Introduction to using a High-Performance Computing cluster

Mike Smith

Connecting to our cluster

- Connect using

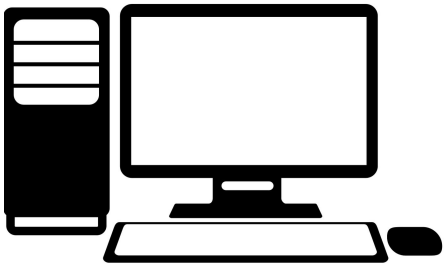
```
ssh -X testN@54.187.71.37
```

- Replace “N” with the number of your workstation e.g. `test10`
- Password is `SoftwareC`
- Hosted by Amazon, IP address will change when we shutdown

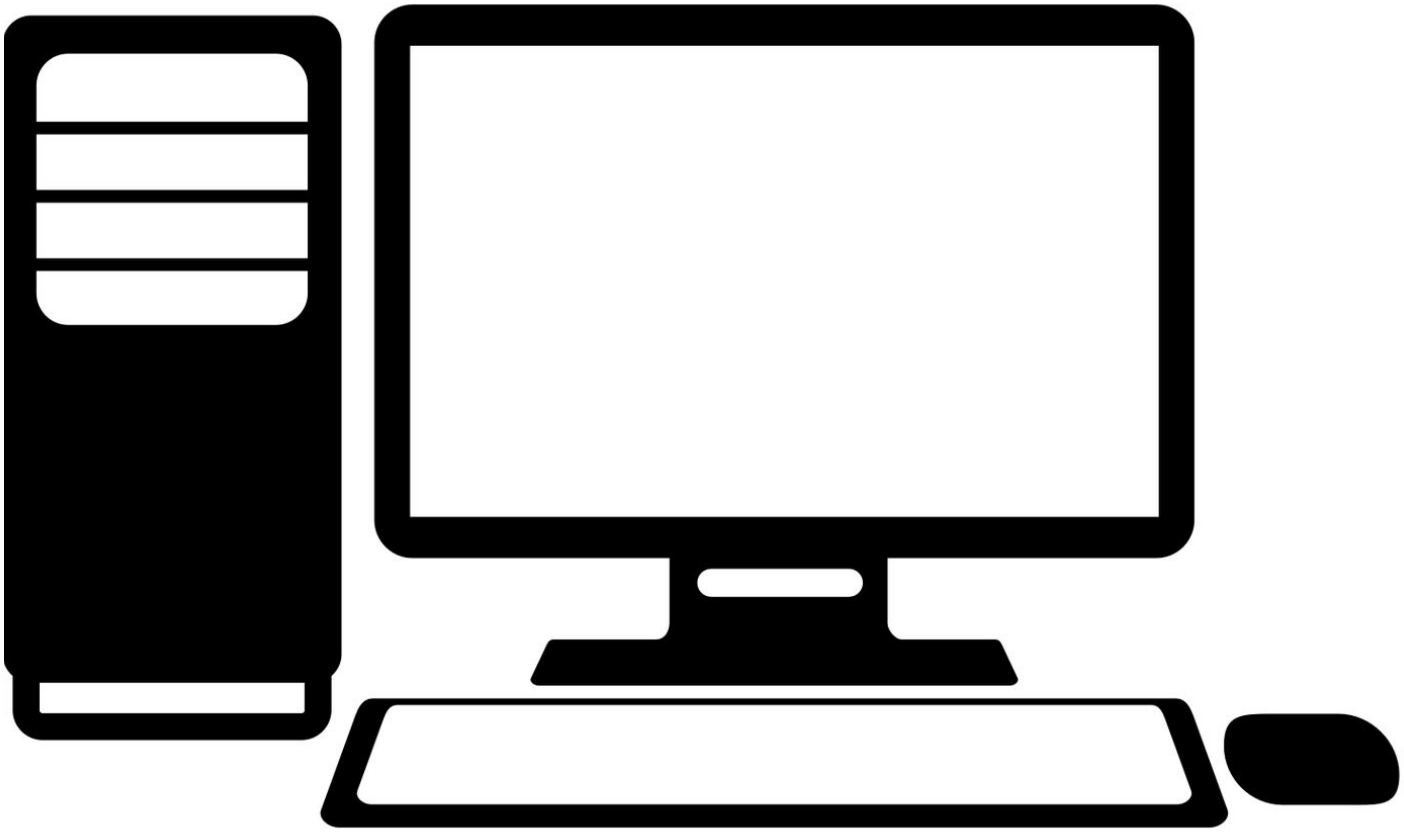
When is HPC useful?

- When you realise your standard computer is too small or too slow for your data
  - Compute Intensive: Task requiring a large amount of computation
    - e.g. more rigourous sequence alignment
  - Memory Intensive: Task requiring a large amount of memory
    - e.g. scaling up from bacteria to human genome
  - Data Intensive: Task involves operating on a large amount of data
    - e.g. 50 human genomes

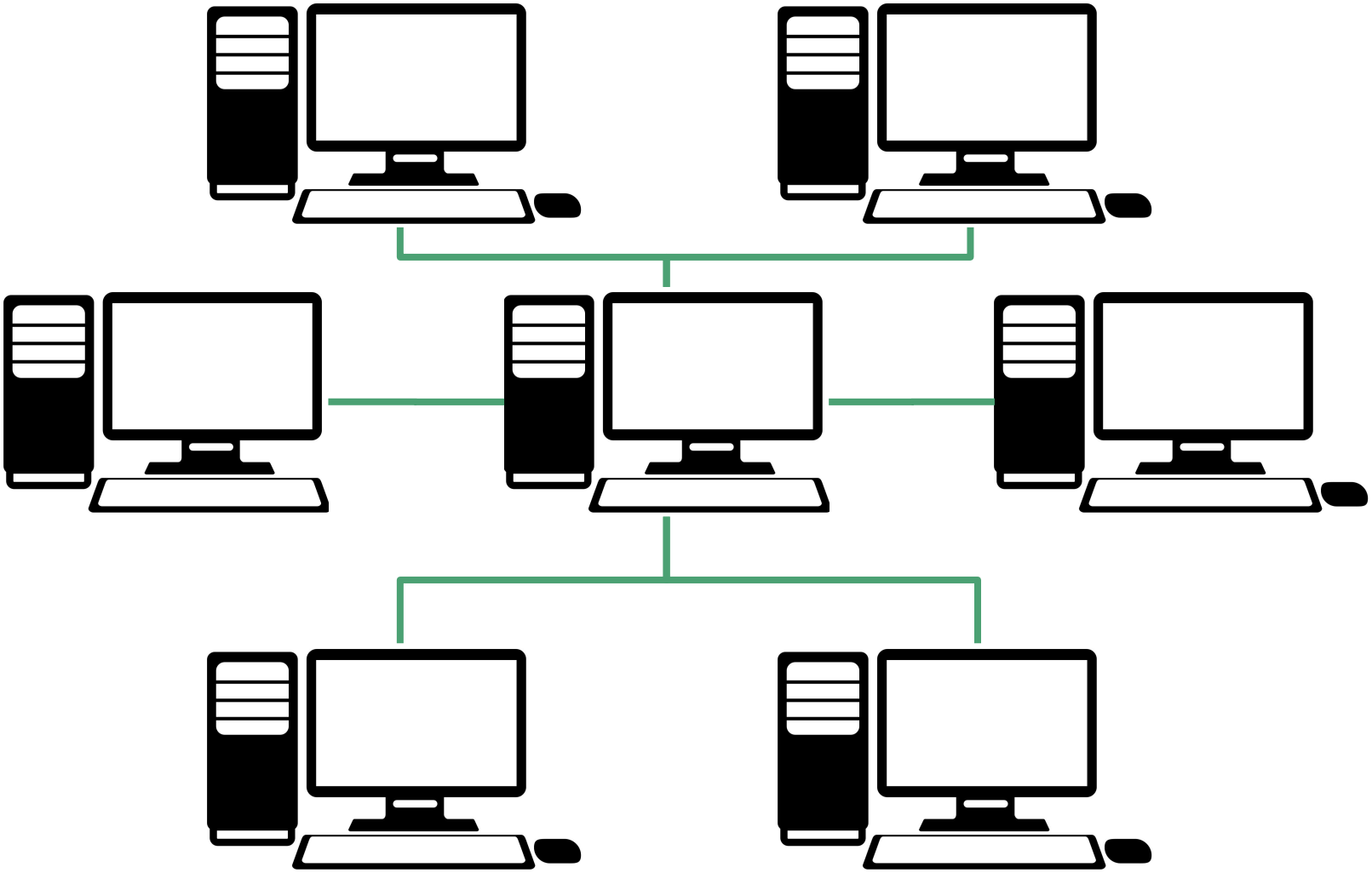
Types of Cluster - Shared Memory



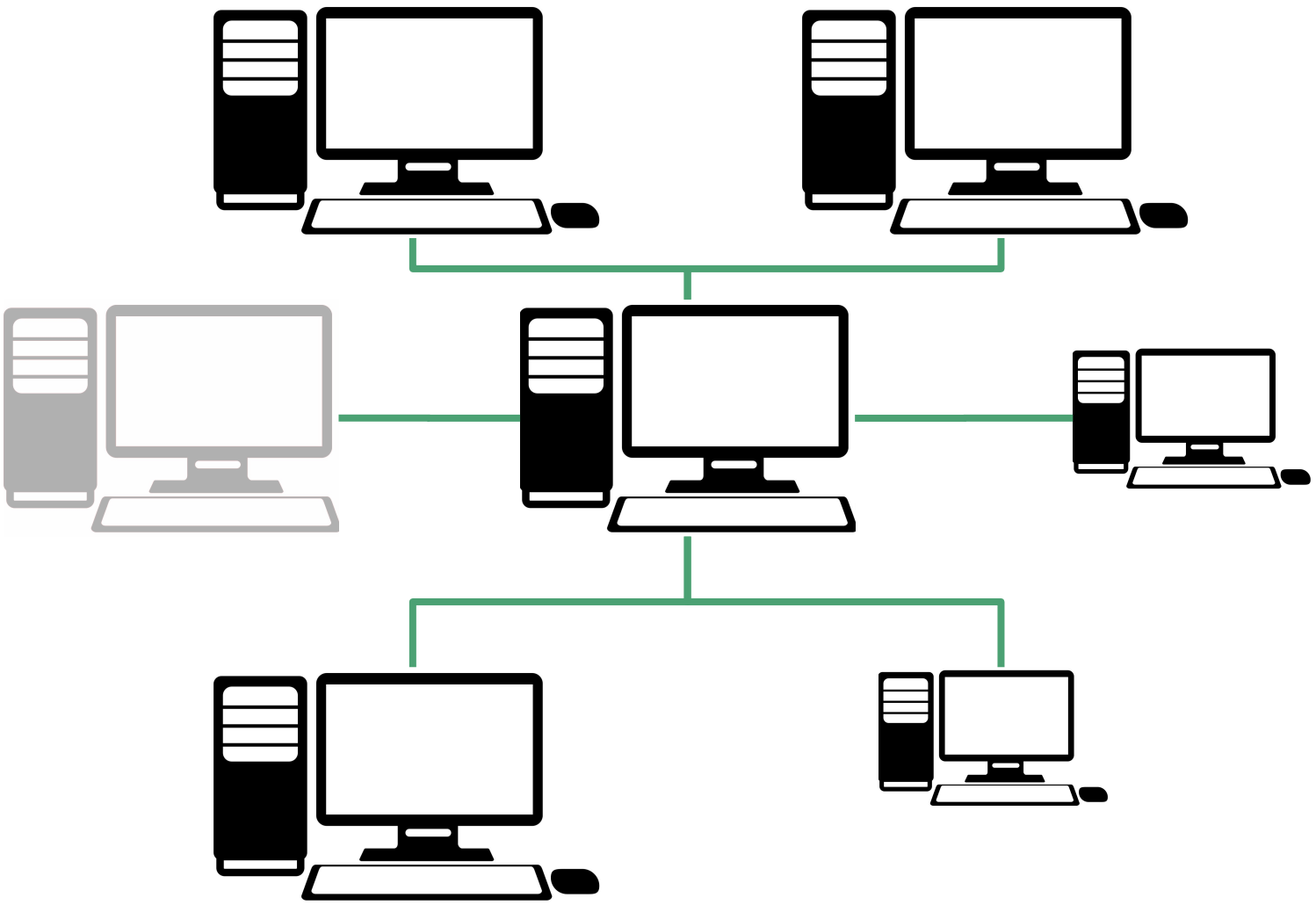
Types of Cluster - Shared Memory



Types of Cluster - Distributed Memory



Types of Cluster - Distributed Memory



How do we work with a distributed cluster?

- Typically interact with a single ‘Master’ node
- A job scheduler manages where and when tasks are run
  - There are many options available e.g. LSF, Openlava, SLURM, Condor, Univa Grid Engine
- Matches job requirements with available resources
- If no slots are available a job will wait until resources are available

Our example cluster

- Consists of three nodes:
  - `ip-172-31-0-10` - 1 core, 1GB RAM
  - `ip-172-31-0-20` - 8 cores, 32GB RAM
  - `ip-172-31-0-30` - 4 cores, 16GB RAM
- Not enough resources for us all to run programs simultaneously
- Clusters are about sharing!

Example python program

- Program should be present in your home directory
- Takes two arguments
  - `-t` Time to wait in seconds
  - `-l` Length of list to create (don't go over 1,000,000 !)

```
python ~/hpc_example.py -t 10 -l 100
```

- Prints arguments to screen ⇨ creates list ⇨ waits ⇨ prints memory usage ⇨ exits

(A few) Openlava Commands

- `lshosts` - shows the makeup of your cluster

- `bsub` - submits a job to the cluster

```
bsub "python ~/hpc_example.py -t 60 -l 100"
```

- `bjobs` - lists current jobs (by default only yours are shown)
- `bhosts` - lists how many jobs are running on each node

Getting some feedback on our jobs

- Don’t want to type `bjobs` to see if things are still running
- Also have no idea if it actually worked successfully
- Use `bjobs -o “output.txt”`
- We also need to use `-f` to copy files back to the master node

```
bsub -o "output.txt" \
     -f "output.txt < output.txt" \
     "python ~/hpc_example.py -t 60 -l 100"
```

Try creating a larger list

```
bsub -o "output.txt" \
     -f "output.txt < output.txt" \
     "python ~/hpc_example.py -t 60 -l 200000"
```

- Now look in "output.txt" to see the outcome
- Should see ‘Exited with exit code 1.’

Requesting Additional Resources

- Sharing resources between users is a key function of the job scheduler
- Jobs are killed if they try to use more than their allocated share

Requesting Additional Resources

- Sharing resources between users is a key function of the job scheduler
- Jobs are killed if they try to use more than their allocated share
- We can raise this limit using `-M 20000`

```
bsub -M 20000 \
     -o "output.txt" \
     -f "output.txt < output.txt" \
     "python ~/hpc_example.py -t 60 -l 200000"
```

Openlava memory requirements are actually a little more subtle

- `-M 20000` is a promise not to use more than 20,000KB limit
- `-R "rusage[mem=20]"` specifies you need 20MB and reserves it exclusively for your job

```
bsub -M 20000 \
     -R "rusage[mem=20]"
     -o "output.txt" \
     -f "output.txt < output.txt" \
     "python ~/hpc_example.py -t 60 -l 200000"
```

Try reserving a LARGE amount of memory

```
bsub -M 20000 \
     -R "rusage[mem=14000]" \
     -o "output.txt" \
     -f "output.txt < output.txt" \
     "python ~/hpc_example.py -t 60 -l 200000"
```

- Look at the running jobs with `bjobs -u all`
- Only a small number of jobs will be allowed to run simultaneously

Understanding the compute requirements of your task is key to effectively using a HPC cluster

- Ask for too much
  - Job will wait for a long time necessarily
  - Reserve resources you don’t need
- Ask for too little
  - Job may be killed without finishing
  - It’s easy to ‘cheat’ the system!
  - You start using resources you haven’t asked for, potentially slowing things down for everyone
- Run tests on a subset
- Some programs let you specify resource usage, so read the manual

Interactive jobs

- Sometimes we want to interact with a job
  - e.g. if we're testing code works
- This can be done with `bsub -Is`
- All other parameters can also be used as before

```
bsub -M 20000 \
     -Is \
     "sh"
```

Job dependencies

- We can give our jobs names
- Then we can make part 2 run only when part 1 was finished

```
bsub -J "job_part1" \
      -o "output.txt" \
      -f "output.txt < output.txt" \
      "python ~/hpc_example.py -t 60 -l 200"

bsub -J "job_part2" -w "done(job_part1)" \
      -o "output.txt" \
      -f "output.txt < output.txt" \
      "python ~/hpc_example.py -t 60 -l 200"
```

- IME this can be temperamental with LSF

Things we haven’t covered

- We have discussed only memory, jobs can have many more resource requirements
  - In particular the number of cores / threads you want to use
- Submission files
- Job checkpoints, suspension and resumption
- Executing more complex parallel programs
- …

Questions?