How to Run Parallel Jobs Efficiently

Shao-Ching Huang

High Performance Computing Group
UCLA Institute for Digital Research and Education

May 9, 2013

The big picture: running parallel jobs on Hoffman2

2-step process

1. Request resources

- Resources are described by job parameters
- e.g. CPU type, memory size, time limit, node distribution

The submitted job goes into the queue, pending. Once the requested resources become available, the job is dispatched to the resources (compute nodes).

2. Run the job using the granted resources

- Map threads/processes to CPUs
- In many cases the default works just fine
- You can customize it to your specific needs

Outline

- Understanding the environment
  - Grid Engine (resource manager + scheduler)
  - OpenMP
  - MPI
  - Modules
- Computational bottlenecks
- Putting things together
- Summary and suggestions

A few words

- On Hoffman2 cluster, many users use the convenient `mpi.q` (or similar) command to construct job scripts
- The purpose of this class is to explain some of the mechanisms ”behind the scene”
- You may use what you learn here to modify the job script generated by `mpi.q`, or write customized job script(s)
- Let me know immediately if you do not understand

Computational bottlenecks

- Memory bound
  - Computational speed depends on how fast data is fed to CPU
  - Typical examples: mesh-based PDE solvers
- CPU bound
  - Computational speed depends on how fast CPUs crunch numbers
  - Typical examples: particle methods, some high-order methods
- I/O bound
  - Computational speeds depends on how fast files are read or written

In a real simulation, different types of bottlenecks may dominate at different stages of the computation.

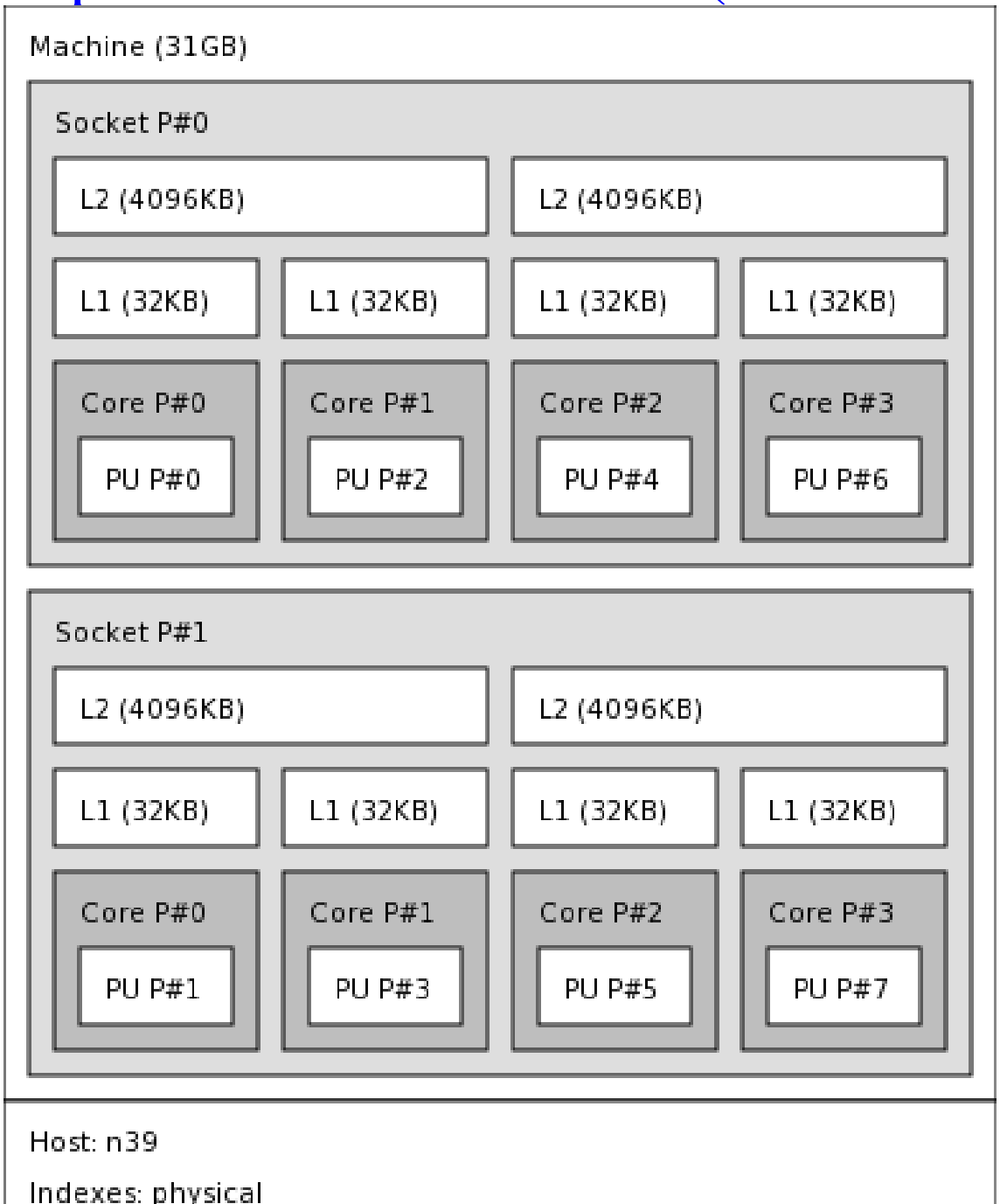
OpenMP Basics

- OpenMP is a standard API for shared memory thread programming
  - Compiler directives
  - Library
  - Environment variables
- All threads are on the same compute node
- All threads have access to the same memory space
- Directly supported by compilers
- `http://www.openmp.org`

MPI Basics

- MPI is a standard API for message-passing programming
- Run on virtually any computer (from laptop to supercomputers)
- distributed memory
  - Each MPI process has its own memory space
  - data needs to be explicitly "copied" from one process to anther when needed
- Each process is numbered by a "rank", numbered from 0 to n-1 for n MPI processes
- Two popular MPI "implementations" (same API):
  - OpenMPI `openmpi.org`
  - MPICH (+ derivatives, e.g. MVAPICH2) `mpich.org`

Example: dual 4-core CPUs (8 cores in total)



Effects of Thread Affinity

Set up

- Test problem: conjugate gradient solver (in a CFD solver)
- OpenMP + Intel compiler, dual 4-core Intel E5335 CPUs
- Thread placement controlled by `KMP_AFFINITY` (`GOMP_CPU_AFFINITY` if using gcc)

Results

- Case 1 : one thread per socket
  thread 0 $\rightarrow$ core 0, thread 1 $\rightarrow$ core 1
  timing: 75 seconds
- Case 2 : two threads on one socket (the other socket is "idle")
  thread 0 $\rightarrow$ core 0, thread 1 $\rightarrow$ core 2
  timing: 102 seconds

Example: dual 8-core CPUs (16 cores in total)

Mapping MPI ranks to nodes – host file

- Host file (aka machine file)
  - a text file listing all the machines (+ some details) how MPI ranks will be mapped to at run time
- Typical syntax

```
mpirun -n 4 -hostfile my_host a.out
mpirun -n 4 -machinefile my_host a.out
```

- Different MPI implementations (i.e. MPICH) have different (but similiar) syntax; check the manual

OpenMPI host file example (e.g. `my_host`)

```
n1 slots=2
n2 slots=4
# this is a comment line
n3 slots=2
```

Testing MPI Host File (OpenMPI)

```
$ cat my_host
n1 slots=2
n2 slots=4
$ mpirun -n 6 hostname
sch@n1:~ $ mpirun -n 6  hostname
n1
n1
n1
n1
n1
n1
sch@n1:~ $ mpirun -n 6 -hostfile my_host hostname
n1
n1
n2
n2
n2
n2
```

OpenMPI: core distribution across nodes

`--byslot`

- To maximum adjacent ranks on the same node
- For example, ranks 0–3 on the first node, rank 4–6 on the second node, etc.

`--bynode`

- Put one rank per node in a round robin fasion
- For example, rank 0 on the first node, rank 1 on the second node, etc.

Other MPI implementations have similiar features (but under different option names).

OpenMPI: Infiniband or Ethernet

Using Infiniband for communication

```
mpirun --mca btl ^tcp <other options>
```

Using Ethernet for communication

```
mpirun --mca btl ^openib <other options>
```

^ means “exclude”.

Hoffman2 Cluster

- 1000+ compute nodes, 9000+ CPU cores
- Any UCLA-affiliated person can apply for an account
- Shared cluster model
  - Many compute nodes are contributed by sponsoring PI/groups
  - Nodes are purchased and added at different times
    (cpu speed and memory size are different on different nodes!)
  - Priority job scheduling for contributors on the nodes they "own"
  - Idle cycles are utilized by others
- Optimized for batch processing – large number of jobs submitted by a wide range of users
  - High-throughput computing (lots of independent jobs)
  - Massively parallel computing (multiple-cpu/node jobs)
- Interactive use possible
  - debugging, testing, running GUI-based software, etc.
- `http://www.idre.ucla.edu/hoffman2`

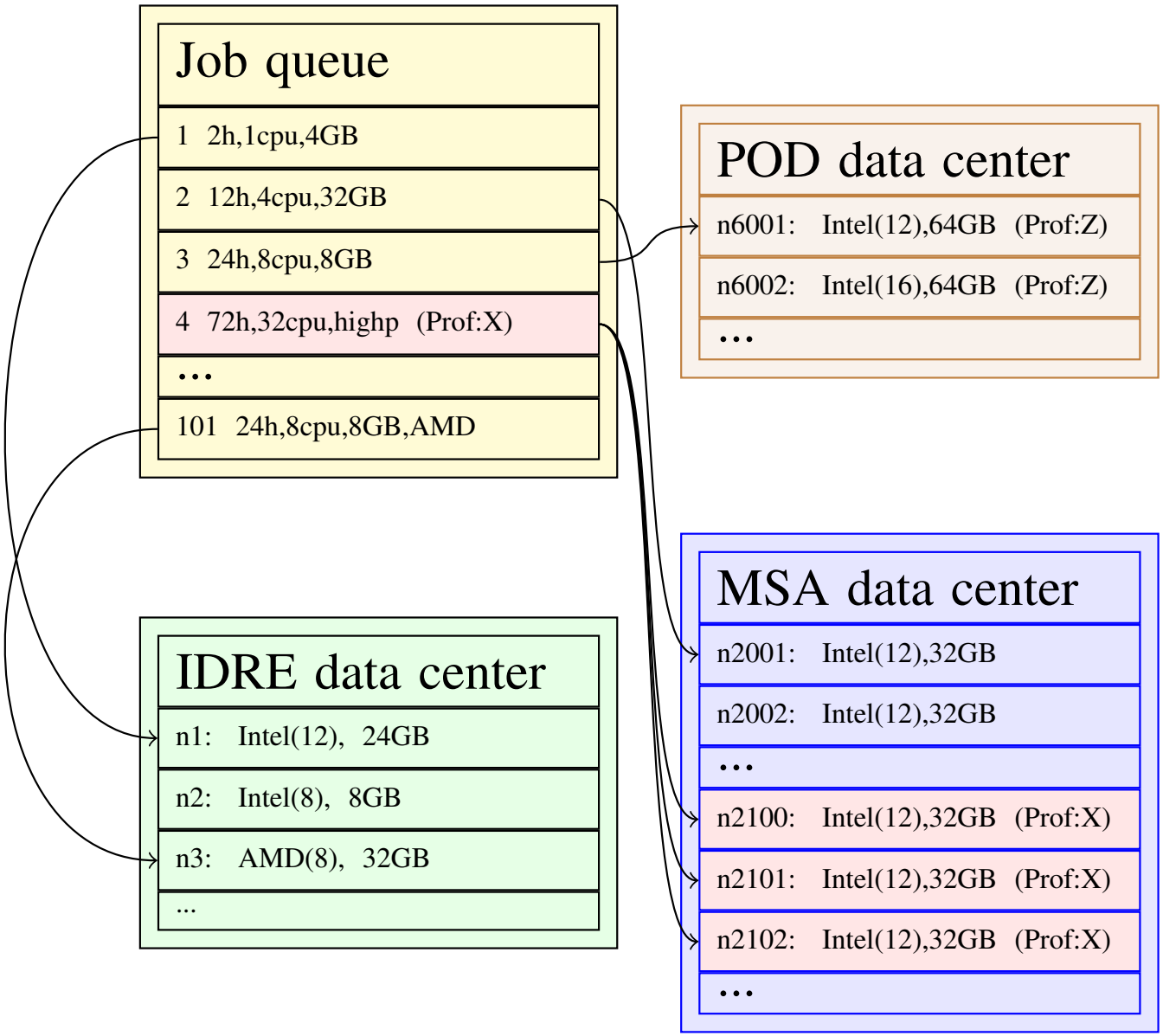
Resource management using Grid Engine

- Hardware diversity
  - CPU types
    - Intel vs AMD, 8,12,16-core, Nahalem vs Sandy Bridge
  - Memory sizes
    - 8 $\sim$ 128 GB per node
  - GPU
  - Commercial software licenses
- Queuing
- Scheduling
  - Dispatch a job to compute node(s)
  - Backfilling
  - Reservation
  - Administrative control (e.g. rolling updates)

Job types

- High-priority (or “highp”) jobs
  - Guaranteed to start in 24 hours, run up to 14 days
  - highp job “stream” runs back-to-back
  - Once completed, nodes become avaiable for shared jobs
  - Maximum job size = number of CPUs your group owns
- Shared jobs
  - Time limit: 24 hours
  - The rest of the cluster is $\gg$ your nodes
  - Opportunity to utilize unused CPUs purchased by others
- Interactive jobs
  - Time limit: 24 hours
  - Login nodes not suitable for heavy computing
  - Interactive-only nodes
  - Running tests, GUI software, etc

Job Dispatch Schematic



Grid Engine job script

- A job script defines what to be done using what resources (CPU, run time, memory, etc)
- A job script contains:
  - job parameters, prefixed by `#$`
  - a standard shell script (e.g. bash)

A bash-based job script example

```
#!/bin/bash
#$ -l h_rt=00:10:00
#$ -pe dc* 4
#$ -cwd
#$ -o mytest.log
echo "hello world!"
```

You could also use the `job.q` or `mpi.q` commands on Hoffman2 to create a job script template.

Which Node(s) Are My Job Running On?

- Do not know at submission time
- Info is available in `$PE_HOSTFILE` – only at run time

Job Script

```
#!/bin/bash
#$ -pe dc* 20
#$ -cwd
#$ -o stdout.log
cat $PE_HOSTFILE
```

Output (stdout.log)

```
n6269 12 pod.q@n6269 UNDEFINED
n6145 8 pod.q@n6145 UNDEFINED
```

Construct MPI host file using $PE_HOSTFILE

Content of $PE_HOSTFILE:

```
n6269 12 pod.q@n6269 UNDEFINED
n6145 8 pod.q@n6145 UNDEFINED
```

We want MPI host file to be: (using OpenMPI)

```
n6269 slots=12
n6145 slots=8
```

One solution

```
cat $PE_HOSTFILE | awk '{print "$1 slots=$2"}'\
    > myhost
mpirun -n 20 -hostfile myhost a.out
```

Grid Engine as a Resource Manager

- Hoffman2 cluster has different CPU types, memory sizes, etc.
- You can request specific hardware feature(s) to run your code
- Warning: if the requested hardware is being occipied, your job may not start immediately

```
$ qhost   # output edited
HOSTNAME    ARCH          NCPU  LOAD  MEMTOT  MEMUSE  SWAPTO  S
-------------------------------------------------------------------
global      -                -     -       -       -       -
n1          amd-2376         8  2.95   31.5G    5.4G   15.3G  7
n39         intel-E5335      8  0.94   31.5G  791.4M   15.3G  6
n6155       intel-E5-2670   16  0.32   63.0G  764.5M   15.3G
n6255       intel-X5650     12 11.98   47.3G    5.0G   15.3G
...
```

Requesting resources for parallel jobs

Hardware features

- Total number of CPU/cores, distribution (parallel environment)
- CPU types (Intel or AMD), `-l arch`
- Wall-clock time limit, `h_rt`
- Per-core memory size, `h_data`, processors `num_proc`
- High priority, `highp` (if you have access)

Example

```
#!/bin/bash
#$ -l h_rt=8:00:00,arch=intel-*,h_data=2G
#$ -pe dc* 8
#$ -cwd
#$ -o stdout.log
```

Note: The stricter the conditions are, the less nodes the scheduler will be able to find

Core distribution across nodes

Grid Engine Parallel Environment (PE)

Commonly used PEs on Hoffman2

- Fixed number of cores per node:
  - `-pe 8threads*`
  - `-pe 12threads*`
- Arbitrary core distribution:
  - `-pe dc*`
- Multiple cores on the SAME node (single node only):
  - `-pe shared`
- Single core per node
  - `-pe node`

Hoffman2 cluster is distributed across 3 data centers

The * in the PEs above means that grid engine can pick nodes from any data center. Include * unless you know what you are doing.

SGE PE: `-pe dc*`

- Example: `-pe dc* 20`
  Get 20 cpu/cores from any node(s)
- Possible distribution:

```
n1    6
n5    8
n6    2
n14   1
n20   3
```

- The scheduler is free to pick any nodes to satisfy the request
- Some nodes may be running others' jobs (unless `-l exclusive`)

SGE PE: `-pe 8threads*`

- Request 8 cpu/cores per node
- Some of the nodes could be 8-, 12- or 16-core nodes – the other remaining cores may be used by other jobs (unless `-l exclusive`)
- Possible distribution:

```
n5001    8   (out of 8 cores)
n5005    8   (out of 8 cores)
n5010    8   (out of 8 cores)
n5035    8   (out of 16 cores)
n5042    8   (out of 12 cores)
```

SGE PE: `-pe 12threads*`

- Request 12 cpu/cores per node
- Some of the nodes could be 12- or 16-core nodes
- Not possible to get any 8-core nodes
- Possible distribution:

```
n6001   12  (out of 12 cores)
n6005   12  (out of 12 cores)
n6010   12  (out of 16 cores)
n6035   12  (out of 12 cores)
n6042   12  (out of 16 cores)
```

SGE PE: `-pe shared`

- Allocating all cpu/cores on the SAME node
- Examples
  - `-pe shared 4`: can start on any 8-, 12- or 16-nodes
  - `-pe shared 12`: can start only on 12- or 16-core nodes
  - `-pe shared 16`: can start only on 16-core nodes
- As of today (May 2013), Hoffman2 has:
  - 690+ 8-core nodes
  - 290+ 12-core nodes
  - 38 16-core nodes

PE examples

- `-pe dc* 20`
  - Get 20 cores from any node(s) (no restriction on distribution)
- `-pe 8threads* 40`
  - Get 40 cores with 8 cores on each node
- `-pe 12threads* 48`
  - Get 48 cores with 12 cores on each node
- `-pe 12threads* 40`
  - Wrong – must be a multiple of "12"
- `-pe node* 4 -l exclusive`
  - Get 4 whole nodes
- `-pe shared 16`
  - Get 16 cores from a single node
  - Note: You are looking for a node with at least 16 cores. Many 8- or 12-core nodes are out of question. (Read: possibly longer wait time)

SGE option: `-l exclusive`

- Your job is the the only one on the node(s)
- Useful option to protect performance
  - Use with parallel environment
  - A parallel job can be slowed down if one of its nodes is slowed down by other users' job
- Wait time could be longer – because the scheduler needs to find/make a completely empty node for this
- But you got to do what you got to do...
- More on this later

SGE option: `-l h_data`

- Memory size associated with each slot (cpu/core)
- Examples
  - `-l h_data=2G`
- Note: If the product `h_data` × (number of slots per node) exceeds any node's total memory, then the job will not be able to find a node to start.
- More on this later

SGE option: `-l num_proc`

- Request a node that has this many cpu/cores
- A way to select CPU types
- Examples
  - `-l num_proc=8` – select from 8-core nodes
  - `-l num_proc=12` – select from 12-core nodes
  - `-l num_proc=16` – select from 16-core node
- Use with caution – using this limits the choices of nodes (e.g. longer wait time)

SGE option: `-l arch`

- Select the CPU architecture
- Examples
  - `-l arch=intel*`
  - `-l arch=amd*`
  - `-l arch=intel-E5-*`
  - `-l arch=intel-E5*|intel-X5650`
- You can choose the CPU types you want
- Use with caution – using this limits the choice of nodes your job can start (i.e. longer wait time)

When `h_data` and PE are used together

- `h_data` is memory size **per slot**
- Each cluster node has a total memory size

```
$ qhost   # output edited
HOSTNAME    ARCH          NCPU  LOAD  MEMTOT  MEMUSE  SWAPT
-------------------------------------------------------------
global      -                -     -       -       -
n1          amd-2376         8  2.95   31.5G    5.4G   15.3
n39         intel-E5335      8  0.94   31.5G  791.4M   15.3
n6155       intel-E5-2670   16  0.32   63.0G  764.5M   15.3
n6255       intel-X5650     12 11.98   47.3G    5.0G   15.3
...
```

- For example, this job will never start:

  `-l h_data=8G,num_proc=12 -pe shared 12`

  The cluster has no 12-core nodes which have 8G$\times$12=96GB memory.

Q: What does the following job script do?

```
#!/bin/bash
#$ -pe node* 4
#$ -l h_rt=4:00:00,num_proc=12,exclusive
cat $PE_HOSTFILE | awk '{print $1" slots=12"}' >mf
mpirun -n 48 -hostfile mf a.out
```

Q: What does the following job script do?

```
#!/bin/bash
#$ -pe node* 4
#$ -l h_rt=4:00:00,num_proc=12,exclusive
cat $PE_HOSTFILE | awk '{print $1" slots=12"}' >mf
mpirun -n 48 -hostfile mf a.out
```

Answer:

Request 4 12-processor nodes, with no other users on these nodes, run 12 MPI processes on each node (total of 48 MPI processes) for 4 hours.

Modules

- Modules are used to load certain environment variables into user's space, e.g.
  - `PATH`: path to executables
  - `LD_LIBRARY_PATH`: dynamic linked libraries
  - `PYTHONPATH`: Python module location(s)
- Hoffman2 has a large number (>100) of software packages and libraries; this is the default mechanism to setup your particular computing environment
- To use Modules
  1. Initialize modules, e.g.

     ```
     source /u/local/Modules/default/init/bash
     ```
  2. Load modules in job scripts, e.g.

     ```
     module load fftw
     module load vtk
     ```

Initializing Modules

This will fail

```
#!/bin/bash
#$ -pe shared 12
#$ -l h_rt=8:00:00
module load openmpi
module load graphviz
module load fftw
mpirun -n 12 a.out
```

Because

- By default job script runs in a non-login shell. This means Modules is not initialized if you put it in shell init files.
- There are two ways to fix this

Remedy 1: initialize Modules inside job scripts

```
#!/bin/bash
#$ -pe shared 12
#$ -l h_rt=8:00:00

# initialize Modules
source /u/local/Modules/default/init/bash

module load openmpi
module load graphviz
module load fftw
mpirun -n 12 a.out
```

Remedy 2: force job script to start as a login shell

Add this line to e.g. `$HOME/.profile`:

```
source /u/local/Modules/default/init/bash
```

```
#!/bin/bash -l
#$ -pe shared 12
#$ -l h_rt=8:00:00
module load openmpi
module load graphviz
module load fftw
mpirun -n 12 a.out
```

Job parameter suggestions

Based on true stories

Here are some general suggestions (for *most* jobs):

- If you are using ≤ 12 cores, have them on the same node: e.g. `-pe shared 12`
- Submit high priority (highp, if you have access) jobs only when necessary
- Do not submit highp jobs if you do not have access. The job will not start.
- Specify time limit long enough to finish your jobs
- Specify time limit short enough to take advantage of backfilling (sooner to start)
- Do not submit a huge number of very short (minutes) jobs. In such cases, try to pack several short jobs into a longer one.

Why is my parallel job not starting

- You specify `h_rt` more than 24 hours without using `highp` (assuming you have `highp` access)
  - You do not have `highp` access if your group has not contributed nodes to Hoffman2 cluster
  - Do not submit highp jobs if you have no access
- You specify conflicting job parameters, e.g.
  - `-l num_proc=8 -pe shared 12`
- You specify very strict parameters
  - `-l exclusive,arch=intel-E5-* -pe node* 20`

Summary and Suggested Strategies

- Request just enough resources for what you need
  - # of CPUs, memory size, time limit
- Use job array(s) instead of individual jobs when submitting many ($>$ 100) independent jobs
- Balanced use of highp and non-highp jobs
  - Submit non-highp jobs if not time critical
- Do not compute on the login nodes
  - Use interactive session (`qrsh`) to do your work
  - Use `dtn2.hoffman2.idre.ucla.edu` to transfer files
- Questions and Problems – email `hpc@ucla.edu`