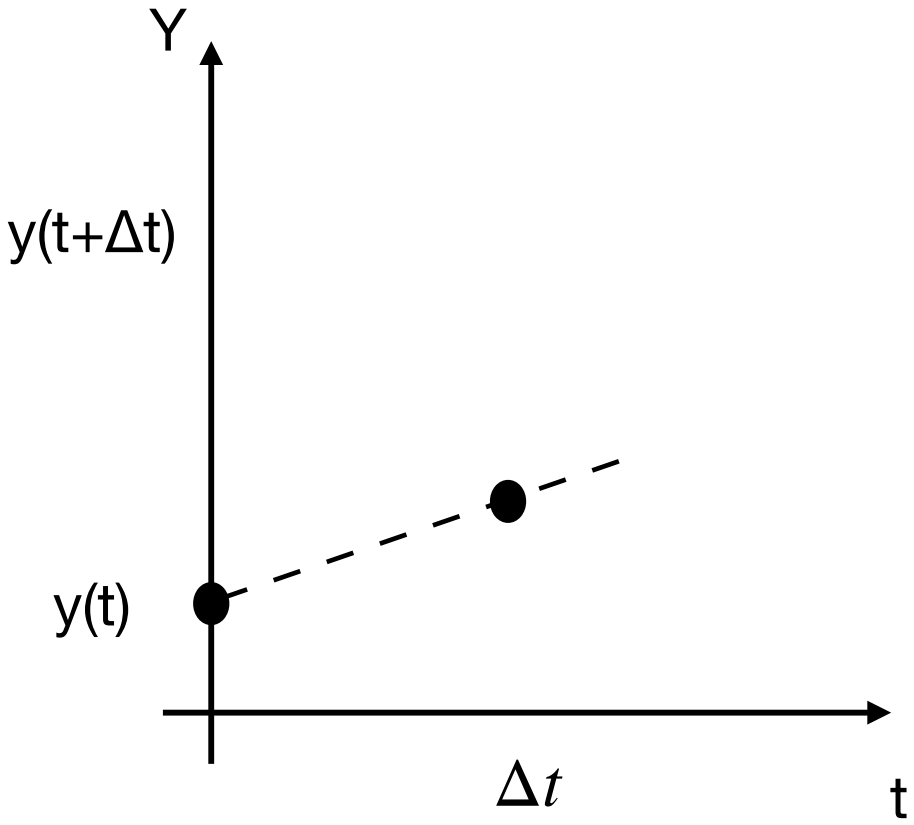
e.g. This node has 12 processors, each with 4 cores and a total of 8GB memory

Parallel demo: Euler again!

Consider a first order multi dimensional system

$$\frac{dy_i}{dt} = y_{i+1} - y_{i-1}$$

$$y_i(t = 0) = f(i)$$



Parallel Euler

$$\frac{dy_i}{dt} = y_{i+1} - y_{i-1}$$

$$y_i(t = 0) = f(i)$$

	y_1	y_2	y_3	y_4	y_5	y_6
t=1	**0**	**1**	**2**	**3**	**1**	**0**
t=2	0					0
t=3	0					0
t=4	0					0

Parallel Euler

$$\frac{dy_i}{dt} = y_{i+1} - y_{i-1}$$

$$y_i(t = 0) = f(i)$$

	y_1	y_2	y_3	y_4	y_5	y_6
t=1	**0**	**1**	**2**	**3**	**1**	**0**
t=2	0	2	2	-1	-3	0
t=3	0					0
t=4	0					0

Parallel Euler

$$\frac{dy_i}{dt} = y_{i+1} - y_{i-1}$$

$$y_i(t = 0) = f(i)$$

	y_1	y_2	y_3	y_4	y_5	y_6
t=1	**0**	**1**	**2**	**3**	**1**	**0**
t=2	0	2	2	-1	-3	0
t=3	0	2	-3	-5	1	0
t=4	0					0

Parallel Euler

$$\frac{dy_i}{dt} = y_{i+1} - y_{i-1}$$

Each variable is assigned one core.
Volunteers to be the cores!

$$y_i(t = 0) = f(i)$$

	y_1	y_2	y_3	y_4	y_5	y_6
t=1	**0**	**1**	**2**	**3**	**1**	**0**
t=2	0	2	2	-1	-3	0
t=3	0	2	-3	-5	1	0
t=4	0					0

What happens if?

1. Now one core gets three more variables to work on?
2. Now each core can call up some friends to help them?
3. Now each core only has one memory space to write in?
4. Now you can’t see the other core’s memory?
5. I want now the sum of all the variables?

Consider the case where:

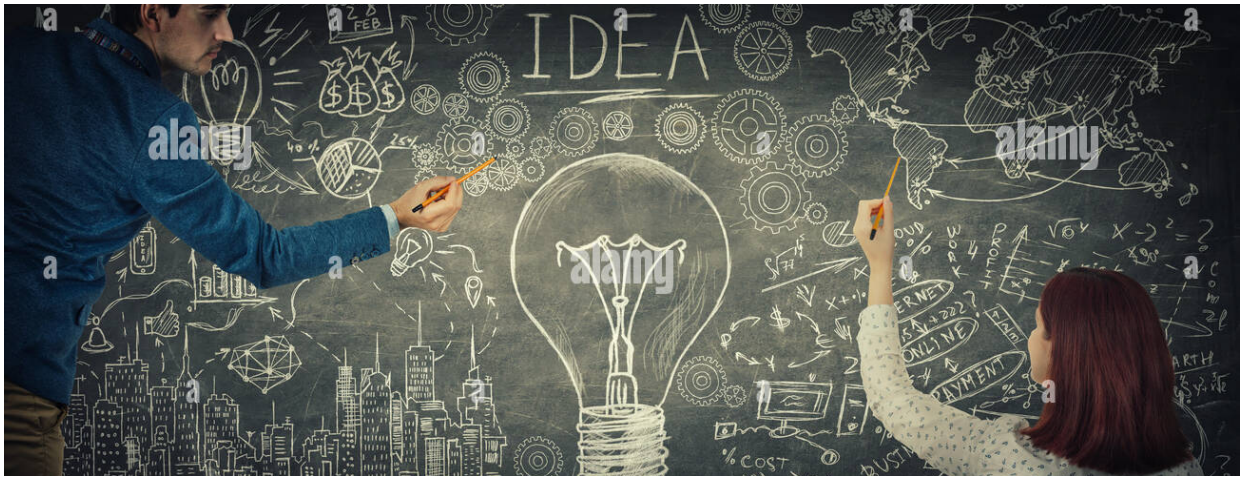
1. One core is assigned all the variables (“**primary thread**”), but can call on its friend cores (“**sub-thread**”) to come and help. It has to organise things and tell them which variables to work on, which takes a bit of time (“**overhead**”).

2. The threads can see the memory of the neighbouring thread, so they don’t need to ask for their results

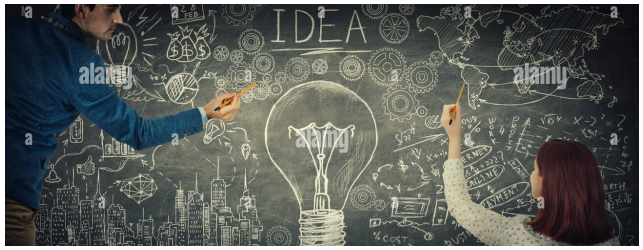
3. Each thread only has one memory space to write in, so they have to take care to avoid going out of sync
(“**race conditions**”)

4. Global operations need to be done by one thread or coordinated between threads.

   This is multithreading / shared memory parallelisation.



Shared memory parallelisation / threading / multithreading



1. Work is split amongst **sub-threads,** which are **forked** by a **primary thread.**

2. The sharing of memory is **most efficient within one processor**, and definitely can only be used **within one node**

3. Most common library is **OpenMP** (= Open Multi Processing)

4. Python Global Interpreter Lock or GIL, is a lock that allows only one thread to hold the control of the Python interpreter. This means that only one thread can be in a state of execution at any point in time, and makes implementation of multithreading challenging in python. This may change in future.

Consider the case where:

1. Each core is given a **rank** and assigned a subset of the variables - we have to make sure they are fairly shared out (“**load balancing**”)

2. These ranks are not allowed to look at each other’s memory, they have to ask for the results (“**message passing**”)

3. We might like to add in some checks to make sure the communication worked - “hey rank 2, did you get my message?" (“**send”/“receive"**)

4. Global operations need to be coordinated by one rank, usually rank 0 (“**reduce” / “gather” / “broadcast**”)

   This is message passing / MPI parallelisation.

Message passing interface (MPI)

1. Work is split amongst **processes**, which are labelled by their **rank.**

2. Each **process** may use one or more **cores** (if it has more than one core it can call these up and use multithreading- this is “hybrid parallelisation”)

3. The message passing communication can happen **between processors and between nodes**. It is extremely flexible.

4. Most compilers (e.g. gnu or Intel) contain an MPI library, that contains the necessary commands to get ranks to communicate. The python one is MPI4py. There is one called OpenMPI - don’t confuse this with OpenMP!

MPI4py - try it yourself!

```
4
5  from mpi4py import MPI
6  import sys
7
8  # note: This week I show you some optional content on parallelisation. You should first install MPI4py into your
   environment, using:
9
10 # conda activate myenv
11 # conda install mpi4py
12
13 # Then try running the following code - it must be run on the command line using (e.g. for 4 processes):
14
15 # mpirun -n 4 python Week7MPIDemo.py
16
17 # (Note that this code is run by each process, so imagine each person (process)
18 # receiving this code and executing it independently.)
19
20 # Find out who you are in the group of MPI processes
21 comm = MPI.COMM_WORLD
22 num_ranks = comm.Get_size()
23 my_rank = comm.Get_rank()
24
25 # First just print off your rank
26 print("Hello, World! I am process " + str(my_rank) + " of "
27                   + str(num_ranks) + " on " + str(comm))
28
```

What gets printed?

MPI4py - try it yourself!

```python

from mpi4py import MPI
import sys

# note: This week I show you some optional content on parallelisation. You should first install MPI4py into your environment, using:

# conda activate myenv
# conda install mpi4py

# Then try running the following code - it must be run on the command line using (e.g. for 4 processes):

# mpirun -n 4 python Week7MPIDemo.py

# (Note that this code is run by each process, so imagine each person (process)
# receiving this code and executing it independently.)

# Find out who you are in the group of MPI processes
comm = MPI.COMM_WORLD
num_ranks = comm.Get_size()
my_rank = comm.Get_rank()

# First just print off your rank
print("Hello, World! I am process " + str(my_rank) + " of "
                + str(num_ranks) + " on " + str(comm))

```

```
(myenv) katy@macadmins-MacBook-Pro Notebooks % mpirun -np 4 python Week7MPIDemo.py
Hello, World! I am process 0 of 4 on <mpi4py.MPI.Intracomm object at 0x113f785d0>
Hello, World! I am process 1 of 4 on <mpi4py.MPI.Intracomm object at 0x1132335d0>
Hello, World! I am process 2 of 4 on <mpi4py.MPI.Intracomm object at 0x1174da5d0>
Hello, World! I am process 3 of 4 on <mpi4py.MPI.Intracomm object at 0x1173265d0>
```

MPI4py - try it yourself!

```
29 # Wait until everyone gets here before going forward
30 comm.Barrier()
31
32 # This just makes sure the printing is done in the terminal before we move on
33 # Otherwise the output gets mixed up (even though the program order is right)
34 sys.stdout.flush()
35
36 # Now do some communication with the other ranks
37 message_value = my_rank
38 max_rank = num_ranks - 1
39 if my_rank == 0:
40     comm.send(message_value, dest = my_rank+1)
41     print("I am the first rank I have no message")
42
43 elif ((my_rank > 0) and (my_rank < max_rank)) :
44     message_received = comm.recv()
45     print("I am rank " + str(my_rank) + " with message "
46            + str(message_received))
47     comm.send(message_value, dest = my_rank+1)
48
49 else :
50     message_received = comm.recv()
51     print("I am the last rank " + str(my_rank) + " with message "
52            + str(message_received))
53
```

What gets printed?

MPI4py - try it yourself!

```
29 # Wait until everyone gets here before going forward
30 comm.Barrier()
31
32 # This just makes sure the printing is done in the terminal before we move on
33 # Otherwise the output gets mixed up (even though the program order is right)
34 sys.stdout.flush()
35
36 # Now do some communication with the other ranks
37 message_value = my_rank
38 max_rank = num_ranks - 1
39 if my_rank == 0:
40     comm.send(message_value, dest = my_rank+1)
41     print("I am the first rank I have no message")
42
43 elif ((my_rank > 0) and (my_rank < max_rank)) :
44     message_received = comm.recv()
45     print("I am rank " + str(my_rank) + " with message "
46            + str(message_received))
47     comm.send(message_value, dest = my_rank+1)
48
49 else :
50     message_received = comm.recv()
51     print("I am the last rank " + str(my_rank) + " with message "
52            + str(message_received))
53
```

```
I am the first rank I have no message
I am rank 1 with message 0
I am rank 2 with message 1
I am the last rank 3 with message 2
```

What problems can be parallelised?

1. Embarassingly parallel (yes people really say this and it is a thing)
2. Parallel with communication
3. Inherently serial

Can you think of examples of each of these?

What problems can be parallelised?

1. Embarassingly parallel (yes people really say this and it is a thing)

   -> Analysing multiple (uncoupled) data points using the same formula, refreshing the screen during a computer game…

2. Parallel with communication

   -> Solving coupled ODEs or PDEs where we divide the variables or positions amongst the processes, summing a large array of data…

3. Inherently serial

   -> Each operation depends on the next (e.g. constructing most Ikea furniture), many problems in cryptography…

Hardware versus parallelisation paradigm

1. Process
2. Core
3. OpenMPI
4. Rank
5. Thread
6. Processor
7. GPU
8. CPU
9. Node
10. OpenMP

Which are hardware words?

Which are words related to a parallelisation paradigm?

Which are related to message passing parallelisation?

Which are related to shared memory parallelisation?

Hardware versus parallelisation paradigm

1. Process
2. **Core**
3. OpenMPI
4. Rank
5. Thread
6. **Processor**
7. **GPU**
8. **CPU**
9. **Node**
10. OpenMP

Hardware words

Hardware versus parallelisation paradigm

1. **Process**
2. Core
3. **OpenMPI**
4. **Rank**
5. **Thread**
6. Processor
7. GPU
8. CPU
9. Node
10. **OpenMP**

Words related to a parallelisation paradigm

Hardware versus parallelisation paradigm

1. **Process**
2. Core
3. **OpenMPI**
4. **Rank**
5. Thread
6. Processor
7. GPU
8. CPU
9. Node
10. OpenMP

Related to message passing parallelisation

Hardware versus parallelisation paradigm

1. Process
2. Core
3. OpenMPI
4. Rank
5. **Thread**
6. Processor
7. GPU
8. CPU
9. Node
10. **OpenMP**

Related to shared memory parallelisation

Plan for today

1. What is a supercomputer? (very basic definition!)
2. What is parallelisation? Human computer demo!
3. Paradigm 1: Shared memory parallelisation, “threads”, e.g. OpenMP
4. Paradigm 2: Message passing, “ranks/processes”, e.g. MPI, MPI4py
5. What problems can be parallelised?