



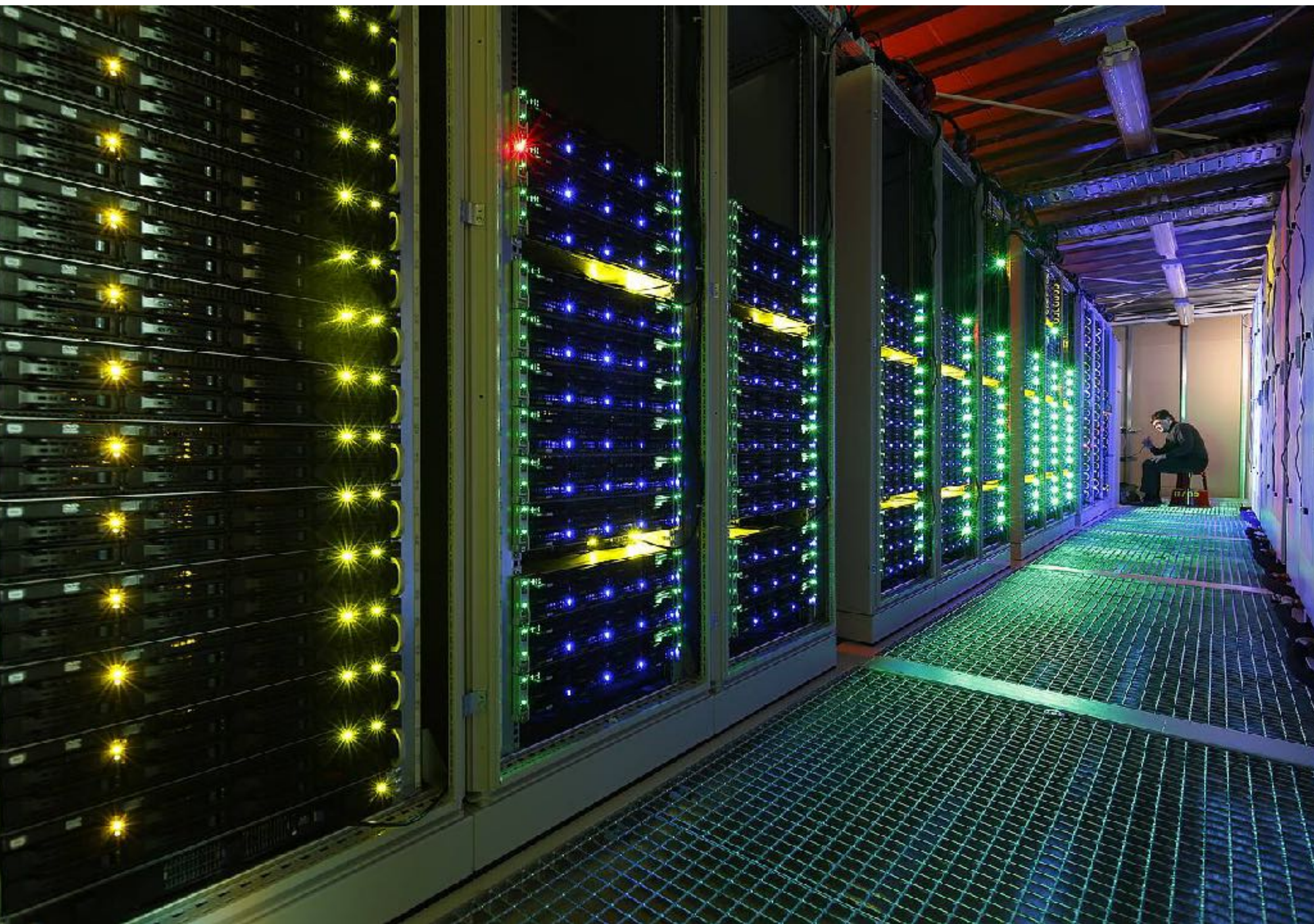
HIGH PERFORMANCE COMPUTING

Ari Maller



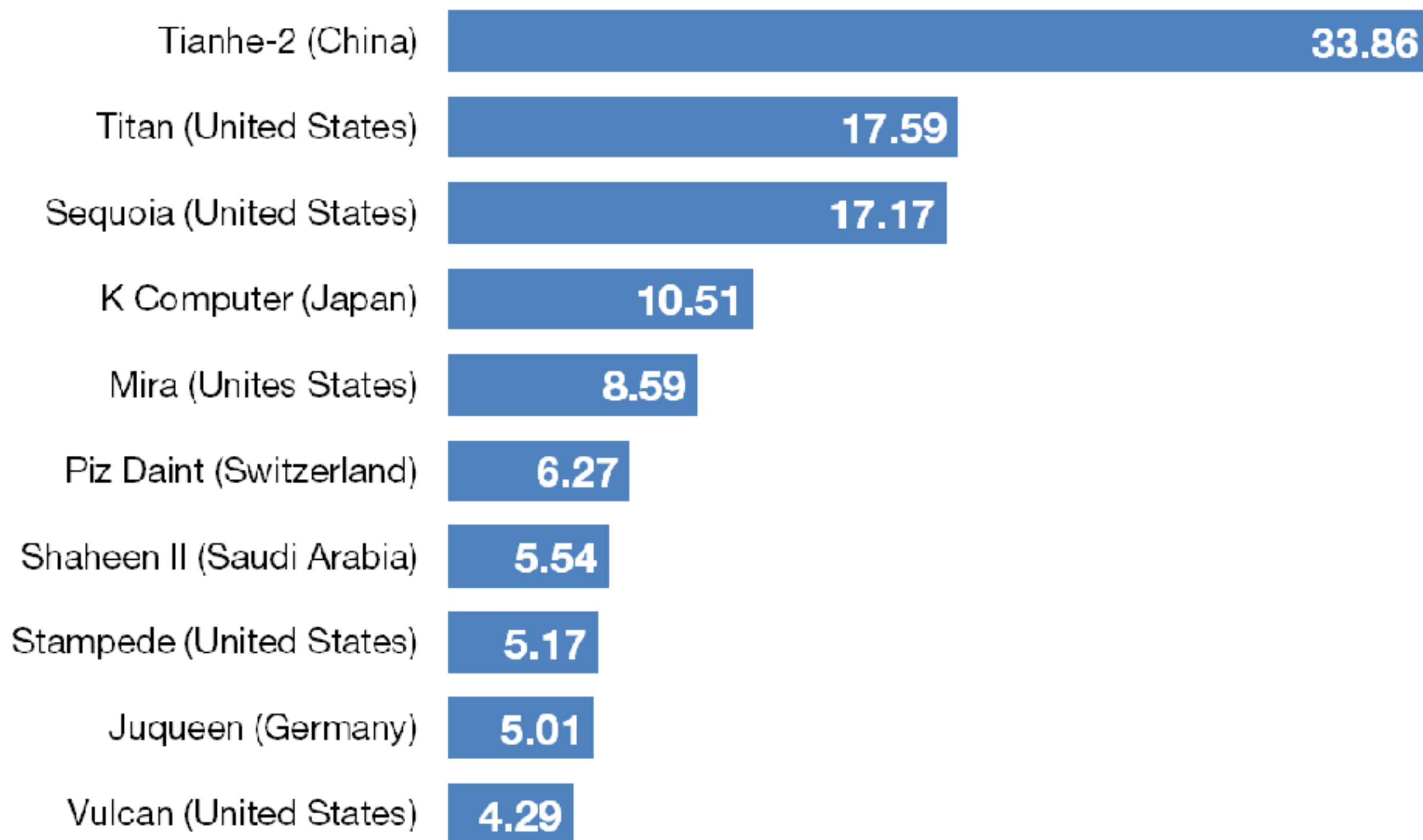
COMPUTER CLUSTERS AND IMPROVED PERFORMANCE

- So far we have been performing all of our numerical calculations on laptops.
- We also have been limiting our calculations to tasks that can finish in a reasonable time on our laptops.
- But what if we want to do calculations that would involve thousands or millions more operations.
- We could wait a month or longer for them to finish.
- Or we have two other options: We could instead run our code on a computer cluster or supercomputer, or we could write our code to be faster.
- We will first talk about computer clusters because making your code run faster on a single machine is based on the same ideas.



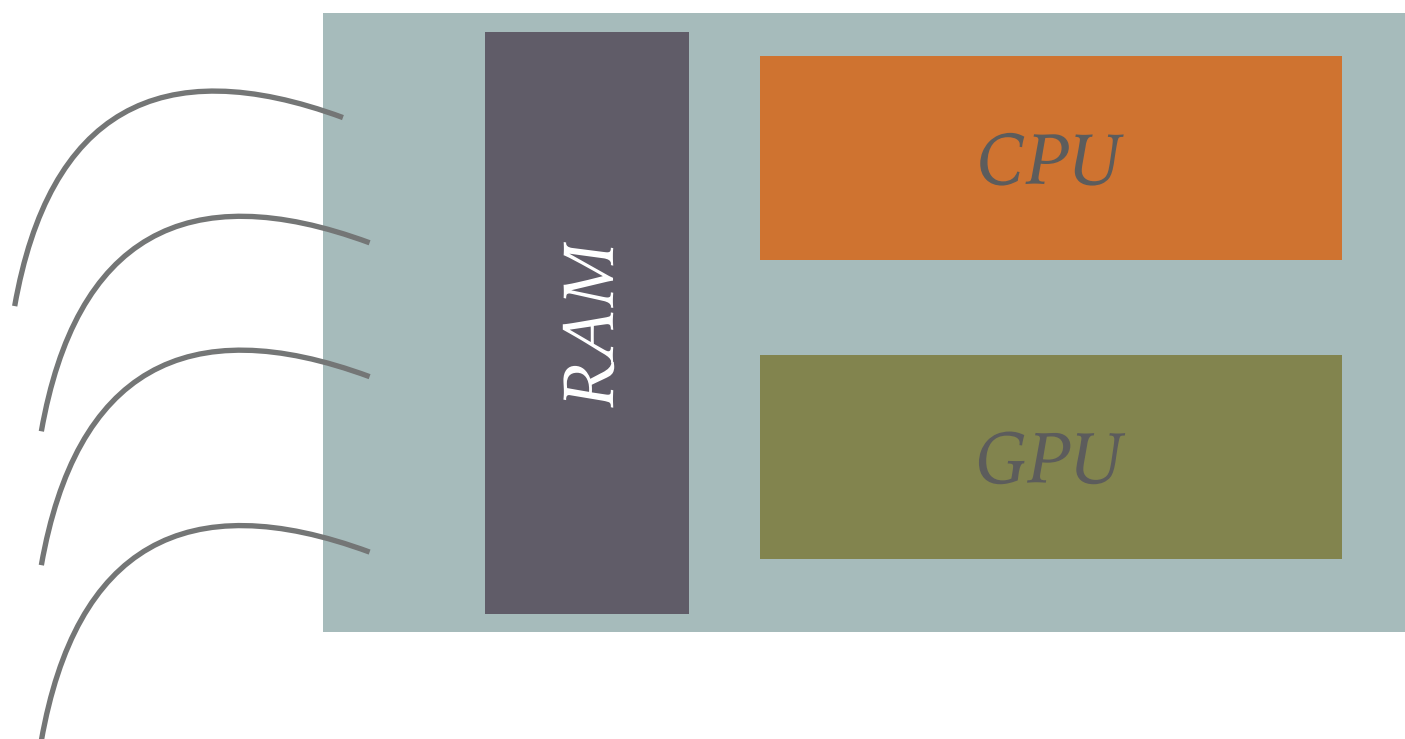
Top 10 supercomputers

Petaflop/s on the Linpack benchmark



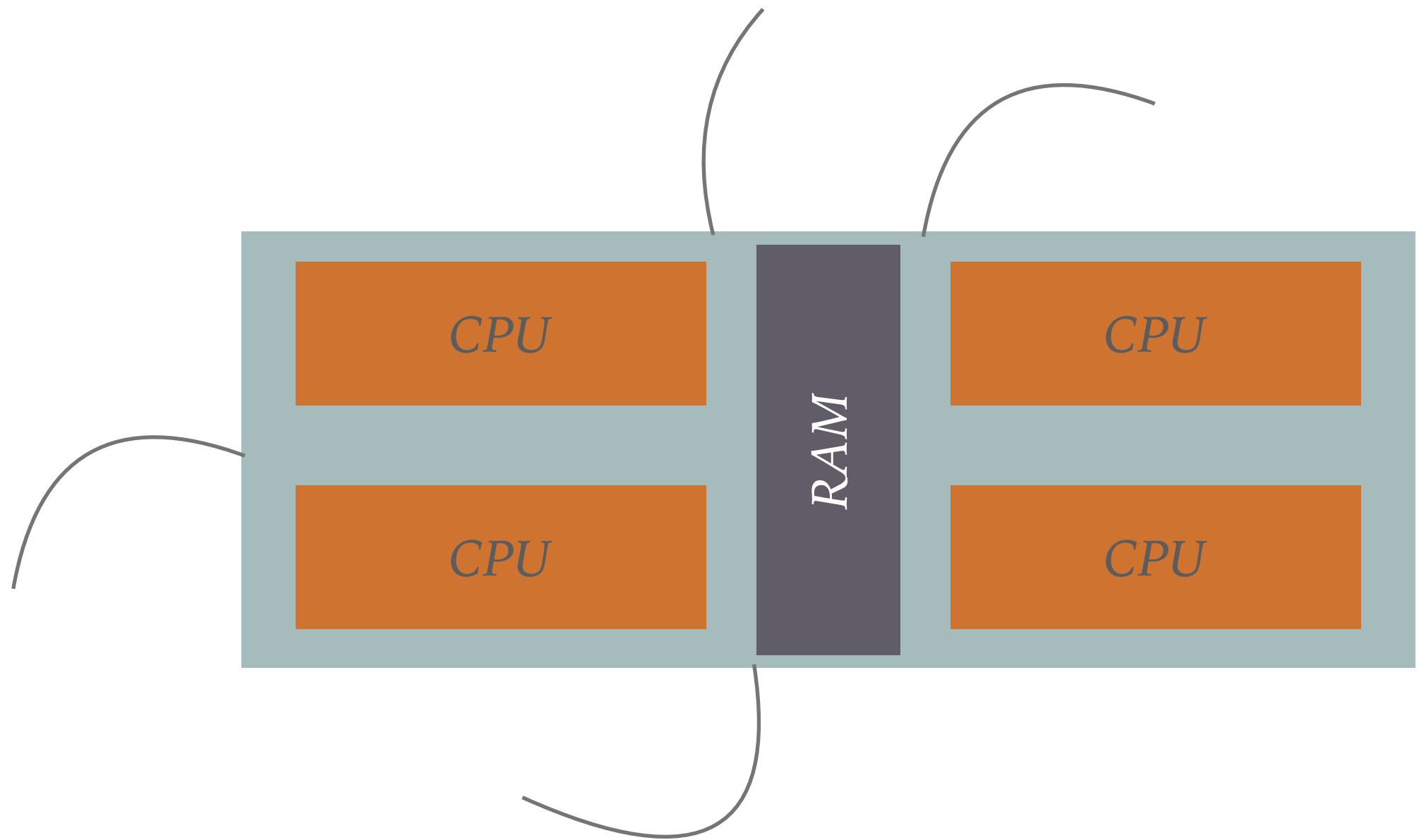
COMPUTE NODE

- Before we start discussing what a supercomputer is let us discuss what is a computer.
- A computer combines a number of hardware pieces into a single device. Most importantly these include
 - CPU - central processing unit
 - GPU - graphical processing unit
 - RAM - random access memory
 - Hard Drive - memory storage
 - Input devices - keyboard, mouse, screen
 - Output devices - screen, speakers



COMPUTER NODES

- However, things aren't that simple anymore. Nowadays chips are designed to act as if they contain more than one CPU.
- Language can get a bit confusing here. Let's think of the CPU as a single chip. But the ability to process instructions lets us call a core.
- A multi-core processor (dual-core, quad-core, etc.) means that on one chip you can run multiple instructions at the same time.
- In addition, a chip can have multi-threading. This is an additional way to get multiple things done, but it isn't a separate one having a different core. That is, it can't take different instructions, but can branch out part of an instruction.
- Finally on a single motherboard you can place more than one CPU. The different CPUs will share the same RAM. This gives us the modern basic unit of a computer, the *computer node*.
- In summary a single computer node can have multiple CPUs. Each CPU can have multiple cores and each core can have multiple threads.



Compute Node

COMPUTER CLUSTER

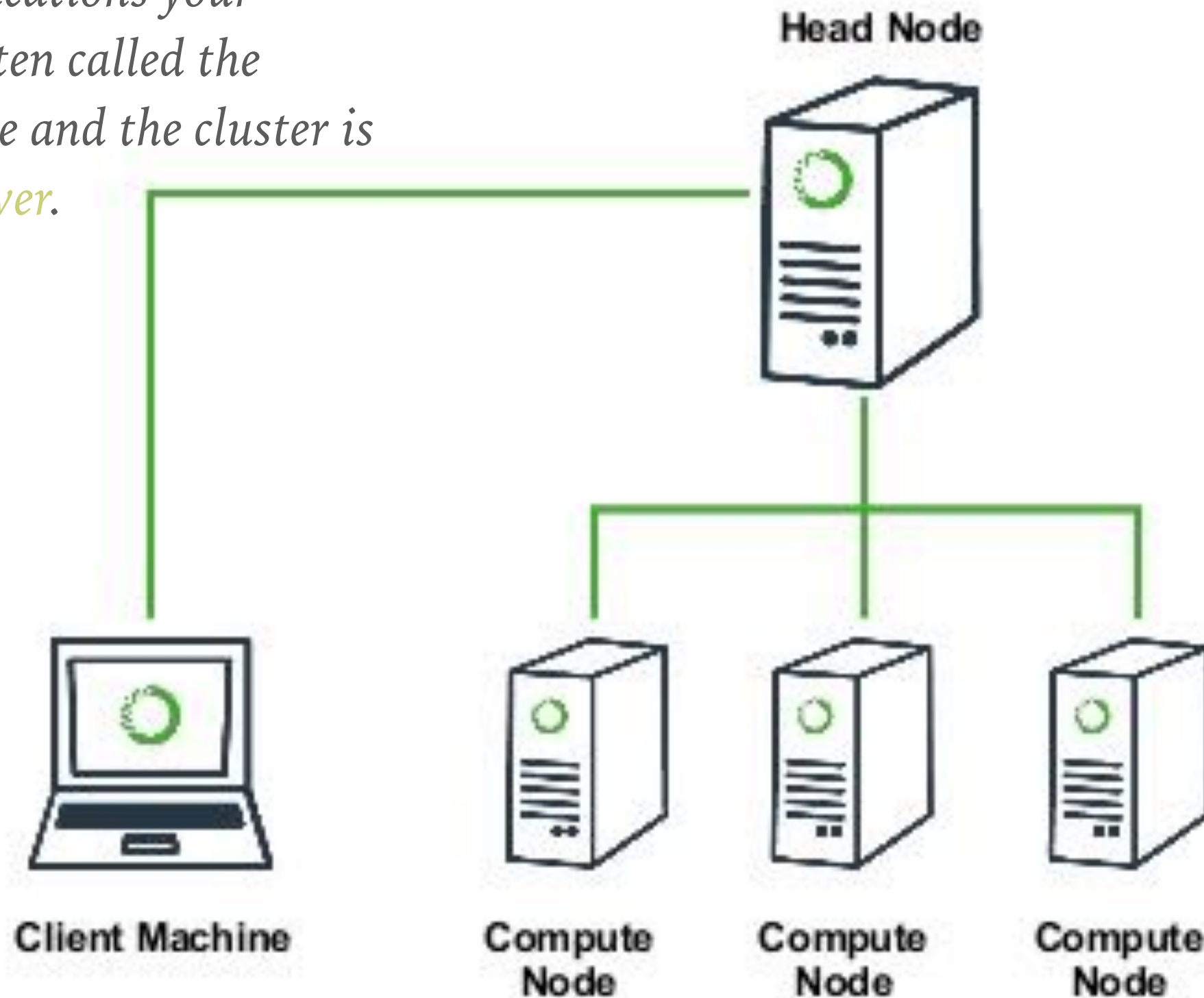
- A computer cluster is made of many compute nodes. This can be as few as 4 or 8 and as many as 40,960 (Sunway TaihuLight current world champ).
- A cluster with thousands of nodes is called a supercomputer.
- Each of the nodes can contain multiple cores, the Sunway TaihuLight has 256 cores per node so this supercomputer has a total of 10,649,600 CPU cores which basically can act like 10 million computers running at once.
- However, this does not mean this supercomputer is 10 million times faster than your laptop. A lot depends on what you are using it to do.

CLUSTER ARCHITECTURE

- The most common way the nodes are set up in a cluster is to have one head node that tells all the other nodes what to do.
- This way normal users only interact with this head node and tell it what they want it to do. The head node then takes those instructions (a job) and makes decisions as to which compute nodes should run different pieces of code.
- In order to be more efficient, jobs are usually placed in a queue. The job will wait there until the resources it needs are freed up. Unlike on your laptop the process doesn't start running as soon as you hit enter. Instead your job may wait in the queue for a long time before it starts running.
- This has the huge advantage that you don't have to sit around waiting until resources are freed up. Instead you simply submit your job and the head node will run it as soon as it possibly can.

Cluster Architecture Diagram

For web applications your machine is often called the *client* machine and the cluster is called the *server*.



REMOTE LOGIN

- As a user, it is likely you will never have physical access to the cluster. Instead you will log in to the head node remotely.
- This is almost always done with the ssh (secure shell) protocol. On Linux and Mac you will have this program installed. On Windows you will need to install a program that runs ssh, Putty is one choice (and it is free).
- On the command line ssh works like this:

`ssh username@remotehost`

- alternatively you can leave out the username and it will ask
- you will need to give a password after this step.

SSH CONFIG

- There is a useful config file for ssh that lives in the director `.ssh` under your home directory.
- Like all unix programs there are dozens of things you can set in the file but the most useful are your username and an alias for the host. For example:

Host `ursa`

 Hostname `ursa.major.edu`

 User `JohnDoe`

 HostKeyAlias `ursa`

- This allows one to just type `ssh ursa` and it knows the host name and username.

PUBLIC KEYS

- public/private key sharing is a more secure way to access remote machines than using a password.
- The idea is you create a key that can be sent to other machines and then it recognizes you when you log in. The public key recognizes your private key, but can't be used to recover it.
- To create a key in a terminal you can use `ssh-keygen`. There are tons of options but just `ssh-keygen` should get you started.
- This will ask for a passphrase and a location for the keys, using the default should create public and private keys in `.ssh/` as the files `id_rsa.pub` and `id_rsa`. The `.pub` one is your public key which you need to copy to any machine you wish to use it for.



EX 13.1

.....

- Let's create a set of keys and share them with your GitHub account. If for some reason you need to use ssh to connect to GitHub this will be useful.
- First generate a key with ssh-keygen if you don't already have one.
- Then copy your key. You can use `pbcopy < .ssh/id_rsa.pub` or open it in a text editor and copy it.
- Go to GitHub settings SSH and GPG keys and add the key. Test it in the terminal with `ssh -T git@github.com`

PBS SUBMISSION SCRIPT

- In order to run a job on a cluster you have to write what is called a submission script. This is a file that contains the instructions for what you would like head node to do.
- You submit with something like
`qsub my_submission_script.pbs`
- You can check on your submission with the command `qstat`.
- There are many other commands and options that are dependent on the program being used to control the submission queue.
- Your script might look something like:

PBS SUBMISSION SCRIPT

```
#!/bin/sh
```

```
#PBS -N job-name
```

```
#How many nodes and processors per node (ppn) you request:
```

```
#PBS -l nodes=4:ppn=32
```

```
#PBS -j oe
```

```
#PBS -o outputfile
```

```
echo "Starting job"
```

```
cd $PBS_O_WORKDIR
```

```
python mycode.py
```

```
python more_code.py
```

```
echo "Job finished"
```

PARALLELIZATION

- To make use of multiple nodes or cores you need to send different instructions to each core. For this to be helpful the key question is how parallel is the task you want to perform.
- For example if your code was the following:

```
for i in range(N):
```

```
    k=k+i
```

- then a different core can do nothing helpful in this calculation, because every addition needs to know what was done previously. That is the code is 100% serial.

PARALLELIZATION

- On the other hand if your code was this:

```
for i in range(N):
```

```
    array[i]=array[i]+i
```

- then each of these additions could be done on a different core. This operation is 100% parallel.
- In general coding contain both parts that can run in parallel and parts that can not. The degree to which a problem can be made to run in parallel is the degree to which using more cores will make it run faster.
- Some tasks are 100% parallel, like solving an ODE for 100 different boundary conditions. This is essentially just running the same code 100 times so it can very easily be run on 100 different cores.
- 100% parallel tasks can actually be run on 100 different computers, but you may want to combine the results are start the process based on some common conditions so that is the advantage to running on the same machine.

NODES AND CORES

- A parallel operation runs the same on a core on the same node or on a different node, but there is an important difference.
- Cores on the same node share memory. But cores on different nodes do not. So if different processes need access to the same variables, then they must be copied to the RAM of the other node.
- This can also be an advantage, one of the uses of clusters is to break up memory needs across many nodes. So if you had a $100,000 \times 100,000$ matrix, that would exceed the RAM on any machine. But you could break that matrix up into many smaller matrices and distribute them onto many nodes.

NODES AND CORES

- In either case, communication between nodes becomes a key aspect of parallel programming. Communication between nodes is much slower than operations on a chip, so a program can get stuck waiting for information from another node and not computing anything.
- Thus both the operations and the memory must be parallelized to use a cluster effectively. The degree to which this is possible will determine how a program will scale on a cluster.
- Even if not running on a cluster it can be useful to think about such things because your single CPU is probably multi-cored and multi-threaded so the more parallel your code it will probably still run faster.

HOW TO PARALLELIZE

- Writing parallel code can be done a number of ways.
- At the lowest level you can write the different parts of code to be run on each processor and then write the code to start them, run them and control how they communicate.
- Most scientific users do not do this. Instead they use libraries that will handle much of the parallelization details for you. These libraries include MPICH and OpenMPI.
- MPI stands for Message Passing Interface and it takes care of some of the hardest parts of parallel programming, getting the information back and forth between nodes.

RUNNING PARALLELIZED CODE

- When you run parallelized code you don't just type a command on the command line, but instead you run a program from MPI (if using MPI) that will then deal with sending the different processes to different cores.

- Your command would look something like

`mpiexec -n 16 python mycode.py`

- the `-n` says you want to use 16 cores. You then pass `python` as the program you want to run. And then to `python` you pass your code that you want to run.
- So `mpiexec` is running `python` and `python` is running your code.



OPTIMIZING CODE

.....

- As we've discussed modern computers are actually mini clusters because the compute node contains multiple cpus, cores, threads and GPUs.
- One can take advantage of these capabilities when writing code.
- The easiest way to do this is by using JAX.

JAX

- JAX is a package for array-oriented numerical computation. It can be used to replace numpy and run code much faster. It also includes a number of features to improve numerical performance.
- One it offers just in time (jit) compiling, this is a way to make functions much faster if you call them many times.
- Two it includes automatic differentiation, this allows you to take derivatives for all functions you write automatically.
- Finally it can be run on graphical processing units (GPUs) or even tensor processing units (TPUs). Note to take advantage of the last point you have to install JAX for your GPU which will differ by machine. For TPUs you need to run on Google Collab.

JAX

The easiest way to start using JAX is just to use

```
import jax.numpy as jnp
```

instead of numpy. Most np routines are mapped to jnp so you can just use them like your numpy code

```
x = jnp.linspace(0,6.*jnp.pi)
```

```
y = jnp.sine(x)
```

```
print(jnp.sum(y),jnp.max(y),jnp.mean(y))
```

JAX

- There are some differences between np and jnp. Most importantly JAX arrays are immutable, which means you can't change a value in an array. Like `x[5]=12`.
- Also, random numbers work somewhat differently because different threads need to have different seeds. So one has to split the keys like this

```
subkey = jax.random.split(key, num=1)
```

- But aside from these issues you should be able to translate numpy code into JAX pretty smoothly.

JIT

- JAX also includes just in time compiling. This compiles a function and will make it run faster after the first time.
- There are two ways to add jit to your code

```
from jax import jit
```

```
def my_cool_func():
```

```
    return stuff
```

```
jit_cool_func = jit(my_cool_func())
```

or use a decorator

```
@jit
```

```
def my_cool_func():
```

```
    return stuff
```

GRAD

- You can also take the gradient of any function you write using grad. Higher order derivatives can also be called, e.g. hessian.
- All you need to do is import grad and use it

```
from jax import grad
```

```
def my_cool_func():
```

```
    return do stuff
```

```
d_my_func = grad(my_cool_func)
```

MULTIPROCESSING

- As mentioned earlier modern cpus have multiple processors on a single chip. While the python interpreter will try to make use of this you can explicitly run different parts of your code on different processors.
- The multiprocessing package can be used to send functions in your code to different processors. This is just like MPI, but because the processors are on the same chip you don't need MPI to pass information between nodes.
- This enables you to parallelize aspects of your code for running on a single compute node.
- One can use the package thread to control to send code to separate threads in a similar way.

```
import multiprocessing
```

```
def print_cube(num):
```

```
    """ function to print cube of given num"""
```

```
    print("Cube: {}".format(num * num * num))
```

```
def print_square(num):
```

```
    """function to print square of given num"""
```

```
    print("Square: {}".format(num * num))
```

```
if __name__ == "__main__":
```

```
    # creating processes
```

```
    p1 = multiprocessing.Process(target=print_square, args=(10, ))
```

```
    p2 = multiprocessing.Process(target=print_cube, args=(10, ))
```

```
    p1.start()    # starting process 1
```

```
    p2.start()    # starting process 2
```

```
    p1.join()    # wait until process 1 is finished
```

```
    p2.join()    # wait until process 2 is finished
```

```
    print("Done!")    #both processes finished
```