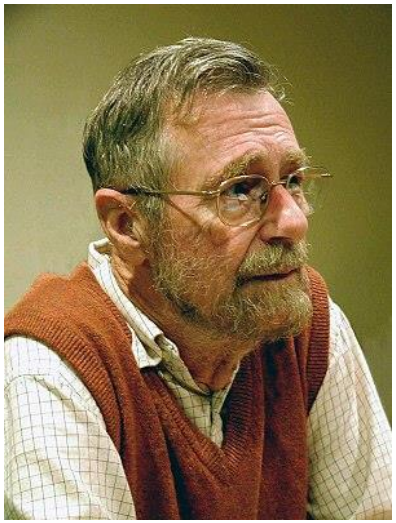
"as long as there were no machines, programming was no problem at all;

when we had a few weak computers, programming became a mild problem, and

now we have gigantic computers, programming has become an equally gigantic problem."



pic: https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

- Edgar Dijkstra, 1972 Turing Award Lecture

A Gigantic Computer

- System 360 / Model 91

Source: https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP2091.html

HPC Clusters and Programming

NSM Nodal Center for Training in HPC and AI, IIT Madras

Nikhil Hegde, IIT Dharwad

March 20, 2021

What is a Cluster ?

- Gigantic computer
  - from interconnecting several smaller computers

VIRGO Super Cluster, IIT Madras. Source: https://cc.iitm.ac.in/node/184

What is a Cluster ?

- Gigantic computer
  - from interconnecting several smaller computers
- Compute power in the order of 10^{15} floating point operations per second (Peta* FLOPS)
  - Your `i7`-based personal computer – few Giga FLOPs (10^9)

** 10^{12} is tera, 10^{18} is exa.*

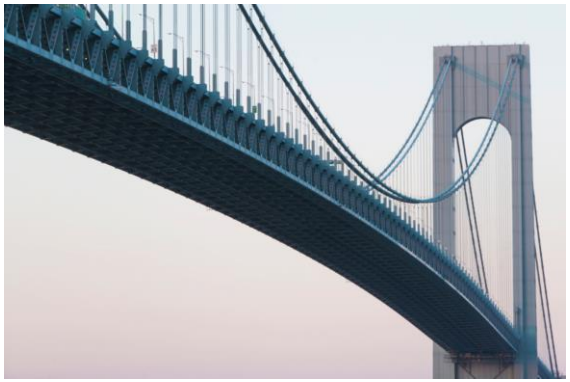
What is a Cluster ?

- Gigantic computer
  - from interconnecting several smaller computers
- Compute power in the order of 10^{15} floating point operations per second (Peta* FLOPS)
  - Your `i7`-based personal computer – few Giga FLOPs (10^9)
- E.g.
  - Chandra (IIT Palakkad), Virgo (IIT Madras), AnantGanak (IIT Dharwad) etc.

** 10^{12} is tera, 10^{18} is exa.*

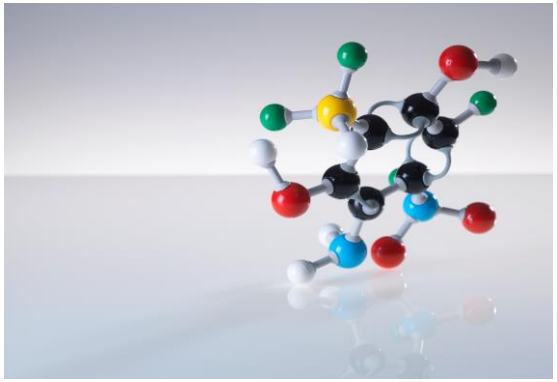
Why Clusters?

Financial Analysis



Design Simulation

Oil Exploration

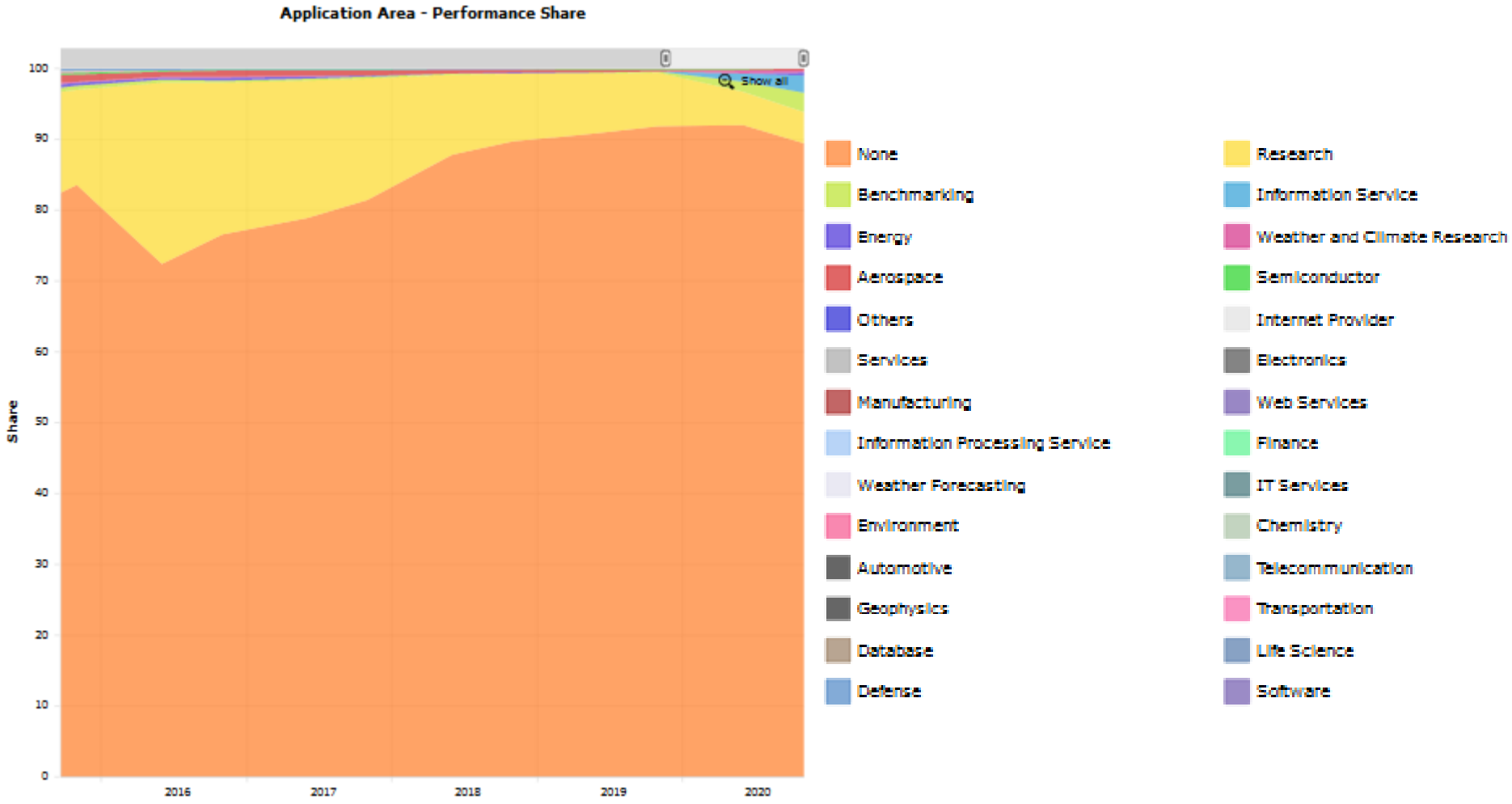


Genomics

pic source: stock images

Weather Prediction

Why Clusters? Application Areas



source: top500.org

Terminology - Cluster Elements

- Processing Element
  - Core, CPU, Node, GPU, Virtual CPUs
- Storage
- Interconnect

Hardware

- Partition
- Job and Job Scheduler
- Operating System (OS), Software Development Tools, Applications

Software

- *Infrastructure* – power, cooling,

Processing Element

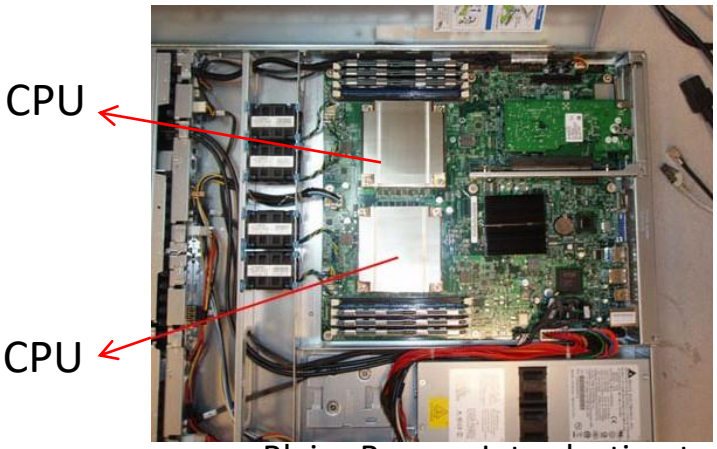
- Node
  - Standalone computer
  - Comprised of multiple CPUs/Processors/Cores, memory, network interfaces.

Each Green dot is a Node

  - `Master / Log-in` node: is what end user interacts with (think: operator's console)

Processing Element

- CPU/Processor and Socket
  - No consensus on terminology. Some vendors call multi-core CPUs as sockets.

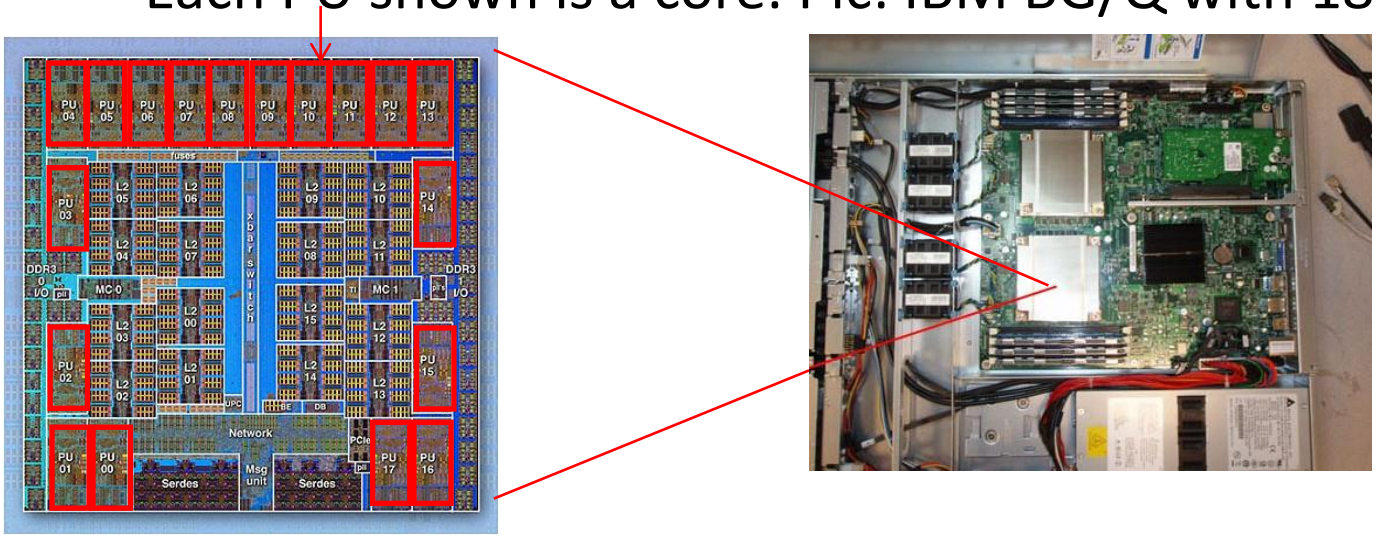


source: Blaise Barney, Introduction to Parallel Computing, https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

  - Socket can also be a place to plug a CPU. E.g. dual-socket motherboard in pic.

Processing Element

- Core
  - Each PU shown is a core. Pic: IBM BG/Q with 18 Cores

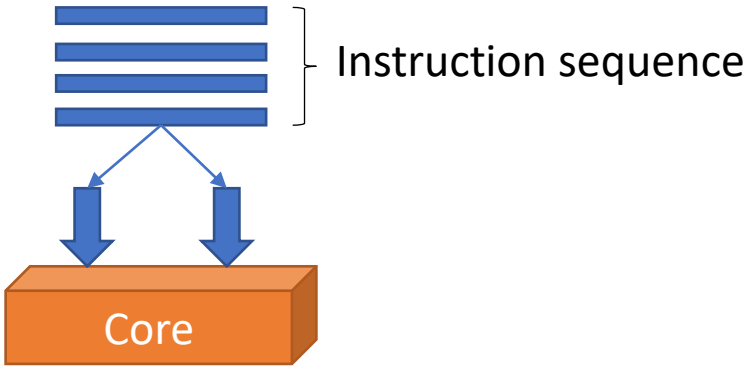


source: Blaise Barney, Introduction to Parallel Computing, https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

  - In the past, each CPU (with just one core) was a single execution unit. Now, each core is an independent execution unit.

Processing Element

- Thread (hardware)
  - Pathway for flow of instruction within a core
  - When exposed to the OS, the OS gives an illusion to the programmer that multiple cores exist (“HyperThreading”)

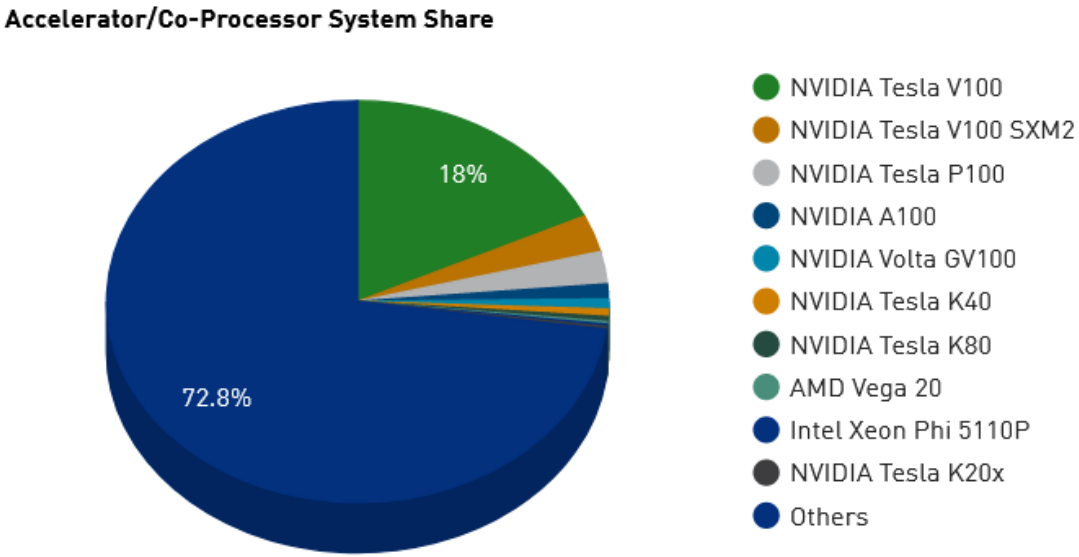


Processing Element

- Virtual CPUs
  - A term that you often get to hear when working with ‘Cloud’ hosted systems
  - Virtual Machine (VM) assigned to a single physical core
    - A VM is an abstraction/emulation of a computer

Processing Element

- GPU (Graphics Processing Unit) – Add-on devices
  - Traditionally: accelerate image creation
  - Now: GPGPUs for large-scale modeling, genetic programming



source:top500.org

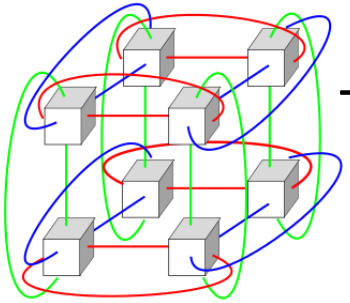
Cluster Elements

- Interconnect
  - Nvidia-Mellanox Switch

source: https://www.nvidia.com/en-in/networking/infiniband-switching/

  - Cabling (determines topology)

https://www.rcac.purdue.edu/training/clusters101/



Torus

source: https://en.wikipedia.org/wiki/Torus_interconnect

Cluster Elements

- Storage
  - $HOME
    - Landing directory when you log in to the master node
  - $SCRATCH and/or Parallel File System
    - A fast memory where you should keep all the data needed for executing the task
    - E.g. Lustre, BeeGFS, etc.
  - (Optionally) storage may be available on each compute node e.g. /tmp
  - (Optionally) archival storage (e.g. LTO-6, Storage Server (e.g. IBM DS3512))

Cluster Elements - Software

- Job
  - A task performed by the cluster
  - Represented by a set of commands to the cluster, captured in a `script`, to precisely tell how to execute the task
  - Usually, the set of commands do not require your intervention i.e. non-interactive
    - You issue the commands (read: “submit a job”) and go for coffee..

Cluster Elements - Software

- Batch System - Job Scheduler and Resource manager
  1. Provide a user interface to submit, monitor, and run jobs
  2. Manage the computational resources mentioned previously
  3. Implement the usage policies set by HPC Admin
- E.g. SLURM (Simple Linux Utility for Resource Management), PBS (Portable Batch System) – Torque, Moab.

Cluster Elements - Software

- Partitions/Queues
  - A logical grouping of (a subset of) nodes in the cluster
  - Single node can belong to multiple partitions (not done in practice)
  - Define attributes (and limits) for a job submitted to a Q

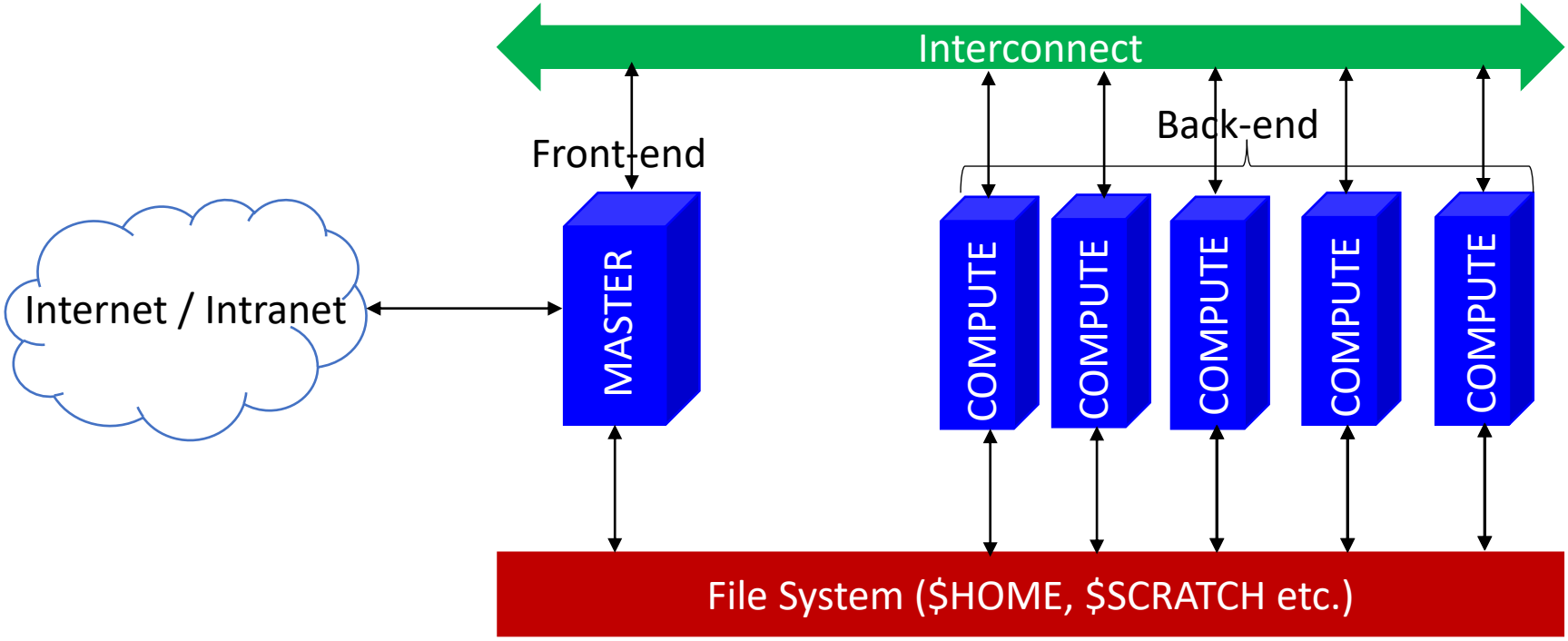
    E.g.
    - A job submitted to a specific partition runs with low priority
    - A job can request a maximum of 4 cores
    - etc.

Cluster Elements - Software

- Operating System (OS), Software Development Tools, Applications
  - Linux-based OS in 100% of the supercomputing clusters in top500.org (2015 onwards)
  - Compiler tool chains, Runtime systems, Profilers
  - E.g. GCC, ICC, MPICH, JDK, Docker, Matlab, Intel Parallel Studio, NVProf, Tau etc.

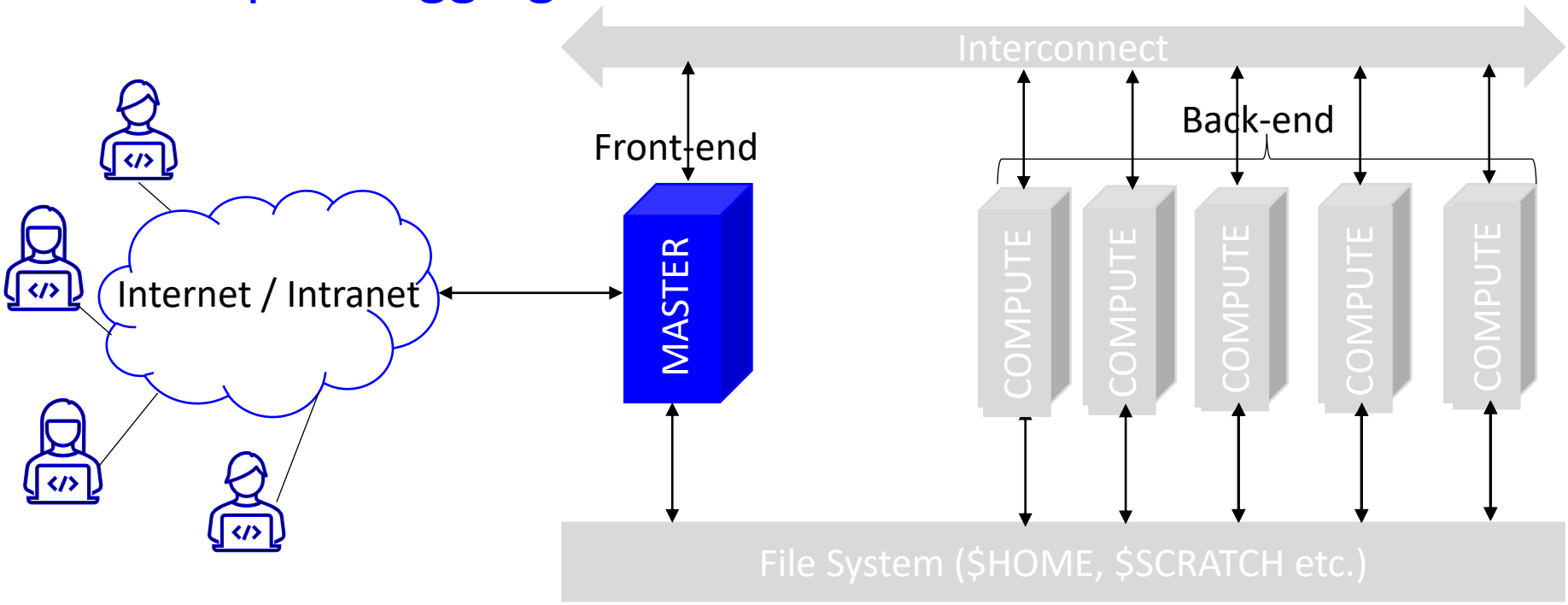
Recap

- Clusters



Clusters

- Step 1: Logging-in

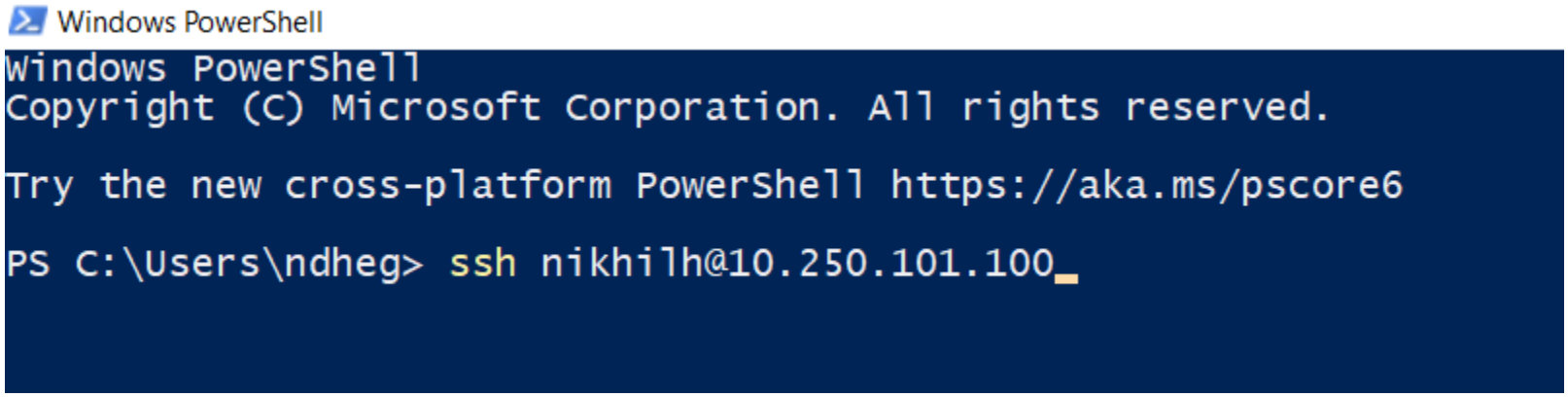


Logging-in

- Logging into remote Linux system (master node) requires you to use SSH ("Secure Shell Protocol")
  - Login credentials are encrypted
  - `SSH server` must be running on the system that you are logging into; Happens on most Linux systems by default.
  - `SSH client`, another piece of software, is used to authenticate and connect to the `SSH server`
    - Client software is available for all platforms (OSs)

Logging-in Windows

- Powershell on Windows 10
  - Press (Windows + R) -> Type “powershell”
  - Type “ssh <username>@<masternode_IP_address>
  - Type ‘Yes’ when prompted (only first time)
  - Provide log-in credentials



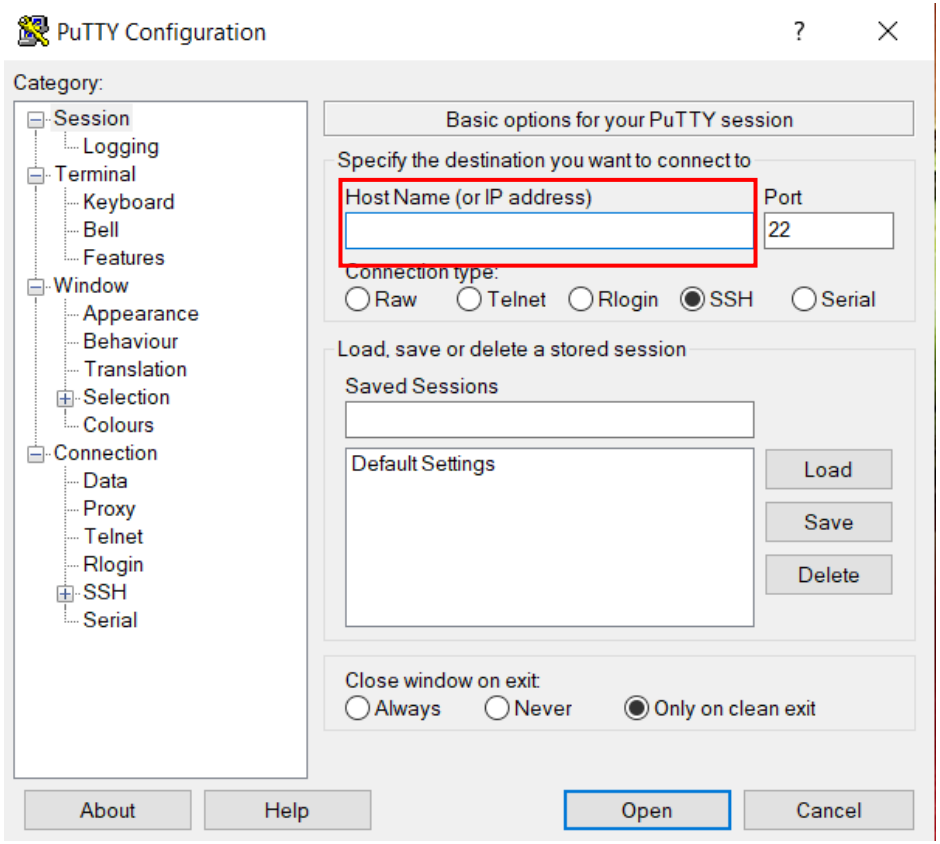
Logging-in Windows

- PuTTY SSH client Windows
  - Download PuTTY from https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html (64-bit .exe)
  - Double click on the 

icon after downloading

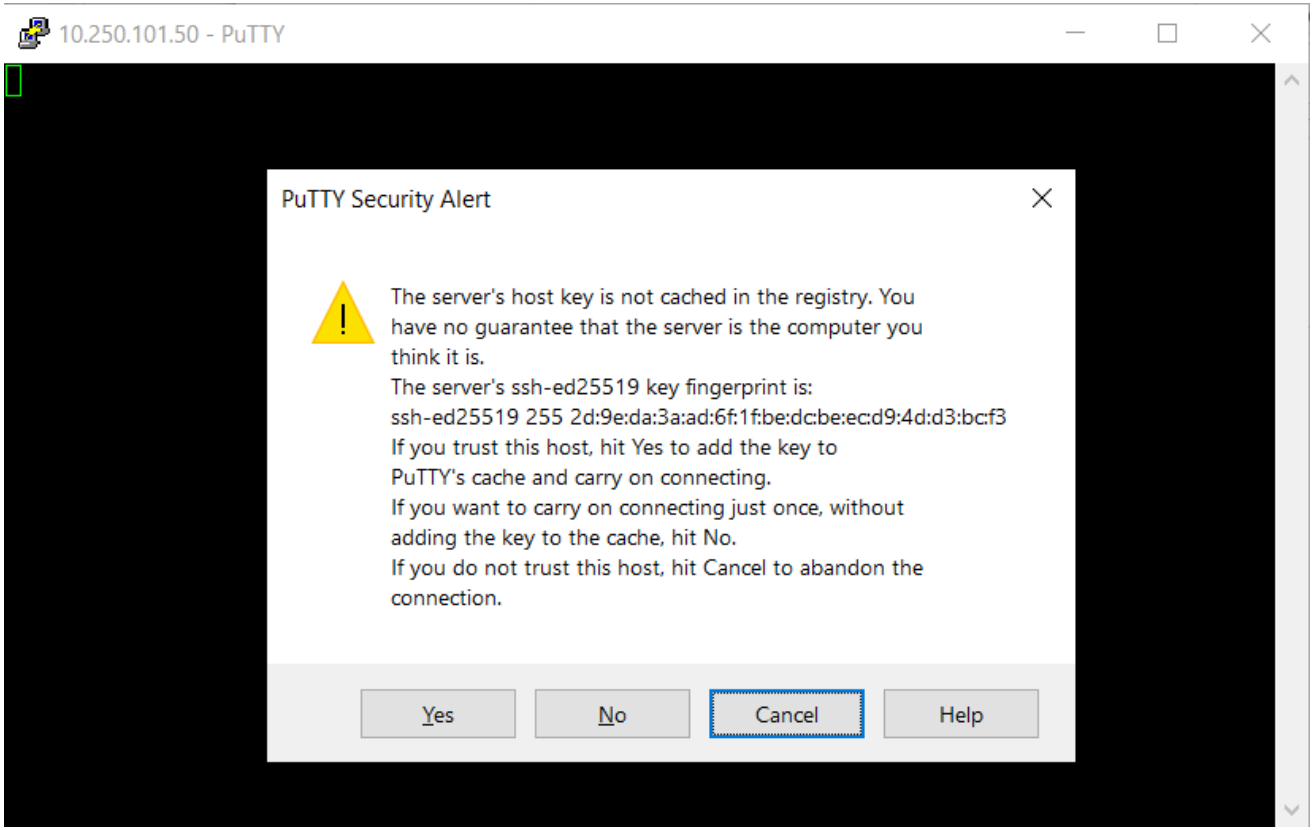
Logging-in Windows

- Type in the Host Name / IP address and click ‘Open’



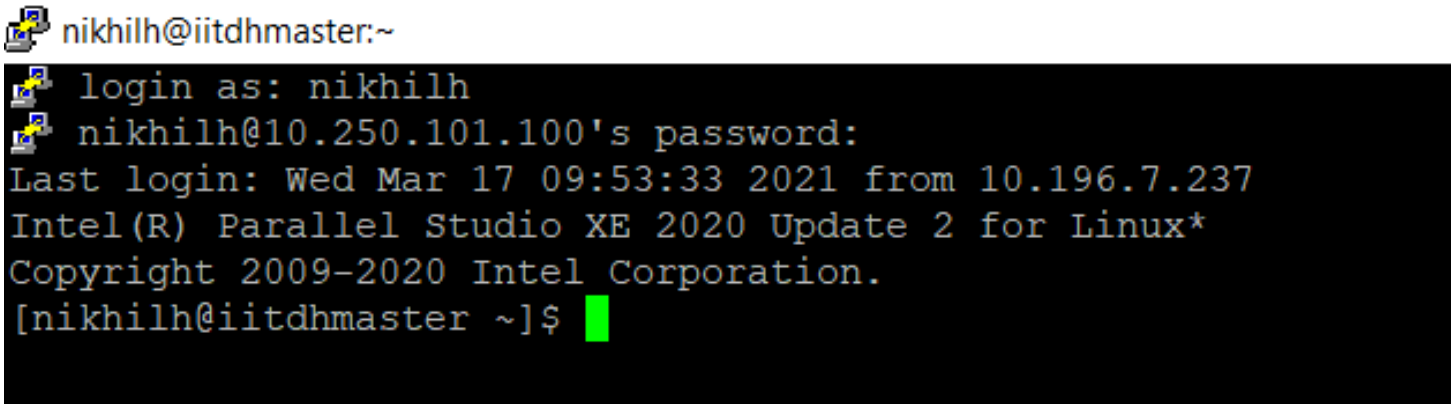
Logging-in Windows

- Click on ‘Yes’ (you are accepting the server host key)



Logging-in Windows

- Enter log-in credentials



Logging-in MAC

- Open the ‘Terminal’ program on MAC (Go -> Applications -> Terminal)

```
Last login: Sun Mar  7 11:35:13 on ttys000

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
apples-MacBook-Pro:~ apple$
```

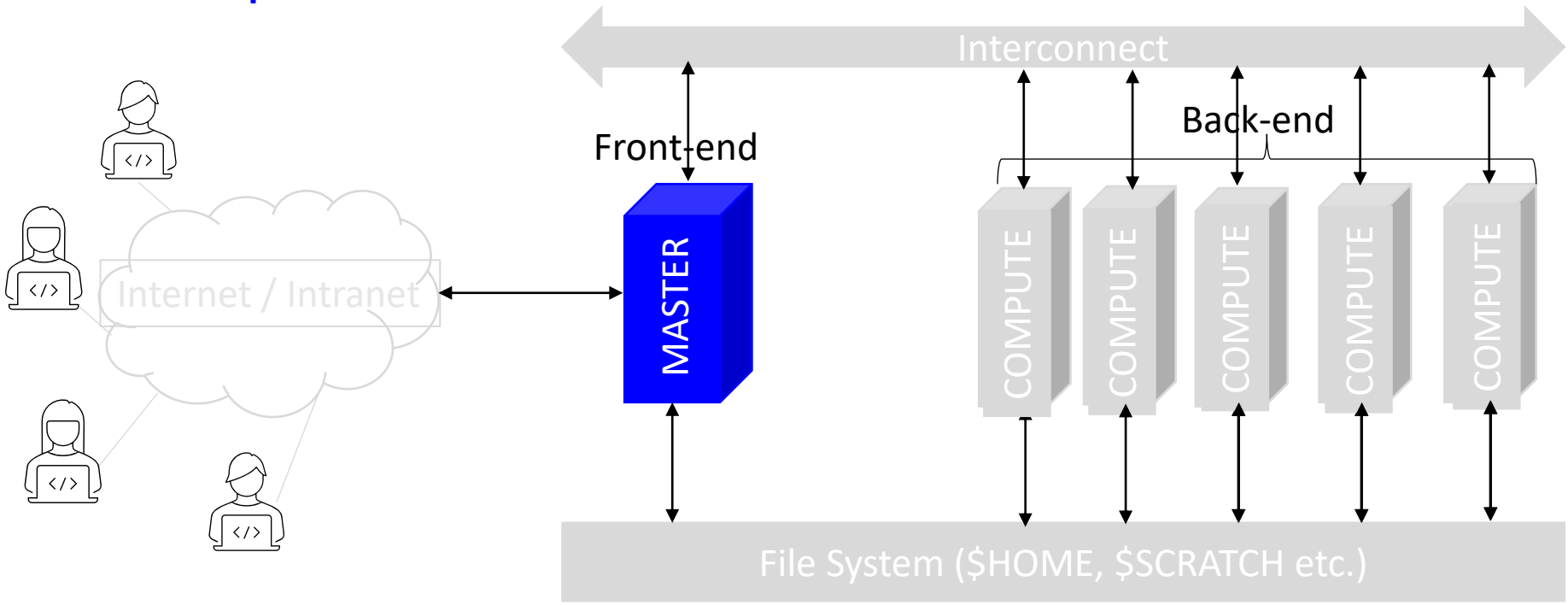
- Type “ssh <username>@<masternode_IP_address>
- Type ‘Yes’ when prompted (only first time)
- Provide log-in credentials

Logging-in Linux

- If you are a Linux user, you know what a ‘Terminal’ is ☺
- Type “ssh <username>@<masternode_IP_address>
- Type ‘Yes’ when prompted (only first time)
- Provide log-in credentials

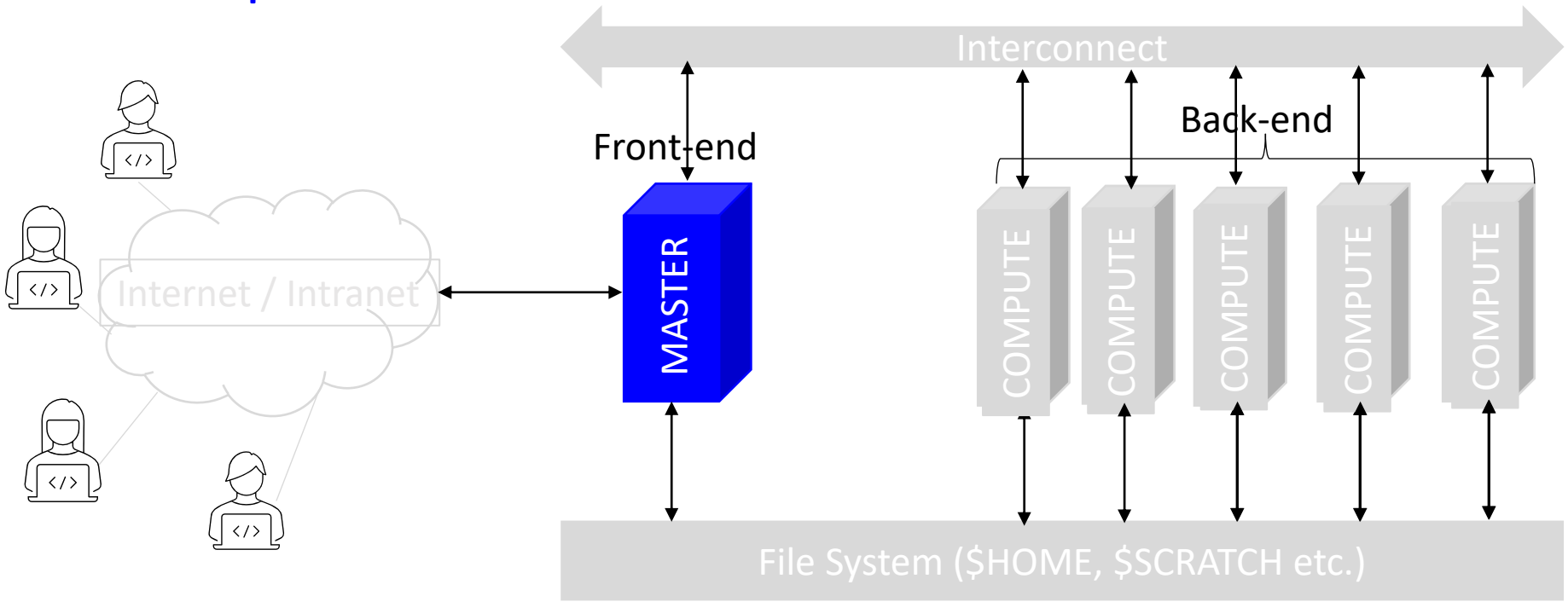
Clusters

- Step 2: Activities on the Master Node



Clusters

- Step 2: Activities on the Master Node



Master node runs Linux-OS. Let's review useful Linux commands

Useful Linux Commands

```
ls, ls -l
man
mkdir
cd
pwd
cp
mv
scp
rm  //use with caution!
```

```
cat
less
head, tail
vi, vim, emacs, nano, pico
gzip, tar, zip
who
cut
wc
bc
echo
```

Type "`man <command_name_here>`*" on the Linux terminal to get help info*

Useful Linux Commands - `scp`

`scp` - To move files back and forth between master node and your local system

```
scp file1 <user_name>@<master_node_ip>
```

from your system to Master node

from Master node to your system

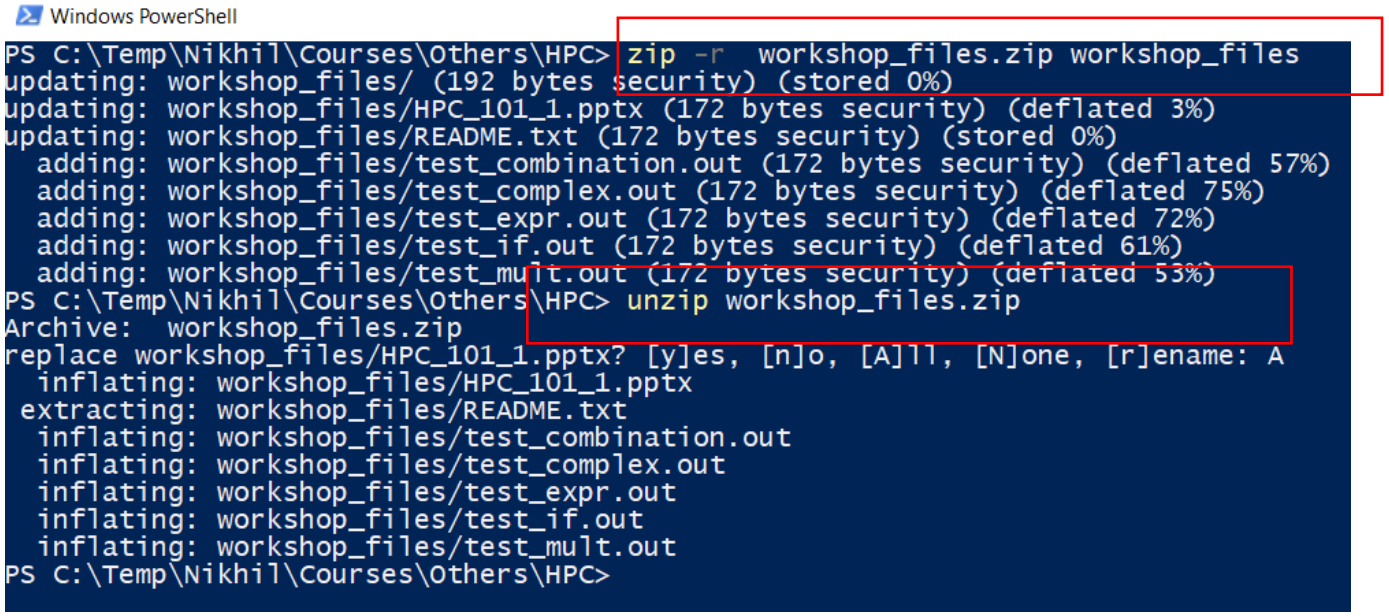
```
scp <user_name>@<master_node_ip>:file1 .
```

Useful Linux Commands – `zip, unzip`

`zip, unzip` - To compress/uncompress folders/directories

```
zip –r compressed.zip workshop_files/
unzip compressed.zip
```

`-r` for recursively applying the compression to folders within



Useful Linux Commands – `tar`

`tar` – **T**ape **Ar**chive to compress/uncompress folders/directories

```
tar –cvf workshop.tar workshop_files/
tar -xvf workshop.tar
```

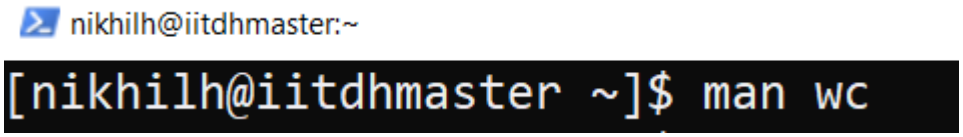
Type `man tar` to know about flags

`tar` followed by `gzip` compression:

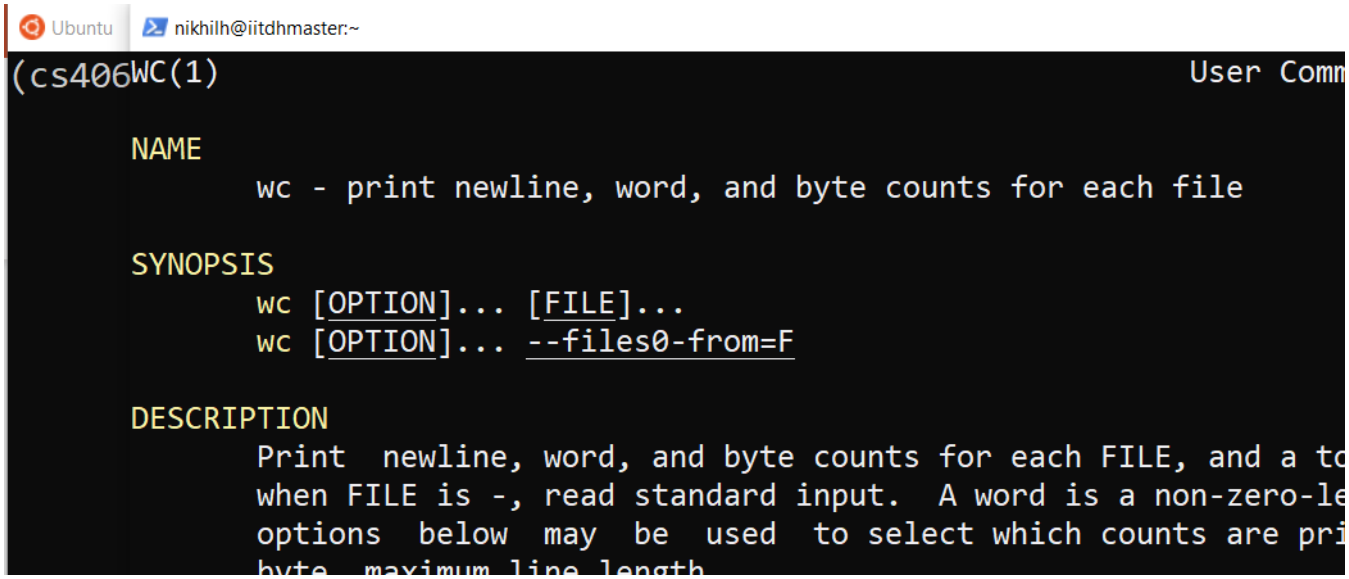
```
tar –czvf workshop.tar.gz workshop_files/
tar -xzvf workshop.tar.gz
```

Useful Linux Commands - **man**

- type `man <command>` and hit `Enter` key to get help



- type `q` to quit. Use arrows to scroll



Useful Linux Commands

Other utility commands

`cd HPC` → change directory to HPC

`cd ..` → change directory to parent

`vim hello.txt` → open a file hello.txt for editing. See vi commands

`ls` → list files in the current directory

`head hello.txt` → display the first few lines of hello.txt

`cat hello.txt` → display entire content of hello.txt

`pwd` → display the name of the present working directory

`who` → display the names of all users who have currently logged-in

`stdout, stdin, stderr` in Linux

- stdout
  - Output that is printed to screen (terminal)
- stdin
  - Keyboard input
- stderr
  - Error messages printed to screen (terminal)

also called streams (input stream, output stream, error stream)

redirects and pipes in Linux

//redirect standard output to file1.txt

- `echo “hello world” > file1.txt`

//feed input to the cat command from file1.txt rather than keyboard input.

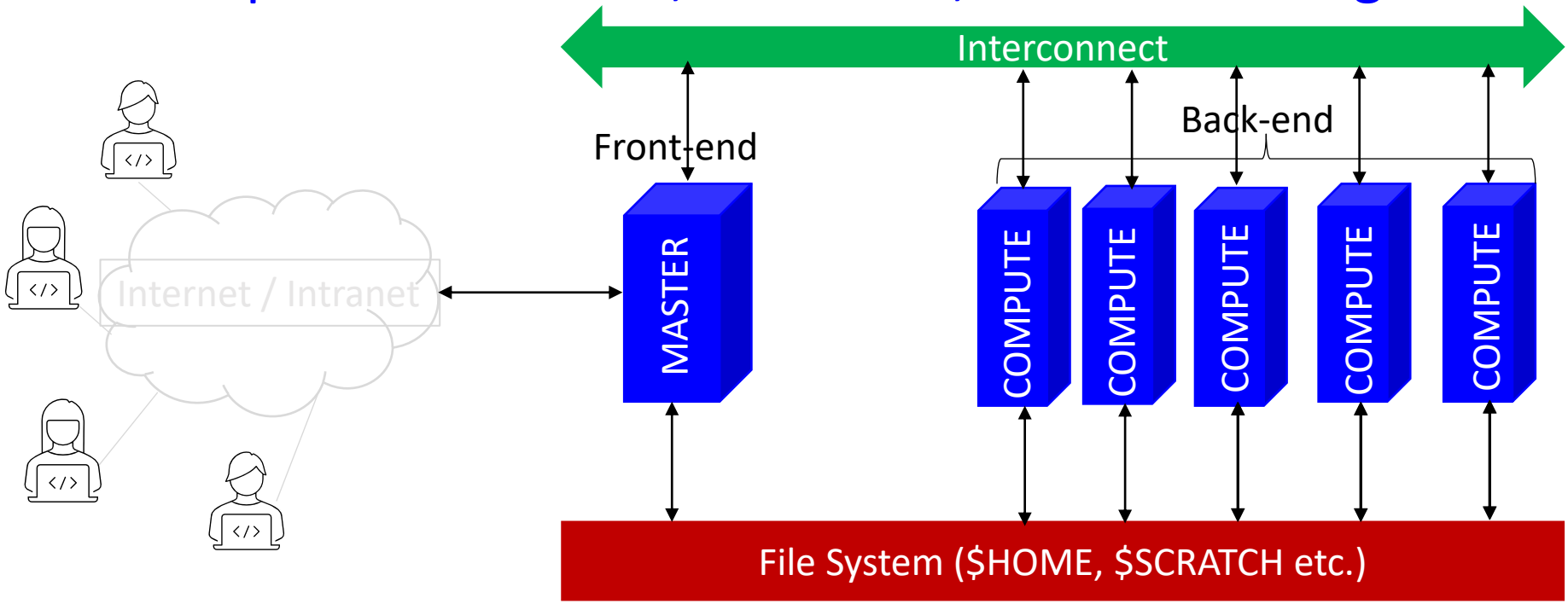
- `cat < file1.txt`

//create a pipeline, where the output of `echo` command is fed to input of `bc` command.

- `echo “100+200” | bc`

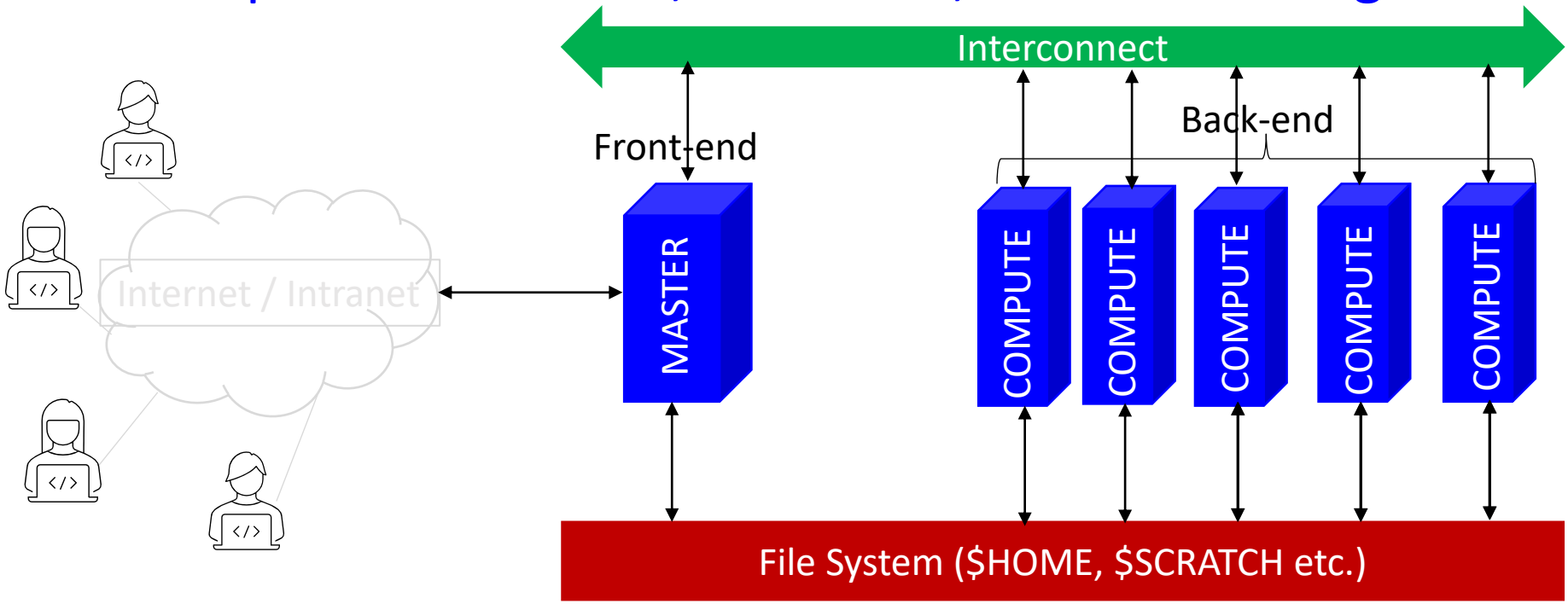
Clusters

- Step 3: Job creation, execution, and monitoring



Clusters

- Step 3: Job creation, execution, and monitoring



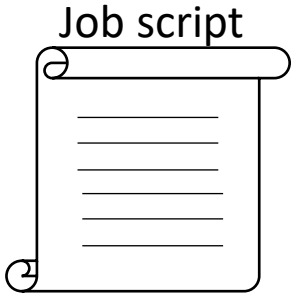
User must go through the Batch System for this step

Life of a Job

- Four Phases
  1. Creation
  2. Submission
  3. Execution
  4. Completion

Life of a Job - Creation

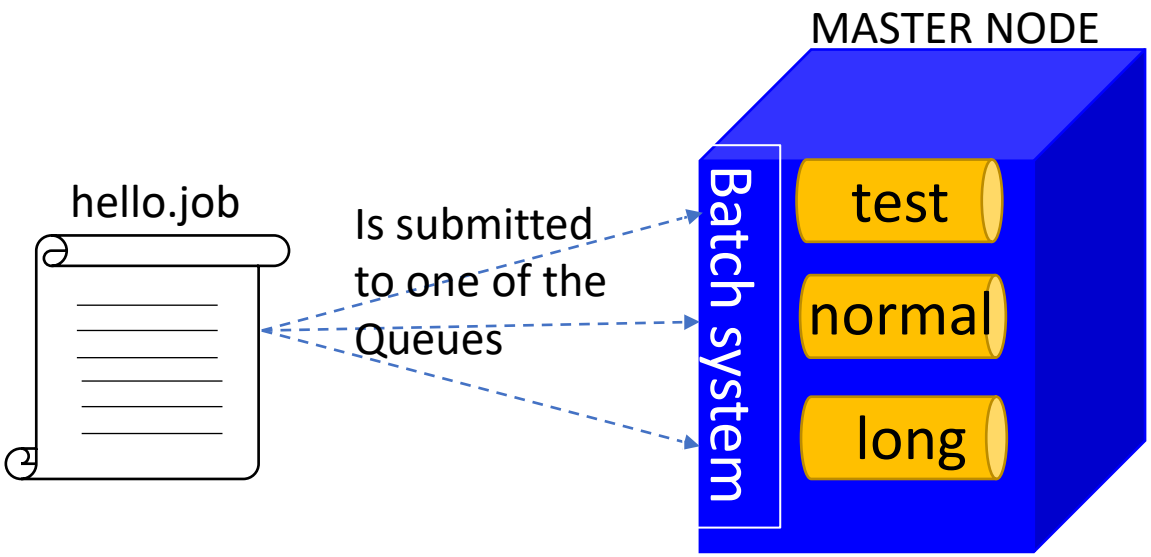
- Job creation specifies:
  1. What resources are needed (at least the following)
     - Maximum number of nodes and cores needed
     - Maximum amount of time needed
     - Maximum amount of memory needed
     - Whether exclusive access to a node is needed i.e. jobs from other users should run on the node simultaneously or not.
     - The partition or queue to run the job in
  2. What commands will be run in the job (actual task)
     - ./a.out
     - mpirun –np 10 helloworld
     - etc.
  3. **Output:** Job script



Life of a Job - Submission

- Successful submission implies:
  - No syntax errors in the specification
  - Amount of resources requested could possibly be allocated
  - A job object is created in the queue in which it would be executed
  - A job ID is returned to the user for reference
- Once the job is submitted (and before it is executed), you can change resource requirements (part 1) but not the actual task (part 2)

Life of a Job - Submission



- E.g. command: `sbatch hello.job`
- `test`, `normal`, and `long` are the names of queues/partitions on my cluster

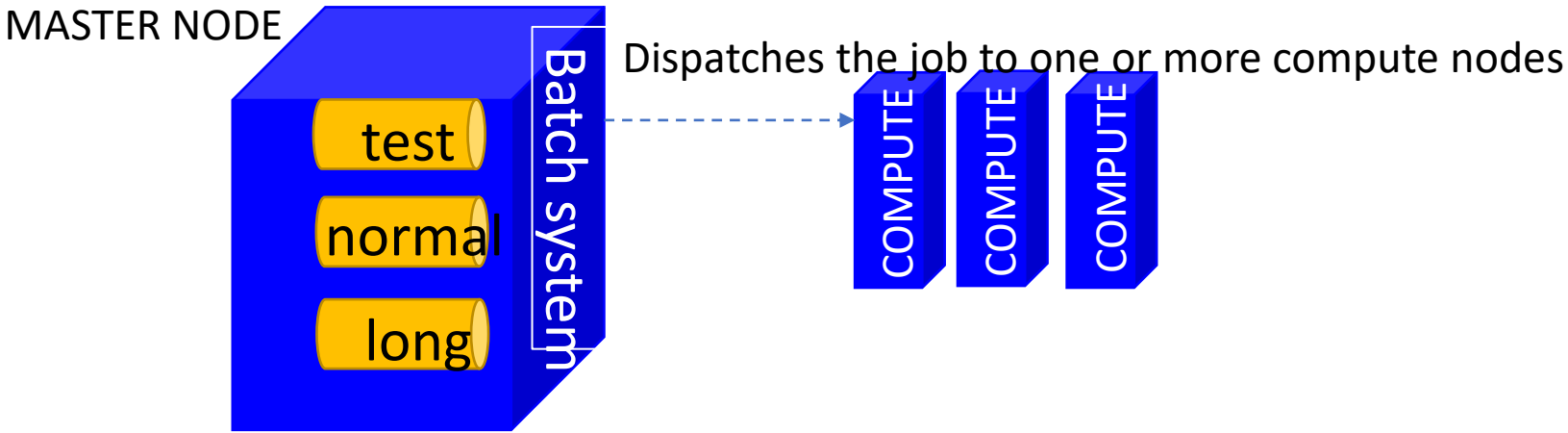
Queues / Partition

- E.g. command: `sinfo`

```
[nikhilh@iitdhmaster ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
test         up      30:00      1   idle cn01
long         up 6-00:00:00      4   idle cn[02-05]
normal*      up   12:00:00     26   idle cn[06-31]
gpu          up   12:00:00      1   idle dgx
[nikhilh@iitdhmaster ~]$
```

Life of a Job - Execution

- When requested resources become available
  - Job is launched on the compute nodes
  - The commands specified in part 2 are executed in sequence
- Neither the job resources nor the actual task can be changed at this time. If job exceeds requested resources, it is terminated and a notification is sent.



Job Monitoring

- check the status (squeue)

```
[nikhilh@iitdhmaster hpc101]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
               851    normal       sh  nikhilh  R       1:08      1 cn06
[nikhilh@iitdhmaster hpc101]$
```

- delete the job if required (scancel)

nikhilh@iitdhmaster:/iitdh/faculty/nikhilh/hpc101

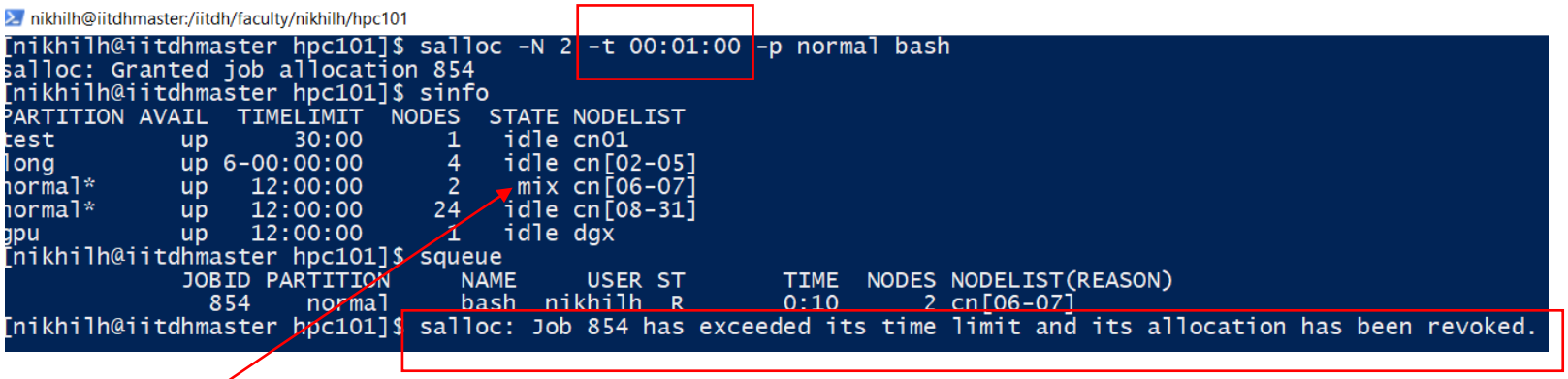
```
[nikhilh@iitdhmaster hpc101]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
[nikhilh@iitdhmaster hpc101]$ salloc -N 1 -p normal bash
salloc: Granted job allocation 852
[nikhilh@iitdhmaster hpc101]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
               852    normal     bash  nikhilh  R       0:07      1 cn06
[nikhilh@iitdhmaster hpc101]$ scancel 852
salloc: Job allocation 852 has been revoked.
[nikhilh@iitdhmaster hpc101]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
[nikhilh@iitdhmaster hpc101]$
```

Life of a Job - Completion

- Can happen because of normal completion of execution, abnormal termination, or job exceeding requested resources
- Output (stdout/err) stored in file
  - At the same location where job was submitted (default)
  - At a location mentioned in the job script
- All resources are reclaimed at this point and allocated to other jobs if needed

Life of a Job - Completion

- job exceeding requested time



- `mix` indicates non-exclusive access
- two nodes (indicated by `–N 2`) `cn06` and `cn07` are requested

Life of a Job - Completion

- job completing normally

```
[nikhilh@iitdhmaster hpc101]$ sbatch mwttest_large.job
Submitted batch job 855
[nikhilh@iitdhmaster hpc101]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
               855    normal mwttest_  nikhilh  R       0:03      8 cn[06-13]
[nikhilh@iitdhmaster hpc101]$ ls
hello.job  mwttest_large.job  slurm-844.out  slurm-845.out  slurm-855.out
```

- slurm-855.out contains the output of the program written to terminal (stdout)
  - In case of error you would have seen a file `mwttest_large.job.e855` and `mwttest_large.job.o855`

Life of a Job - Completion

- Checking the status of completed job (`qstat` in PBS)

```
[nikhilh@iitdhmaster hpc101]$ qstat -u nikhilh

iitdhmaster.iitdh.ac.in:
                                                                  Req'd  Req'd   Elap
Job id               Username Queue    Name                SessID NDS   TSK   Memory Time Use S Time
-------------------- -------- -------- ------------------- ------ ----- ----- ------ ----- - -----
855                  nikhilh  normal   mwttest_large.job   --         8    40     -- 00:30 C 00:00
```

- In SLURM (`sacct`)

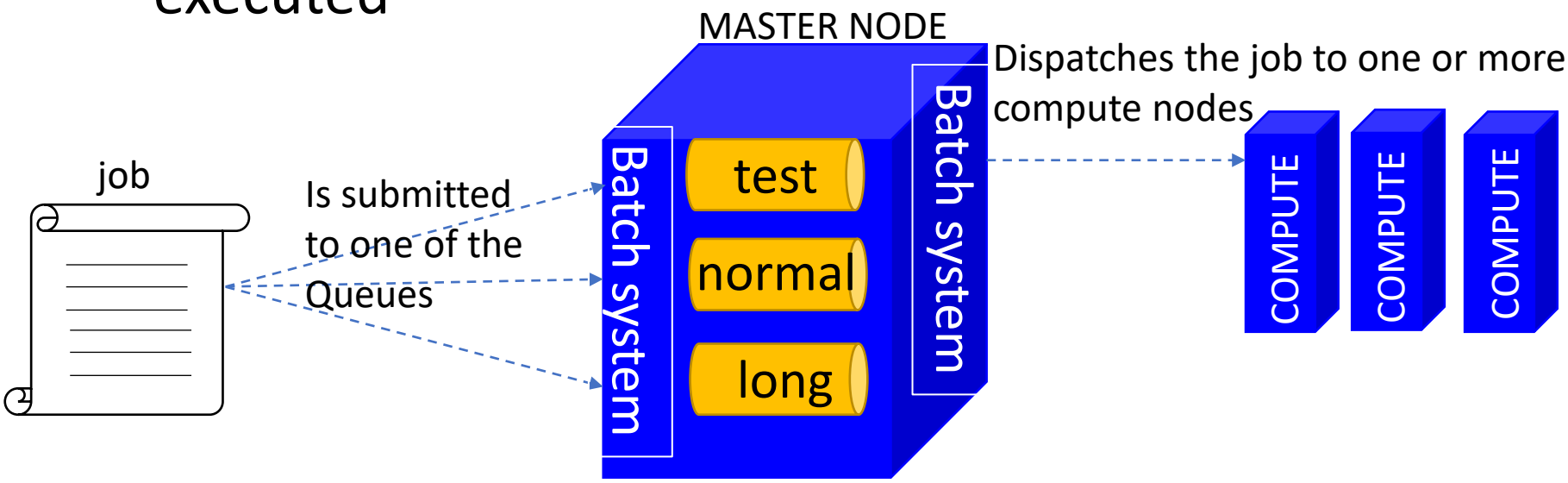
```
[nikhilh@iitdhmaster hpc101]$ sacct -j 855
       JobID    JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
855          mwttest_l+     normal       root         40  COMPLETED      0:0
855.batch         batch                  root          5  COMPLETED      0:0
855.0        hydra_bst+                  root          8  COMPLETED      0:0
[nikhilh@iitdhmaster hpc101]$
```

Recap - SLURM Commands

- `sbatch` → to submit a job to the queue for later execution (batch mode)
- `scancel` → to cancel a queued or running job
- `sinfo` → report system info (queues, nodes, etc.)
- `squeue` → report job status
- `srun` → to create a job and execute
- `sattach` → to connect stdin/out/err to an existing job
- `sreport` → report resource usage by user, account etc.
- `salloc` → to create a job and start a shell to use it (interactive mode)
- `sstat` → report accounting info about currently running jobs

Batch System (SLURM and PBS)

- Runs continuously (daemon) and monitors resources
- Scans the queues repeatedly
  - user jobs are submitted to queues
- Determines when and where the jobs are to be executed



SLURM and PBS

- Provide exclusive or non-exclusive access to compute nodes
- Provide a set of user-commands for
  - job control
  - node control
  - accounting
  - Many other things …

Creating Job Scripts

- Still haven't looked at inside of a `.job` file!

```
[nikhilh@iitdhmaster hpc101]$ cat mwttest_large.job
#!/bin/bash
#SBATCH -p normal
#SBATCH -t 00:30:00
#SBATCH -N 8
#SBATCH --ntasks-per-node 5
#echo commands to stdout
#set -x

#run MPI program
mpirun -np 40 /iitdh/faculty/nikhilh/Nikhil_test/exes/MWT
```

Creating Job Scripts

- Still haven't looked at inside of a `.job` file!

```
[nikhilh@iitdhmast
#!/bin/bash
#SBATCH -p normal
#SBATCH -t 00:30:0
#SBATCH -N 8
#SBATCH --ntasks-p
#echo commands to
#set -x

#run MPI program
mpirun -np 40 /iitdh/faculty/nikhilh/Nikhil_test/exes/MWT
```

First line. Always looks like this: first two characters are # and !. This line means that the job would be starting in job owner's login shell environment. In this case the shell is 'bash'

Creating Job Scripts

- Still haven't looked at inside of a `.job` file!

```
[nikhilh@iitdhmaster hpc101]$ cat mwttest_large.job
#!/bin/bash
#SBATCH -p normal
#SBATCH -t 00:30:00
#SBATCH -N 8
#SBATCH --ntasks-per-node 5
#echo commands to stdout
#set -x

#run MPI program
mpirun -np 40 /iitdh/faculty/nikhilh/Nikhil_test/exes/MWT
```

Means that the partition (p) to be used for placing the job is 'normal'

Creating Job Scripts

- Still haven't looked at inside of a `.job` file!

```
[nikhilh@iitdhmaster hpc101]$ cat mwttest_large.job
#!/bin/bash
#SBATCH -p normal
#SBATCH -t 00:30:00
#SBATCH -N 8
#SBATCH --ntasks-per-node 5
#echo commands to stdout
#set -x

#run MPI program
mpirun -np 40 /iitdh/faculty/nikhilh/Nikhil_test/exes/MWT
```

Means that the maximum time this job could take is 30 mins (note –t option)

Creating Job Scripts

- Still haven't looked at inside of a `.job` file!

```
[nikhilh@iitdhmaster hpc101]$ cat mwttest_large.job
#!/bin/bash
#SBATCH -p normal
#SBATCH -t 00:30:0
#SBATCH -N 8
#SBATCH --ntasks-p
#echo commands to stdout
#set -x

#run MPI program
mpirun -np 40 /iitdh/faculty/nikhilh/Nikhil_test/exes/MWT
```

Means that the 8 nodes are needed by this job (note –N option)

Creating Job Scripts

- Still haven't looked at inside of a `.job` file!

```
[nikhilh@iitdhmaster hpc101]$ cat mwttest_large.job
#!/bin/bash
#SBATCH -p normal
#SBATCH -t 00:30:00
#SBATCH -N 8
#SBATCH --ntasks-per-node 5
#echo commands to stdout
#set -x

#run MPI program
mpirun -np 40 /iitdh/faculty/ni
```

Means that the 5 cores on each of the nodes requested are needed. Remaining cores on the nodes can be given away to other jobs (because of non-exclusive access)

Creating Job Scripts

- Still haven't looked at inside of a `.job` file!

```
[nikhilh@iitdhmaster hpc101]$ cat mwttest_large.job
#!/bin/bash
#SBATCH -p normal
#SBATCH -t 00:30:00
#SBATCH -N 8
#SBATCH --ntasks-per-node 5
#echo commands to stdout
#set -x

#run MPI program
mpirun -np 40 /iitdh/faculty/
```

Any line beginning with a # is a 'comment' in a shell program. Previous lines have #SBATCH, which is interpreted by the batch system SLURM

Creating Job Scripts

- Still haven't looked at inside of a `.job` file!

```
[nikhilh@iitdhmaster hpc101]$ cat mwttest_large.job
#!/bin/bash
#SBATCH -p normal
#SBATCH -t 00:30:00
#SBATCH -N 8
#SBATCH --ntasks-per-node 5
#echo comma
#set -x

#run MPI pr
mpirun -np 40 /lfdn/faculty/nikhilh/Nikhil_test/exes/MWT
```

sets a flag on shell program. All executed commands are printed on the shell.

Creating Job Scripts

- Still haven't looked at inside of a `.job` file!

```
[nikhilh@iitdhmaster hpc101]$ cat mwttest_large.job
#!/bin/bash
#SBATCH -p normal
#SBATCH -t 00:30:00
#SBATCH -N 8
#SBATCH --ntasks-per-node
#echo commands to stdout
#set -x

#run MPI program
mpirun -np 40 /iitdh/faculty/nikhilh/Nikhil_test/exes/MWT
```

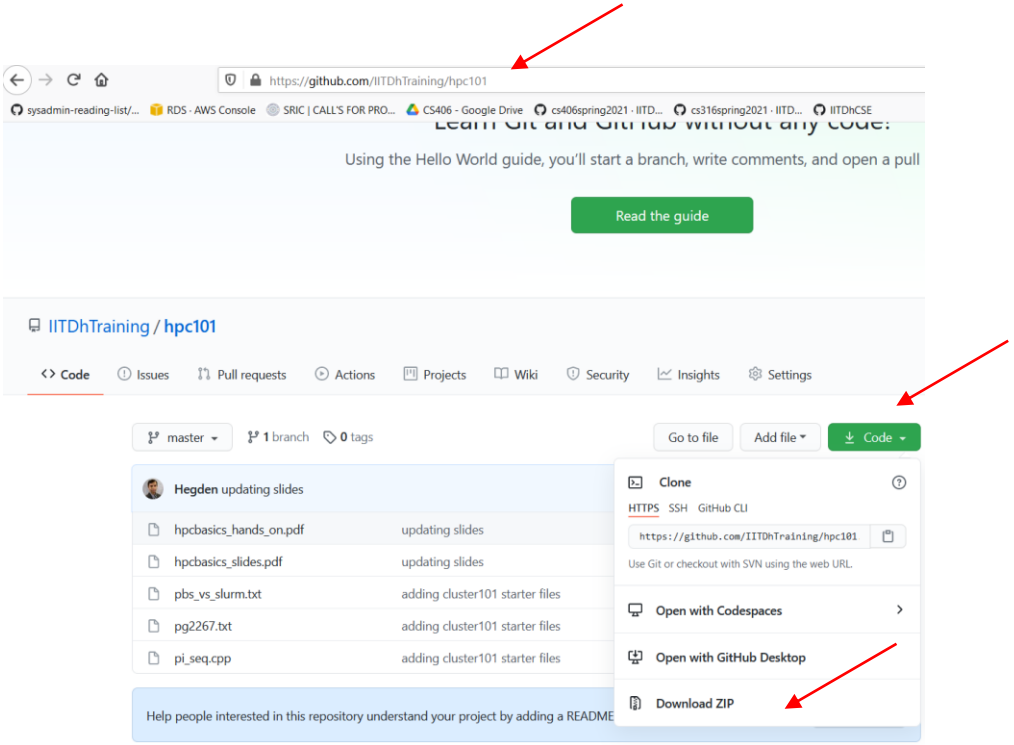
Part 2 of the specification. Command to perform the actual task. mpirun is the program that launches 40 copies of the executable MWT

SLURM/PBS Environment Variables

- $SLURM_JOB_ID
- $SLURM_JOB_NAME
- $SLURM_SUBMIT_DIR
- $SLURM_NTASKS
- $SLURM_NTASKS_PER_NODE
- $SLURM_JOB_NUM_NODES
- $PBS_NODEFILE
  - (in SLURM you will have to do `srun hostname`)

Hands-on Session

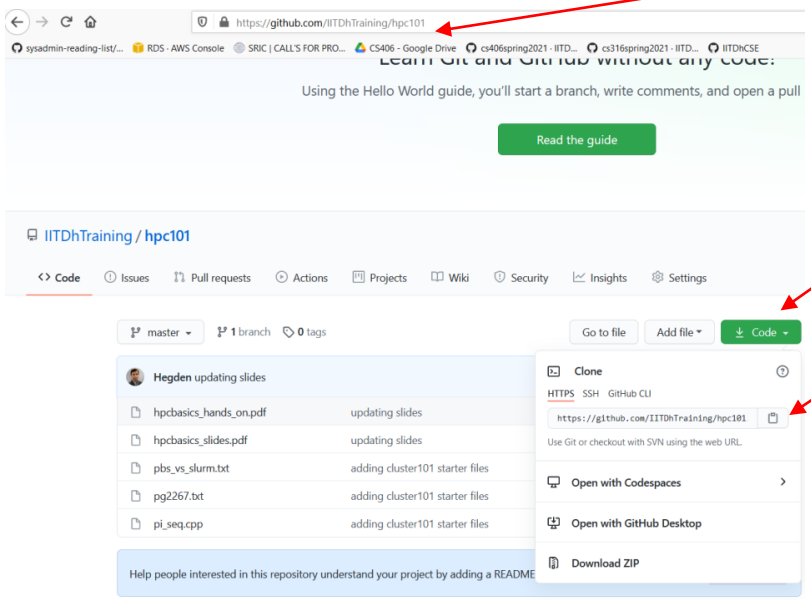
- visit https://github.com/IITDhTraining/hpc101 and download the repository as .zip file to get started



Hands-on Session

- You could also ‘clone’ the repository https://github.com/IITDhTraining/hpc101 to get started

visit the website



click on Code

click the icon to copy the text in the adjoining box

- Type on terminal: `git clone <paste the text copied>`

Clusters – How are they programmed?

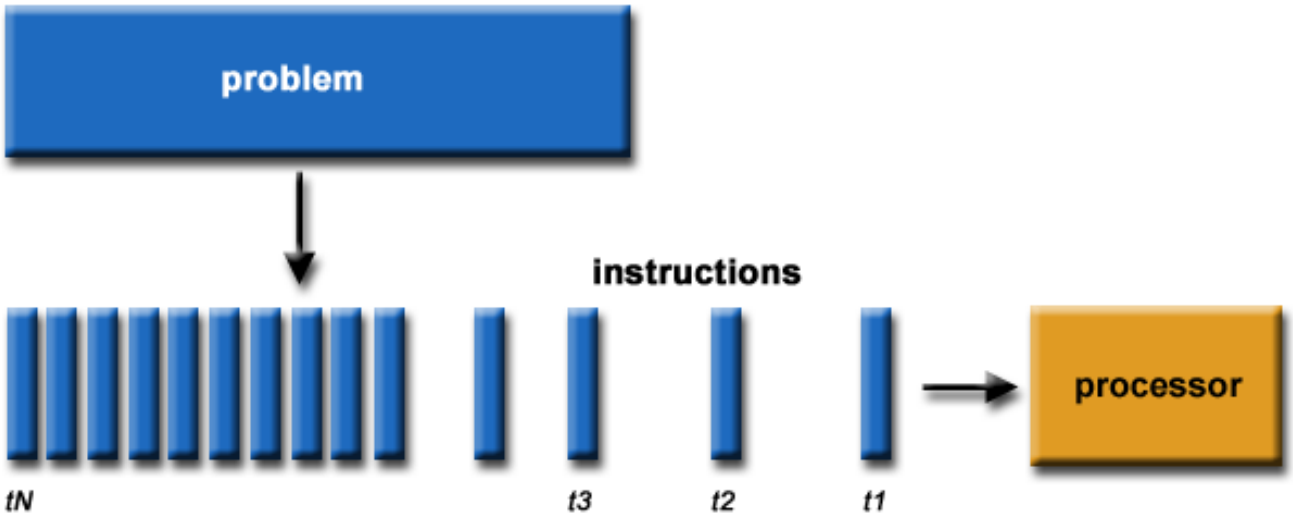
- Programming for clusters is (all) about exploiting massive amount of hardware parallelism!
- Program needs to be executed
- Processes and Threads are units of execution

Threads and Processes

- Abstraction provided by OS
- Process
  - Self-contained i.e. has its own private resources to execute/run programs. E.g. of a resource: memory.
  - Is an instance of a running program.
  - Have an illusion that *entire computer* is for itself.
- Thread
  - Belongs to a process. Share memory and other resources among threads of the same process.
  - Have an illusion that *entire processor* is for itself.
  - Can be considered as a subroutine in the main program

Clusters – How are they programmed?

- It's (all) about parallelism!
- Sequential Program - *single sequence of instructions - single-threaded.*



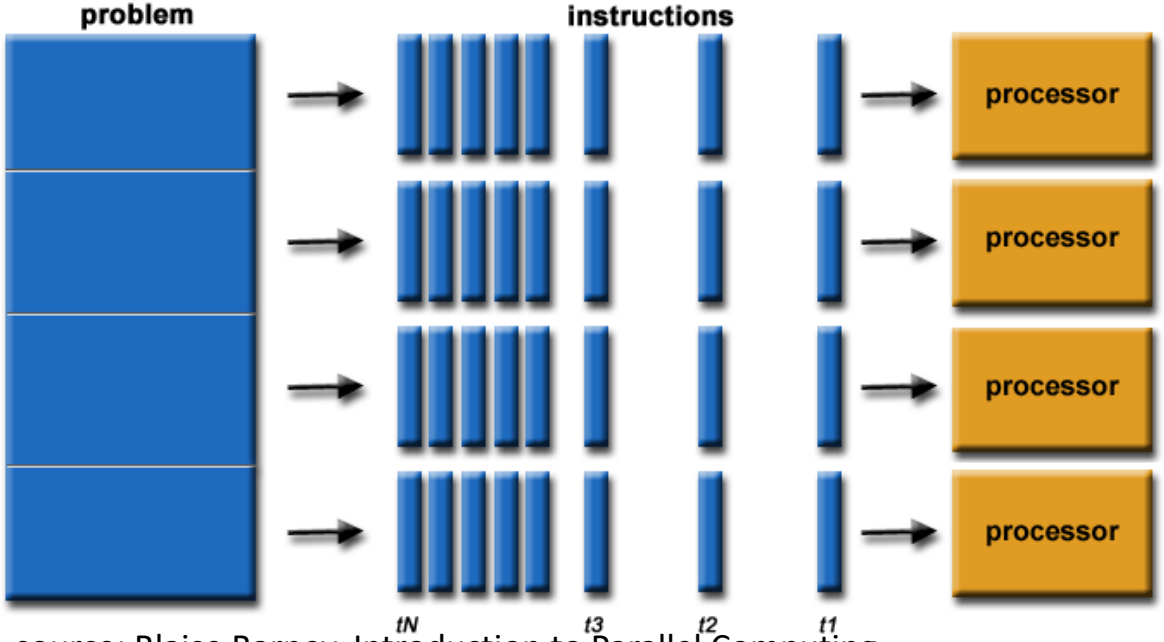
source: Blaise Barney, Introduction to Parallel Computing, https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

Clusters – How are they programmed?

- It's (all) about parallelism!
- Sequential Program - *single sequence of instructions - single-threaded.*
- Concurrent Program - *Multiple sequence of instructions executing concurrently.* Instructions from one sequence *may* communicate and interfere with other.
  - How are they (multi- threads) executing?
    - Multiprogramming – threads multiplexing their execution on **one processor**
    - Multiprocessing – threads multiplexing their execution on multiprocessor or **multicores**
    - Distributed Processing – processes multiplexing their executions on **multiple nodes**

Clusters – How are they programmed?

- Parallel Program – a *concurrent program* designed to execute on ***parallel hardware***
  - Multiple processors in a computer (multiprocessing),
  - Multiple computers in a network (distributed processing)



source: Blaise Barney, Introduction to Parallel Computing, https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

Parallel Hardware

- Flynn's taxonomy - categories of computing systems
  - Based on how processing elements see *instruction* and *data*

	Single Data (SD)	**Multiple Data (MD)**
Single Instruction (SI)	SISD	SIMD
Multiple Instruction (MI)	MISD	MIMD

Parallel Hardware

- Flynn's taxonomy - categories of computing systems
  - Based on how processing elements see *instruction* and *data*

	Single Data (SD)	**Multiple Data (MD)**
Single Instruction (SI)	SISD	SIMD
Multiple Instruction (MI)	MISD	MIMD

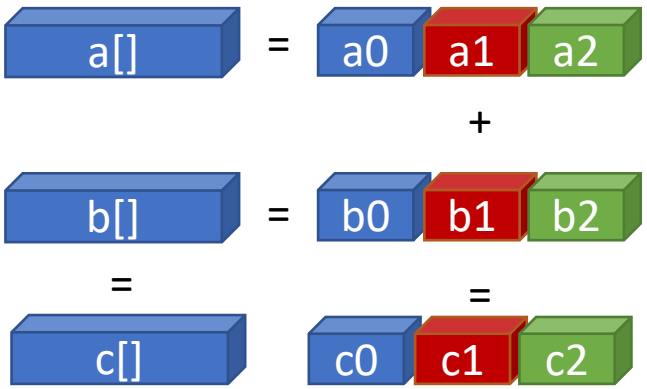
Clusters belong to this category

Parallel Hardware - SISD

- Single stream of instructions and single stream of data
- E.g. single-core computer (without “hyperthreading”)

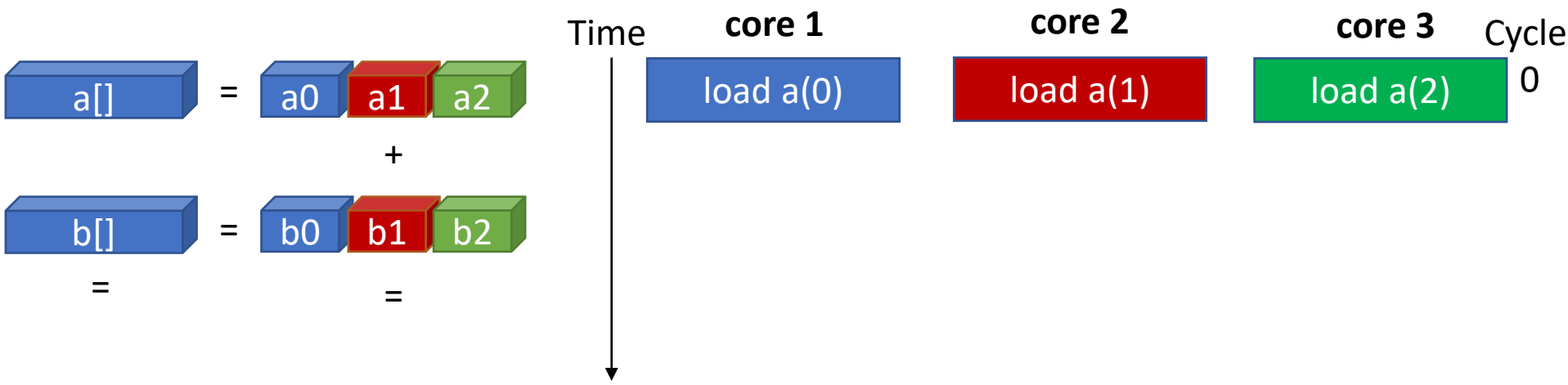
Parallel Hardware - SIMD

- Single stream of instructions and multiple streams of data



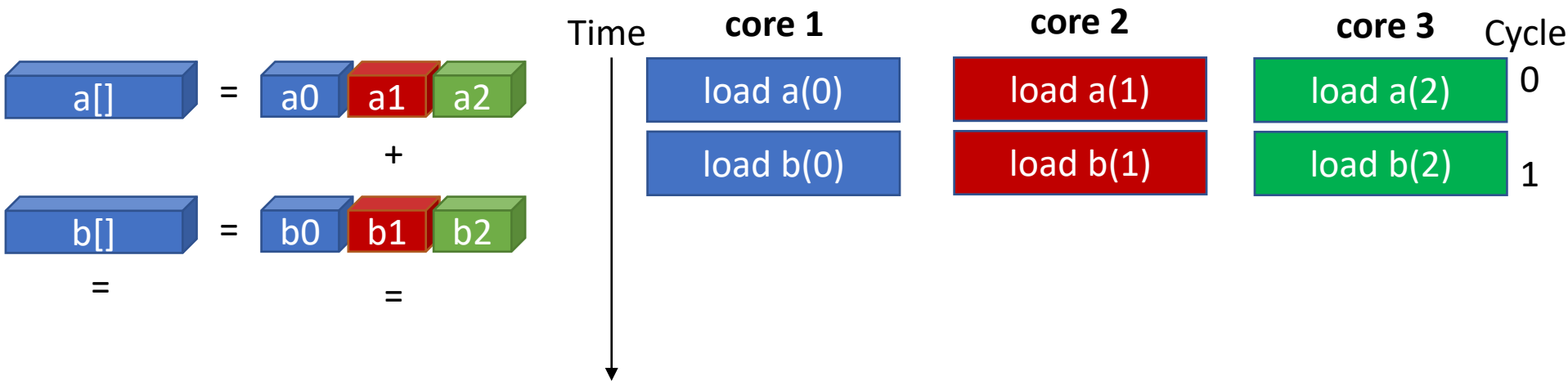
Parallel Hardware - SIMD

- Single stream of instructions and multiple streams of data



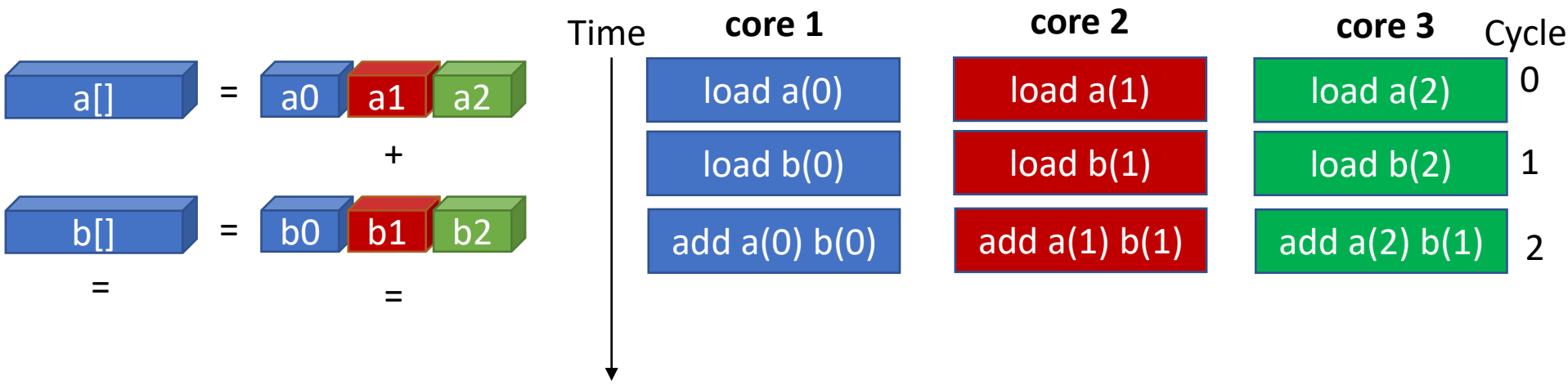
Parallel Hardware - SIMD

- Single stream of instructions and multiple streams of data



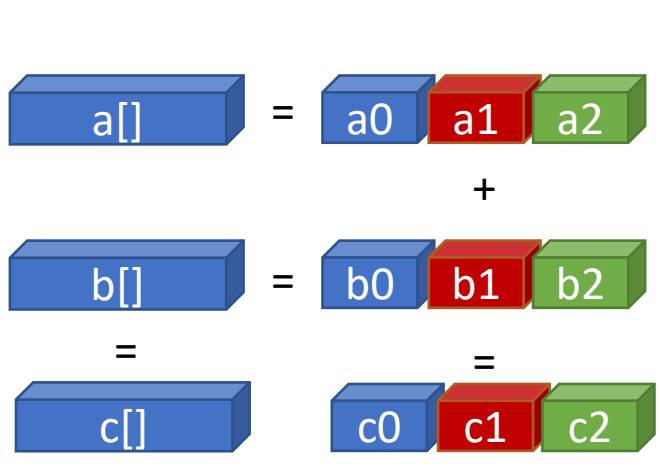
Parallel Hardware - SIMD

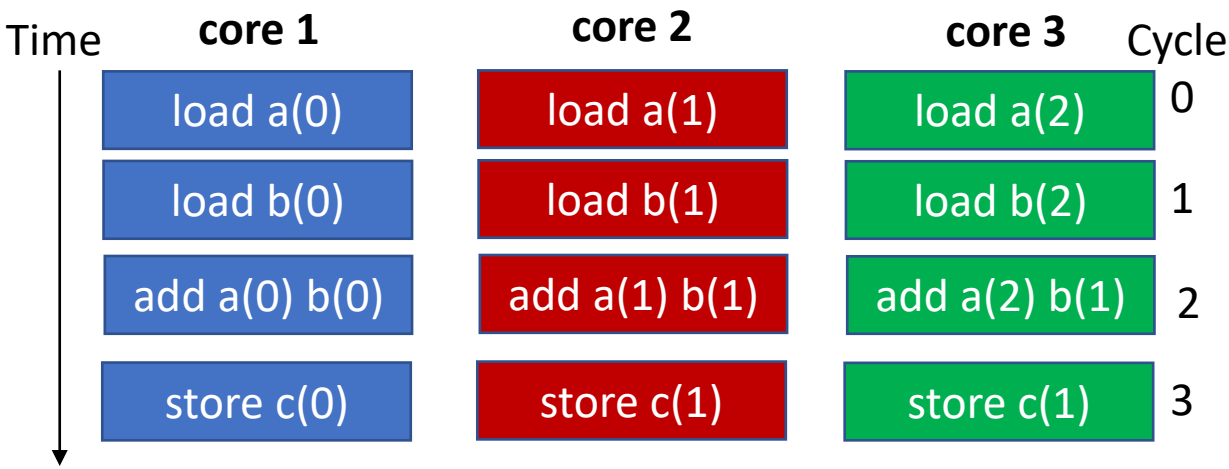
- Single stream of instructions and multiple streams of data



Parallel Hardware - SIMD

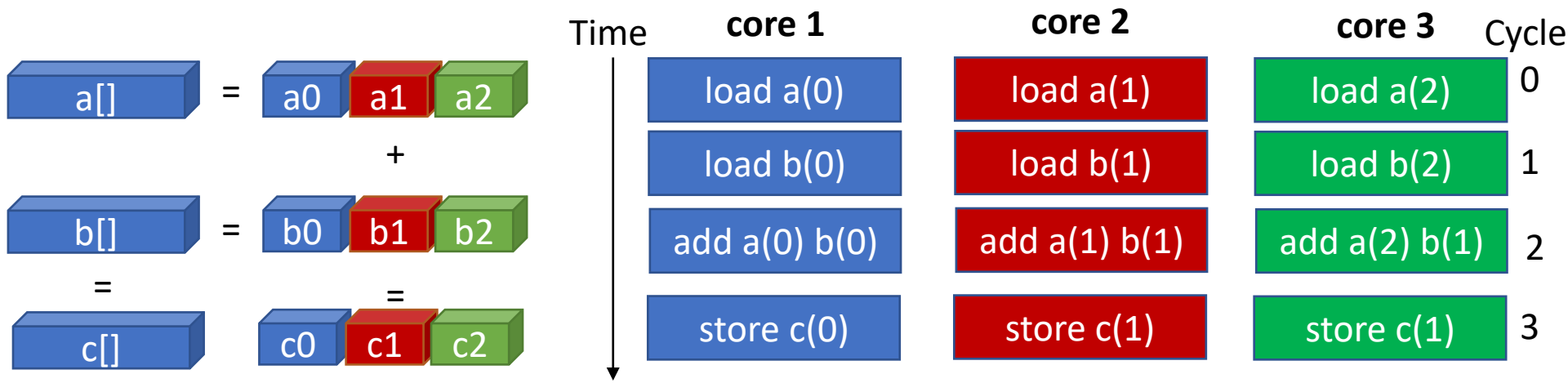
- Single stream of instructions and multiple streams of data





Parallel Hardware - SIMD

- Single stream of instructions and multiple streams of data



- All cores execute same instruction in a clock cycle. But operate on different data
- E.g. GPUs. Modern CPUs also have SIMD subcomponents.

Parallel Hardware - MIMD

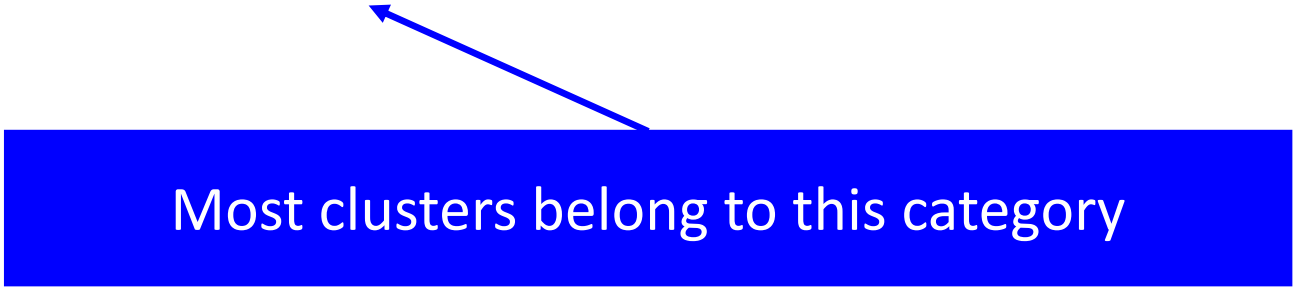
- Most common type of parallel computer
- Every core may be executing a different instruction and data stream
- The cores may or may not go lock-step (synchronous or asynchronous)
- E.g. Clusters, Supercomputers, multi-core PCs.
  - MIMD computers can have SIMD subcomponents

Parallel Hardware

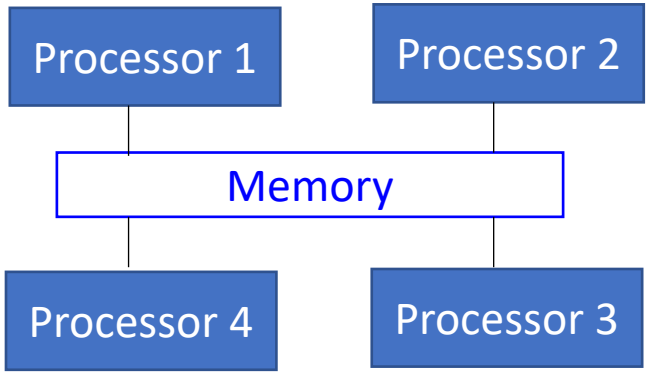
- Categorization based on how processing elements see *system memory*
  - Shared Memory
  - Distributed Memory
  - Distributed-Shared Memory

Parallel Hardware

- Categorization based on how processing elements see *system memory*
  - Shared Memory
  - Distributed Memory
  - Distributed-Shared Memory

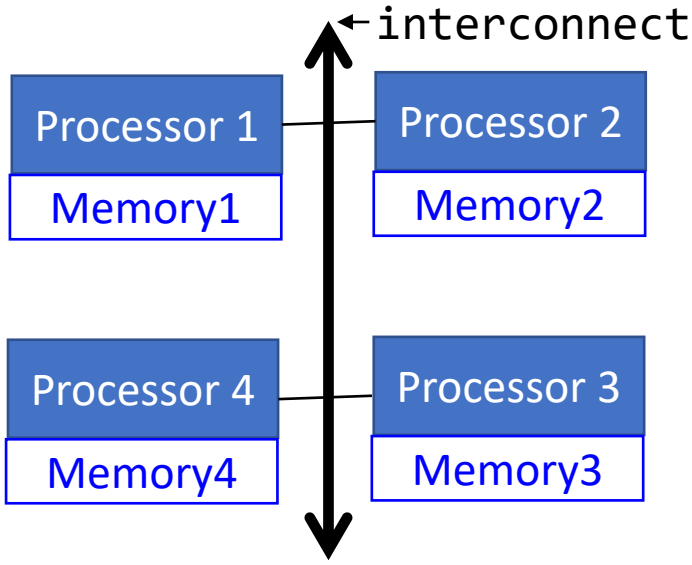


Parallel Hardware



Shared Memory Architecture

Distributed Memory Architecture



Parallelism

- Concurrency is the basis for writing parallel programs
  - E.g. Parallel Programs: C++ threads program, Open MP program, MPI program, Open ACC program, etc.
- Two common types of parallel programming patterns
  - Data parallel and Control / Task parallel

Data and Control Parallelism

- Threads executing the same function but with different data – ***data parallelism***
  - E.g. two construction workers laying bricks to build walls of different parts of a house
- Threads executing different functions – ***control parallelism***.
  - E.g. A carpenter getting a window frame ready while a mason is laying bricks in the wall

What is OpenMP

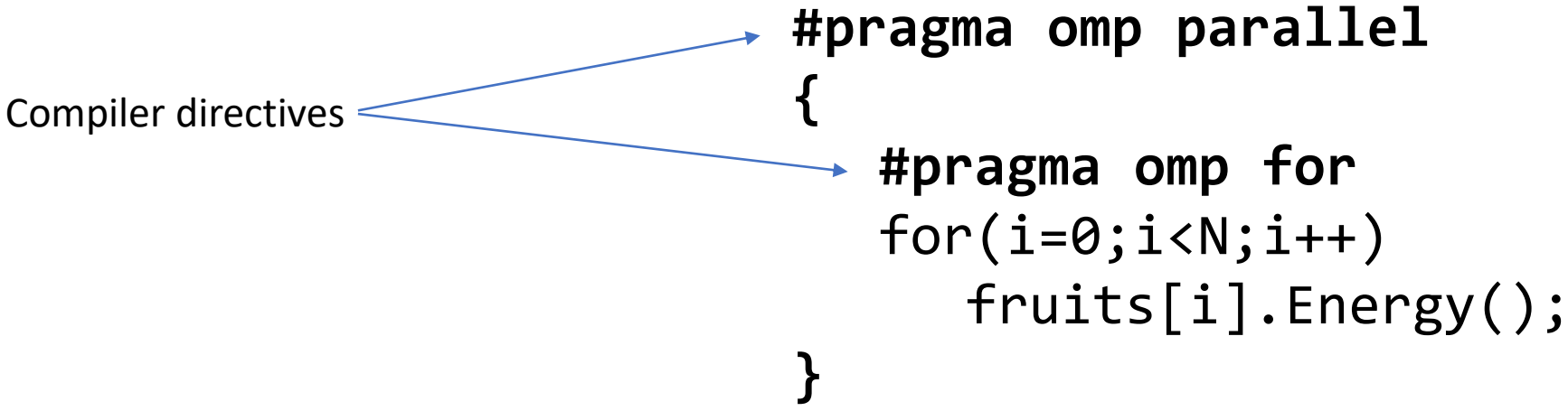
- An open standard for shared-memory programming in C/C++ and Fortran
- Supported by IBM, Intel, GNU and others
- Same program running on multiple threads each operating on different data (single program multiple data – SPMD)

Need for Open MP

- *Allows seamless scaling automatically* - add more processors and you will *not* have to rewrite the program to utilize available processors
  - (pthreads) programs are not performance portable

Open MP (multiprocessing) / OMP provides a scalable and portable alternative to data-parallel computing on *shared-memory architectures*

Programming in OpenMP



- `#pragma parallel`

  executes as many threads as there are processors

- `#pragma omp for`

  divides the whole work among available threads

Programming in OpenMP

```
#pragma omp parallel
{
   #pragma omp for
   for(i=0;i<N;i++)
      fruits[i].Energy();
}
```

- Example of loop parallelism
  - Common in scientific codes
- Programmer is still responsible for handling data races.

Programming in OpenMP

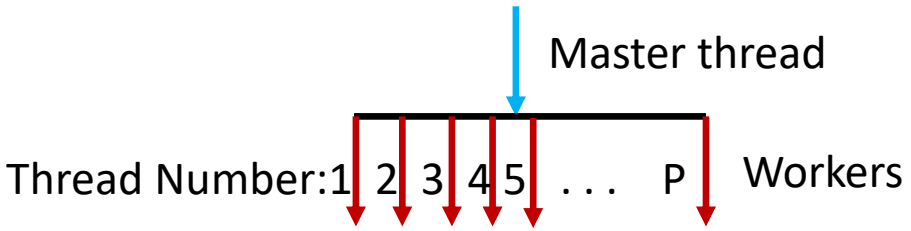
```
//code region 0
```

Master thread

- Execution begins with a master thread

Programming in OpenMP

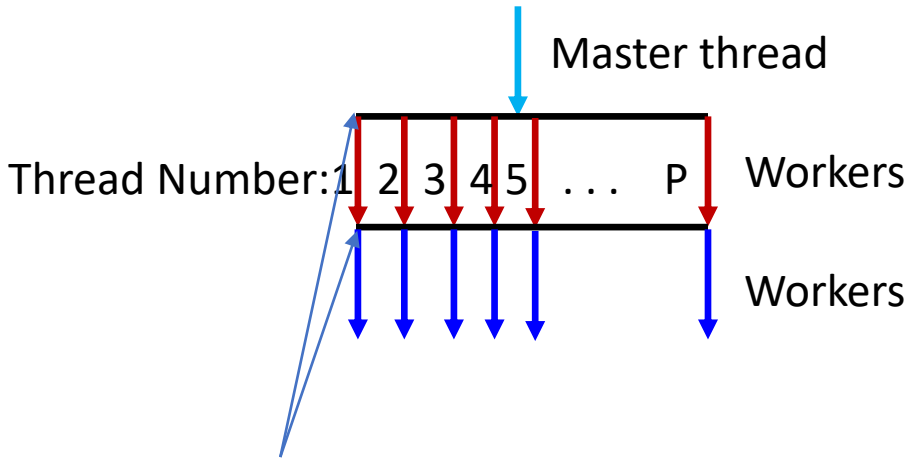
```
//code region 0
#pragma omp parallel
{
    //code region 1
}
```



- Master thread creates / forks worker threads (threads execute **code region 1**)

Programming in OpenMP

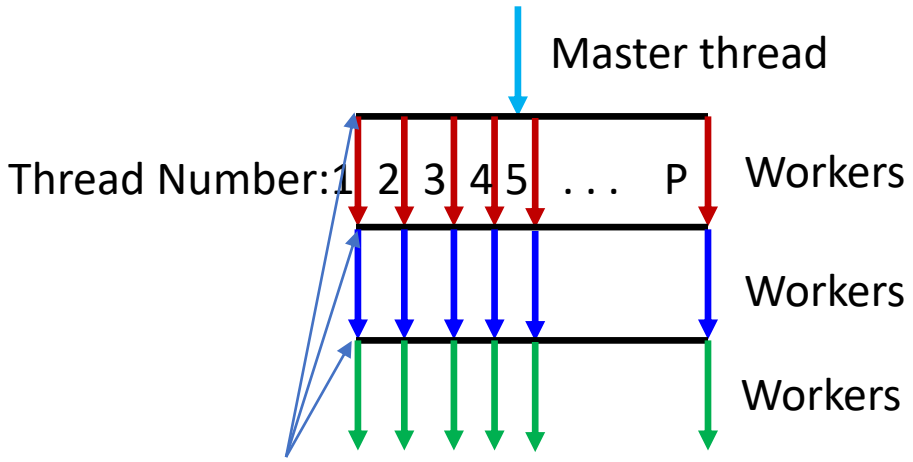
```
//code region 0
#pragma omp parallel
{
    //code region 1
    #pragma omp for
    for(i=0;i<N;i++) {
        //code region 2
    }
}
```



- Implicit barriers hinder worker threads' free run
  - Worker threads all begin to execute **code region 2** at the same time

Programming in OpenMP

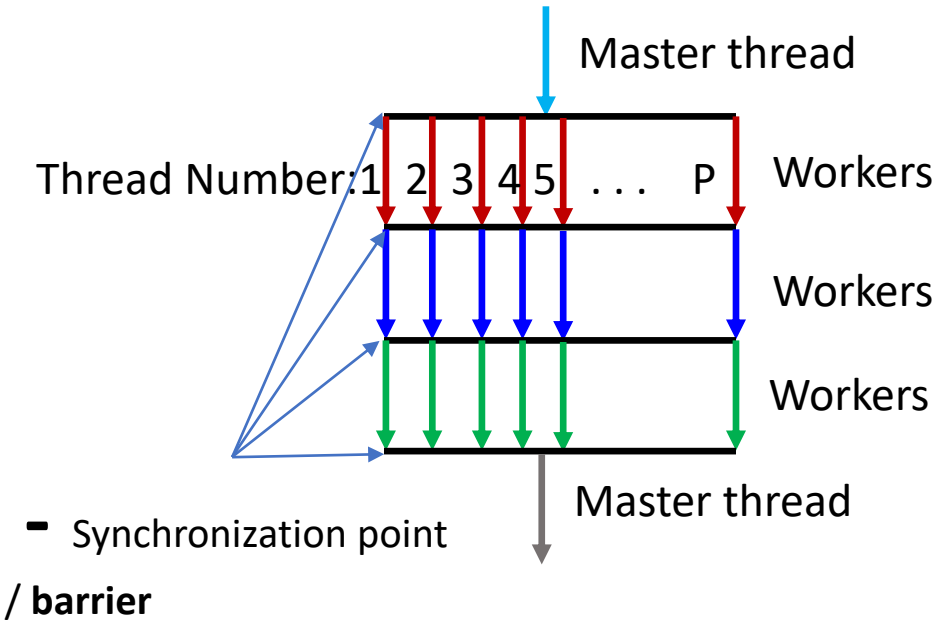
```
//code region 0
#pragma omp parallel
{
    //code region 1
    #pragma omp for
    for(i=0;i<N;i++) {
        //code region 2
    }
    //code region 3
}
```



- Worker threads cross another implicit barrier
  - Start executing **code region 3** all at the same time

Programming in OpenMP

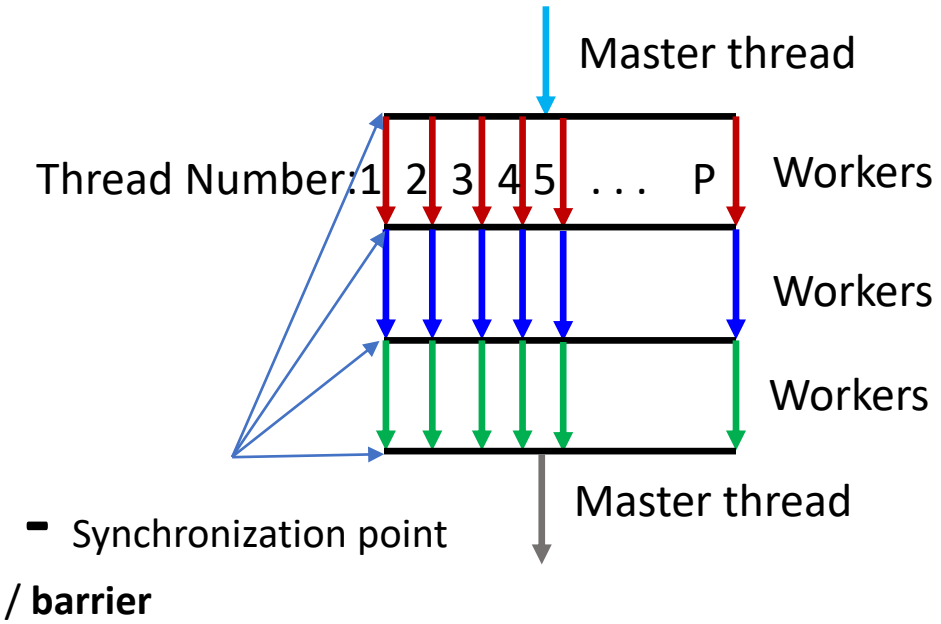
```
//code region 0
#pragma omp parallel
{
    //code region 1
    #pragma omp for
    for(i=0;i<N;i++) {
        //code region 2
    }
    //code region 3
}
//code region 4
```



- Worker threads join master thread
  - Master thread continues executing **code region 4**

Programming in OpenMP

```
//code region 0
#pragma omp parallel
{
    //code region 1
    #pragma omp for
    for(i=0;i<N;i++) {
        //code region 2
    }
    //code region 3
}
//code region 4
```



1. Execution begins with a master thread
2. Master thread creates / forks worker threads
3. Worker threads join master thread

fork / join parallelism

Programming in OpenMP

- How many workers? / threads?
  - = number of processors by default. Can also be set with `omp_set_num_threads(P)`
  - Can query the number of processors available on a machine with `omp_get_num_procs()`
  - Each thread has an ID returned by `omp_get_thread_num()`

Programming in OpenMP

- Example – *sum of array elements*

$$\text{sum} = \sum_{i=0}^{i=9} a_i$$

Programming in OpenMP

- Example – *sum of array elements*

sum is initialized to 0 by master

Worker 1

$$\sum_{i=0}^{i=4} a_i = sum1$$

sum = sum + sum1

Worker 2

$$\sum_{i=5}^{i=9} a_i = sum2$$

sum = sum + sum2

correct value of sum seen by master

Programming in OpenMP

- Example – *sum of array elements*

sum is initialized to 0 by master

Worker 1	Worker 2
$\sum_{i=0} a_i = sum1$	$\sum_{i=5} a_i = sum2$
sum = sum + sum1	sum = sum + sum2

sum is shared among workers. Only one worker should update at a time.

correct value of sum seen by master ?

Race Condition

- **Data race** happens when multiple threads access a data region **simultaneously** and at least one of those accesses is a **write**

Interleaved execution sequence

ATM1

```
if(balance + amount > 0)
// balance = 10, amount = -8

balance = balance + amount;
// balance = 2
```

ATM2

```
if(balance + amount > 0)
// balance = 10, amount = -8

// balance = 2, amount = -8
balance = balance + amount;
// balance = -6
```

time

Nondeterminism

- Total ordering in a sequential program
  - What instruction to execute next in the sequence of instructions is fixed
- Partial ordering in a concurrent program
  - The exact execution sequence is uncertain – *nondeterminsm*
- Modern computer architectures cause nondeterminism even in sequential programs!
  - depends on what behavior of program one visualizes.

Synchronization

- Threads need to coordinate their execution - Synchronization
- Synchronization helps avoid race conditions

Programming in OpenMP

- Example – *sum of array elements*

sum1 and sum2 are initialized to 0 by master

Worker 1

$$\sum_{i=0}^{i=4} a_i = sum1$$

Worker 2

$$\sum_{i=5}^{i=9} a_i = sum2$$

master computes sum = sum1+sum2

Programming in OpenMP

- Example – *sum of array elements*

sum1 and sum2 are initialized to 0 by master

Worker 1 Worker 2

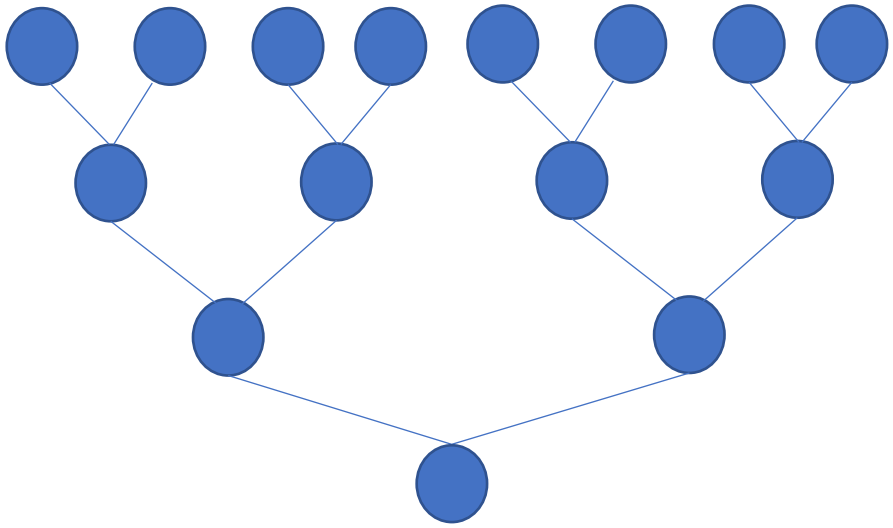
Workers don't race to update a shared variable. Can work independently. Master, in the end, does accumulating of partial sums and computing the sum.

$i=0$ $i=5$

master computes sum = sum1+sum2

Programming in OpenMP

- reductions - take something complex and reduce it to something simpler e.g. array to scalar
- Example: addition of array elements in parallel



Programming in OpenMP

- Example – *reductions (sum of array elements)*

sum is initialized to 0 by master

Worker 1

$$\sum_{i=0}^{i=4} a_i = sum1$$

sum = sum + sum1

Worker 2

$$\sum_{i=5}^{i=9} a_i = sum2$$

sum = sum + sum2

correct value of sum seen by master

Programming in OpenMP

- Example – *reductions (sum of array elements)*

sum is initialized to 0 by master

Worker 1 Worker 2

Reductions rid programmer of need to 'serialize' updates to sum

$\sum_{i=0} a_i = sum1$ $\sum_{i=5} a_i = sum2$

sum = sum + sum1 sum = sum + sum2

correct value of sum seen by master

Programming in OpenMP

- Example – *reductions (sum of array elements)*

```
#include<omp.h>
int main() {
    int total=0, a[NUM_SAMPLES];
    for(int i=0;i<NUM_SAMPLES;i++)
        a[i] = i;
    #pragma omp parallel
    {
        //#pragma omp for reduction(+: total)
        for(i=0;i<NUM_SAMPLES;i++)
            total = total + a[i];
    }
}
```

Programming in OpenMP

- Other operations supported in reductions:
  - +: addition
  - *: multiplication
  - |: bitwise OR
  - &: bitwise AND
  - ^: bitwise exclusive OR
  - ||: logical OR
  - &&: logical AND

  Note the *commutative nature* of these operations

Programming in OpenMP - Summary

- Open MP provides a way to specify what parts of program execute in parallel with one another
- How the work is distributed across different cores
- Whether to serialize (atomic) accesses to memory

All while providing *portable performance*

Distributed Memory Programming

- Program executes as a collection of processes

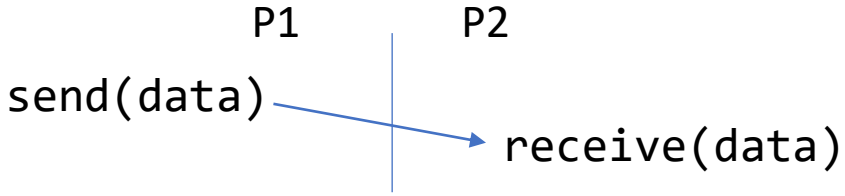
  Distributed Processing – processes multiplex their executions on multiple machines

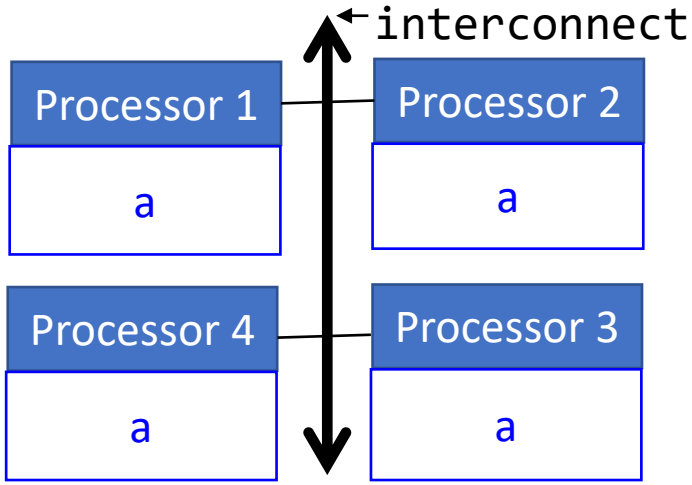
- Each process / processor has its own memory
  - Total memory available for an MPI program is the combined memory space of all processors
  - Exchanging data requires cooperation between two processors

Distributed Memory Programming

- Data exchange requires explicit communication:

Programmer must set up communication channels and exchange data





Value of a in P1 may be different from that in P2. **Why?**

1) P1 sends a copy of a to P2,

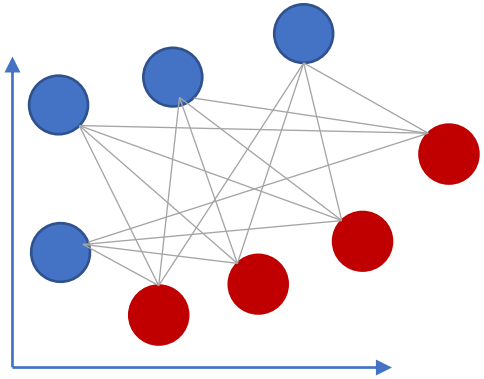
2) P2 receives the copy, stores it in its data region.

- Every data element must belong to one of the memory spaces Programmer must decide where to place data

Distributed Memory Programming

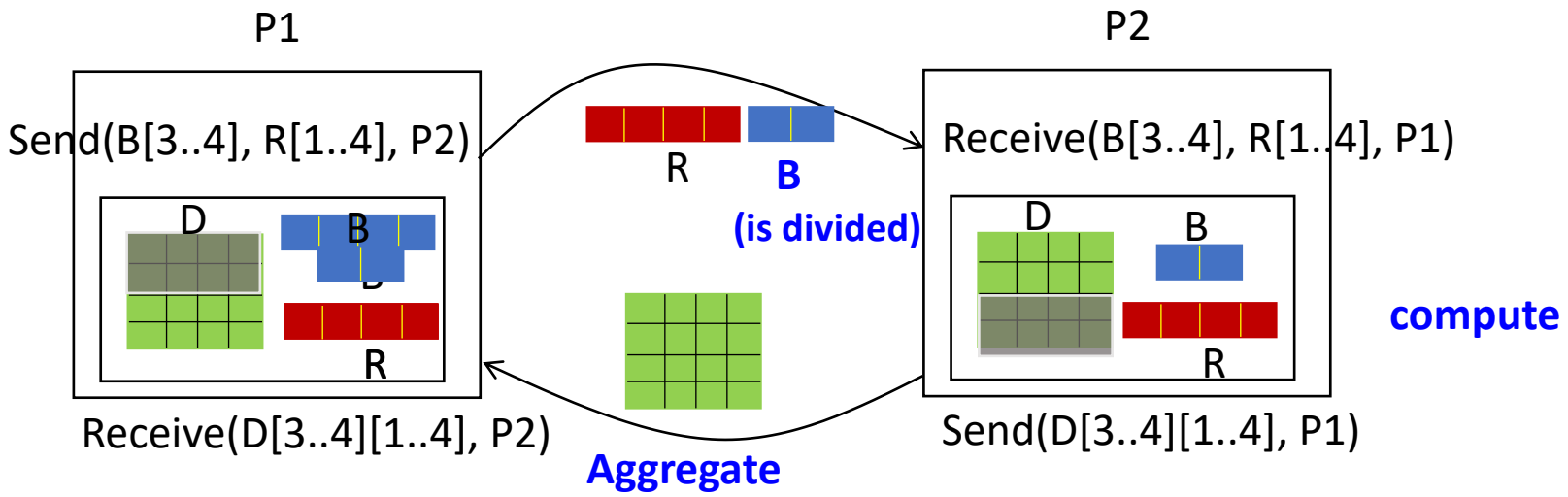
- More complex than shared-memory programming model
- Most programs are written in *Single Program Multiple Data* (SPMD) model
- Computing power and cost scaling is better than with shared memory – e.g. rack mounted blades
- E.g. weather forecasting, simulating dynamics of gases and fluids - where to put exhaust fans in a basement parking? can an ATV topple while wading through body of water?

Distributed Memory Programming - Example



Calculate the distance from each point in set Blue (B) to that in set Red (R) and store the result in set D

```
for(i=1 to 4)
    for(j=1 to 4)
        D[i][j] = distance(B[i],R[j])
```



Distributed Memory Programming - Example

```
        Processor P1
//initialize B, R, and D
Send(B[3..4], R[1..4], P2)
for(i=1 to 2)
    for(j=1 to 3)
        D[i][j] = distance(B[i], R[j])
Receive(D[3..4][1..4], P2)
```

```
        Processor P2
//initialize B, R, and D
Receive(B[3..4], R[1..4], P1)
for(i=3 to 4)
    for(j=1 to 3)
        D[i][j] = distance(B[i], R[j])
Send(D[3..4][1..4], P1)
```

- How is work divided among processors?
- What does it mean for send and receive to complete?
- How does a receiver interpret data that a sender sends?

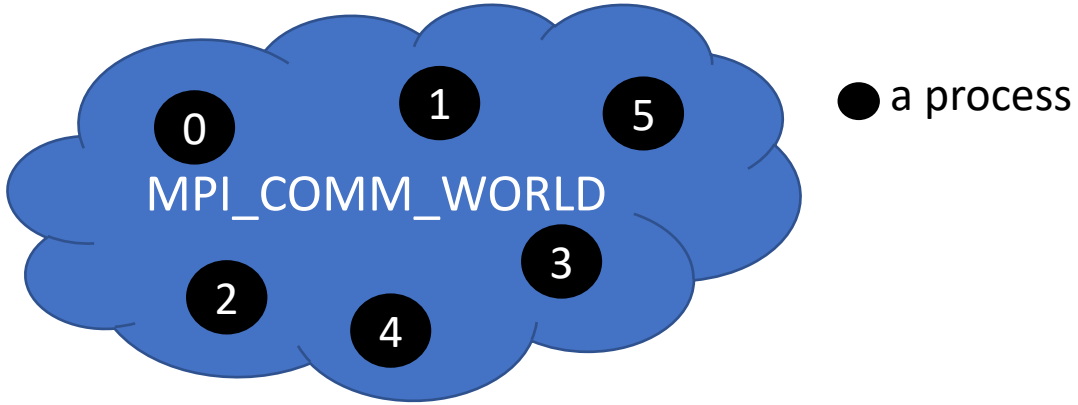
Converting to MPI Program

- Initialization and Termination

```
#include<mpi.h>

int main(int argc, char* argv[]) {
    MPI_Init(&argc,&argv); //initializes MPI Environment

    //all other code here

    MPI_Finalize(); //releases system resources
}
```

Converting to MPI Program

- Environment
  - 0,1,. . 5 – ranks / process numbers
  - MPI_COMM_WORLD – communicator / group of processes that are allowed to exchange messages



  - MPI_Init initializes the communicator.

Converting to MPI Program

- Rank and Size: obtaining process number and number of processes in the execution environment

```
#include<mpi.h>
int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Finalize();
}
```

Converting to MPI Program

- Divide the work and compute

```
#include<mpi.h>
int main(int argc, char* argv[]) {
    int rank, size;
    float d[NUMPOINTS][NUMPOINTS], b[NUMPOINTS], r[NUMPOINTS];
    //initialize b and r arrays from input
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    for(i=rank*NUMPOINTS/size; i<(rank*NUMPOINTS/size)+NUMPOINTS/size; i++)
        for(j=0;j<NUMPOINTS;j++)
            d[i][j] = distance(b[i], r[j]);
    MPI_Finalize();
}
```

Converting to MPI Program

- Aggregate the results at master – MPI_Send, MPI_Recv

```
#include<mpi.h>
int main(int argc, char* argv[]) {
    int rank, size;
    float d[NUMROWS][NUMROWS], b[NUMROWS], r[NUMROWS];
    //initialize b and r arrays from input
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    //compute d[i][j] as before
    if(rank !=0){
     for(int i=rank*NUMPOINTS/size;i<(rank*NUMPOINTS/size + NUMPOINTS/size);i++)
        MPI_Send(d[i], NUMPOINTS, MPI_FLOAT, 0, MY_MESSAGE_TAG, MPI_COMM_WORLD);
    else
     for(int i=NUMPOINTS/size;i<NUMPOINTS;i++)
        MPI_Recv(d[i], NUMPOINTS, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
    MPI_Finalize();
}
```

Converting to MPI Program

- Aggregate the results at master – MPI_Send, MPI_Recv

```
#include<mpi.h>
int main(int argc, char* argv[]) {
    int rank, size;
    float d[NUMROWS][NUMROWS], b[NUMROWS], r[NUMROWS];
    //initialize b and r arrays from input
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    //compute d[i][j] as before
    if(rank !=0){
     for(int i=rank*NUMPOINTS/size;i<(rank*NUMPOINTS/size + NUMPOINTS/size);i++)
        MPI_Send(d[i], NUMPOINTS, MPI_FLOAT, 0, MY_MESSAGE_TAG, MPI_COMM_WORLD);
    else
     for(int i=NUMPOINTS/size;i<NUMPOINTS;i++)
        MPI_Recv(d[i], NUMPOINTS, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
    MPI_Finalize();
}
```

Send buffer

Count of sent items

Data type of sent items

Destination process ID

Message ID

Recv buffer

Recv count

any Message ID

From any Source Process

Converting to MPI Program

- Aggregate the results at master (collectives) – MPI_Gather

```
#include<mpi.h>
int main(int argc, char* argv[]) {
    int rank, size;
    float d[NUMROWS][NUMROWS], tmpd[NUMROWS], b[NUMROWS], r[NUMROWS];
    //Initialize b and r arrays from input
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    //compute distance into tmpd[NUMROWS];

    MPI_Gather(tmpd, NUMROWS, MPI_FLOAT, d, NUMROWS, MPI_FLOAT, 0,
MPI_COMM_WORLD)
    MPI_Finalize();
}
```

Send buffer

Count of sent items

Data type of sent items

Source Process ID

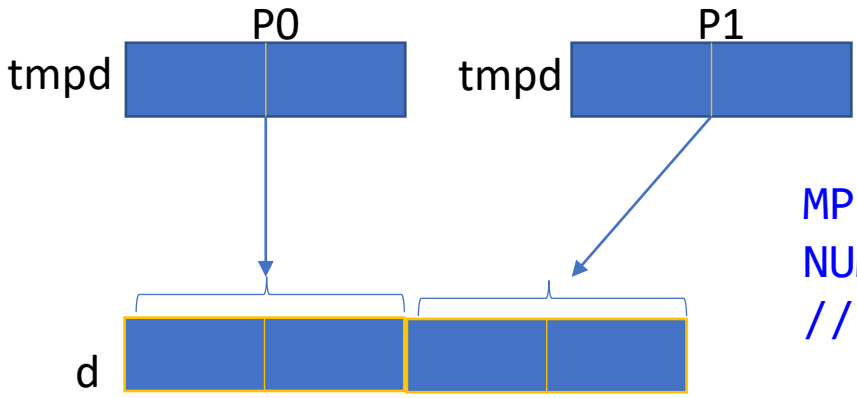
Recv buffer

Recv count

Converting to MPI Program

- collectives – MPI_Gather

```
MPI_Gather(sendbuf, sendcnt, send_type, recvbuf, recvcnt, recv_type,
           source_proc, MPI_COMM_WORLD)
```



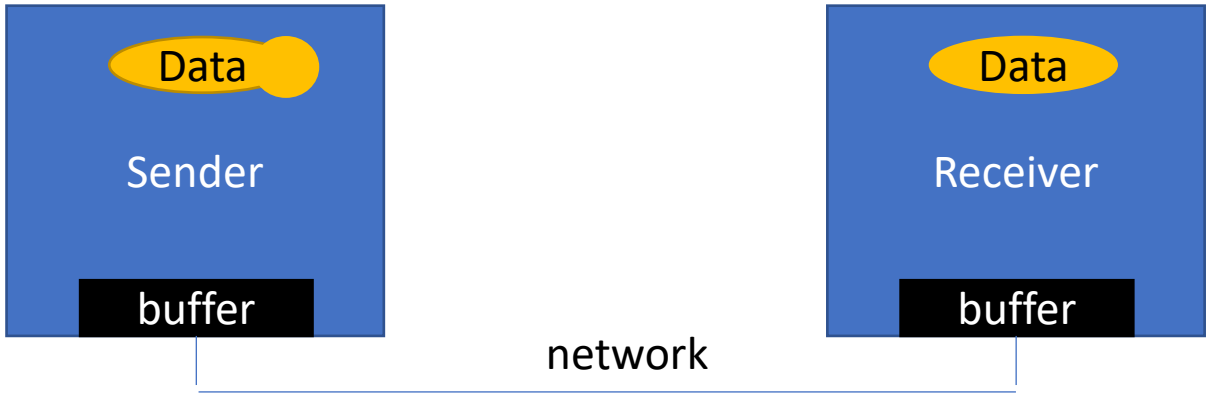

```
MPI_Gather(tmpd, NUMROWS, MPI_FLOAT, d,
NUMROWS, MPI_FLOAT, 0, MPI_COMM_WORLD)
//NUMROWS = 2, 2 processes, 2x2 matrix
```

MPI Programming – Point-to-Point

- In most MPI programs, communication is between a pair of processors.
  - Think other types of communication: Broadcast (one-to-all), Reduce(All-to-one), Scatter (one-to-several), Gather(several to one), All-to-All
- When is `send/receive` complete?
  - Synchronous / Asynchronous
  - Blocking / non-blocking
  - Buffered

MPI Programming – Point-to-Point

- Synchronous vs. Asynchronous
  - Synchronous: sender notified when message is received
  - Asynchronous: sender only knows that the message is sent



MPI Programming – Point-to-Point

- Blocking vs. Non-blocking
  - Blocking:
    - Sender waits until message is transmitted – buffer is empty
    - Receiver waits until message is received – buffer is full
  - Non-blocking
    - sender continues execution immediately after calling send

MPI Programming - Collectives

- MPI_Barrier – all processes wait at that line of code – a synchronization point
- MPI_Bcast – Broadcasting data from one process
- MPI_Scatter – Distribute data from master to all processes
- MPI_Gather – Collect data from all processes at master
- MPI_Allgather – Same as gather but all processes collect results
- MPI_Reduce – Aggregate results at master (recall `reduction` in OMP)
- MPI_Allreduce – Aggregate results at all processes

refer https://computing.llnl.gov/tutorials/mpi/ for API details

Performance

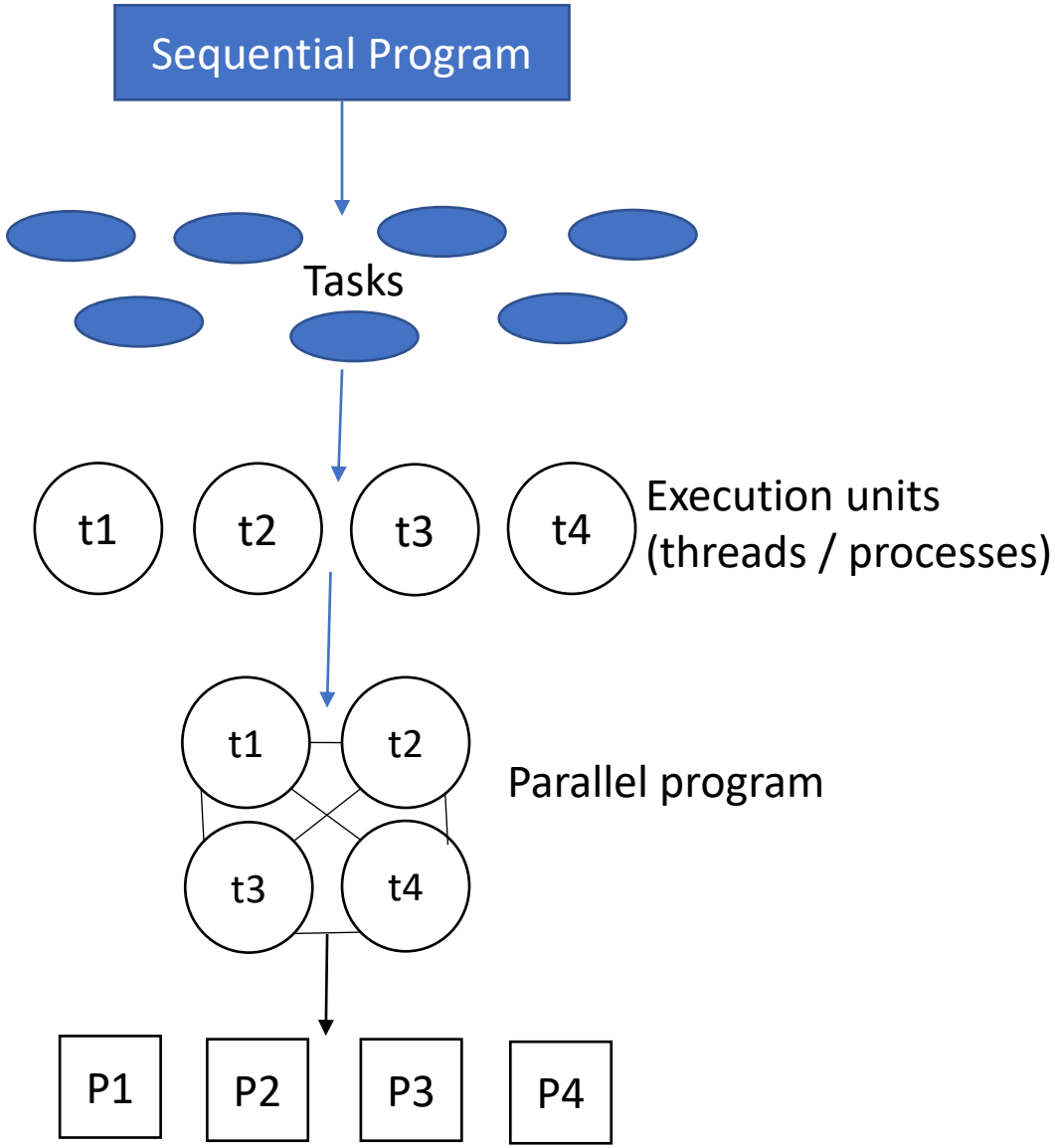
Depends on:

- How much parallelism is there in an algorithm
  - embarassingly parallel vs. sequential
- Granularity (what is the size of a partition when data is partitioned among available processors)
  - Coarse vs. fine grained
- Locality (Does data reside at a place close to where it is needed? is data used while it is in cache?) – impacts communication overhead (both inter-process and intra-process)
  - Spatial vs. Temporal

How to Parallelize

- Identify concurrency
- Decide how to exploit concurrency (whether to have coarse-grained partitions or fine-grained partitions) – comm. overhead vs. parallelism
  - Coarser => less communication but lesser opportunities for parallelizing
  - Finer => more communication but more opportunities for parallelizing
- Divide the computation and form tasks
  - Sometimes more / lesser tasks may become available at runtime

Recap: how to Parallelize



Decompose

Assign

Orchestrate communication

Map

Recap: Programming Models

- SPMD
- Loop Parallelism
- Master – Worker
- Fork - Join

Acknowledgements

- NSM Nodal Center for Training in HPC and AI, IIT Madras
- HPC-Operations Committee at IIT Dharwad
- Dr. Rupesh Nasre, IITM
- Dr. UnniKrishnan, IITPkd
- https://www.rcac.purdue.edu/
- https://top500.org
- many more..