Backgrounding and Task Distribution In Batch Jobs

James A. Lupo, Ph.D.
Assist Dir Computational Enablement
Louisiana State University

jalupo@cct.lsu.edu

Overview

- Description of the Problem Environment
- Quick Review of shell job control features
- WQ (WorkQueueing)
  - Serial Example
  - Multi-Threaded Example
  - MPI Example
  - Advanced Possibilities

The Problem Environment

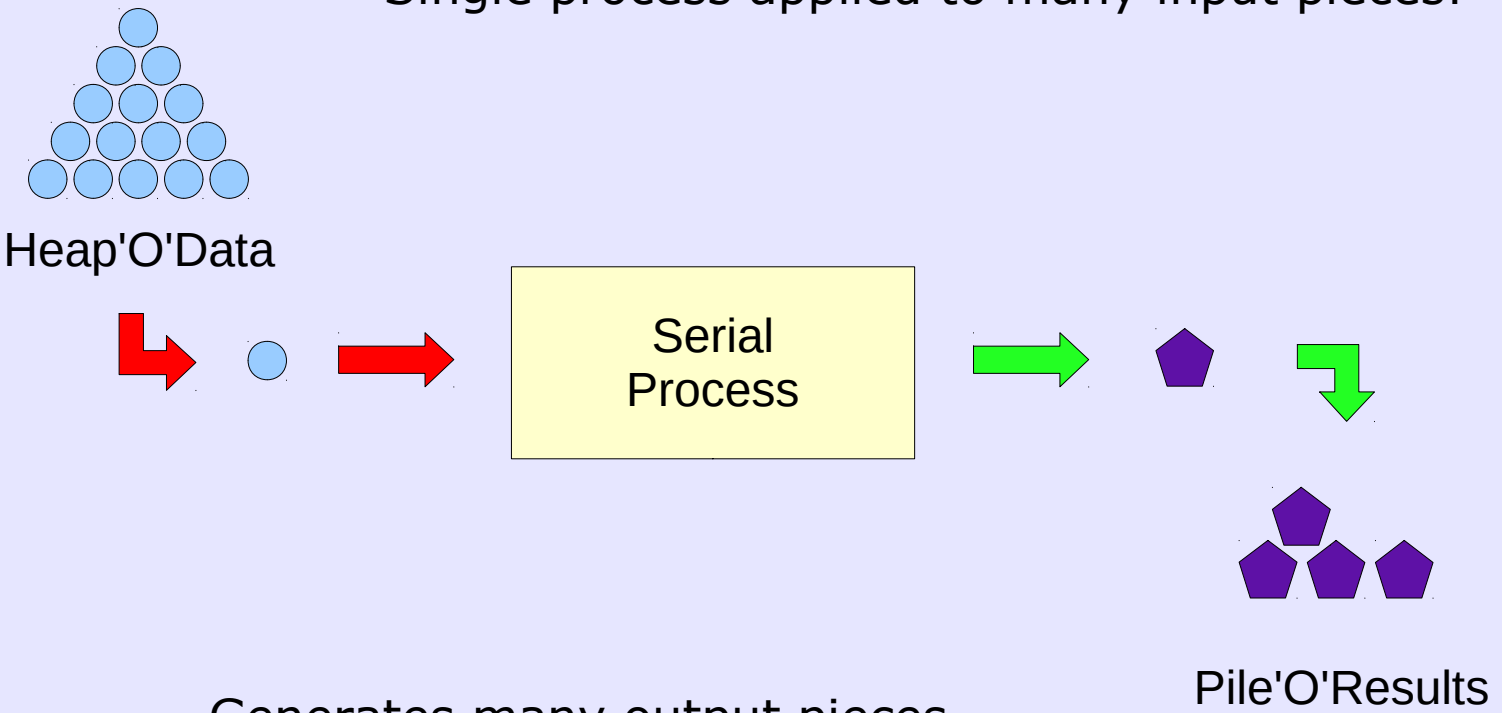
LSU HPC Environment

- Linux operating system.
- Moab/Torque (PBS) environment.
- Clusters tuned for large-scale parallel tasks.
- Full nodes assigned - access to 8, 16, 20, or even 48 cores per node, depending on system.

How does one handle thousands of 1-core tasks without going crazy?

Problem Schematic

Single process applied to many input pieces!



Generates many output pieces.

Manual Command Line

- 10's of thousands of input files processed with the same command syntax:

```
$ myapp infile1 > outfile1
. . .
$ myapp infile1000 > outfile1000
. . .
( and many, many more )
. . .
```

Data sets could come from instruments, automatically generated parameter sweep studies, etc.

Roadblocks to Overcome

- Most workflow tools not well suited to time-limited batch queuing systems.
- Current approach: background or otherwise manually distribute work in the PBS batch script.
  - Requires intermediate-level shell scripting skills
  - Scripting (programming) is foreign to many users of the point/click persuasion.

Desired Solution

- Avoid detailed scripting requirements - but allow flexibility and adaptability.
- Minimize customization and maximize things done *automagically*.
- Make solution batch environment aware, particularly ***wallclock*** time constraints.

Shell Job Control

A Process

- A *process* is the memory, code instructions, and system resources under system control for a running instance of a program.
- Every process has a unique *process ID* (PID).
- Can see with **ps** (process status) command:

```
[jalupo@mike5 ~]$ ps ux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
jalupo   21000  0.0  0.0 110588  2136 ?        S    08:13   0:00 sshd: jalupo@pt
jalupo   21001  0.0  0.0 114424  5800 pts/3    Ss   08:13   0:00 -bash
jalupo   21255  0.0  0.0 112312  1176 pts/3    R+   08:15   0:00 ps ux
```

Shell Job Control => User Process Control

- Not to be confused with ***batch jobs***!
- Shell *jobs* are processes started by the current shell :
- Job control related to the terms: *background*, *suspended*, *foreground*.
- Typically used interactively, but available for use in any shell script.
- Available in all shells: bash, csh, etc.

Job States

- Interactive view of job states (modes):
  - **foreground (Running)** - user / application interaction via keyboard and display - limited to direct control of 1 process.
  - **suspended (Stopped)** - application is stopped, but ready (in memory) to execute - user able to run other processes.
  - **background (Running)** - application runs without user interaction - user is able to do other things - multiprocessing!
- User may move jobs into and out of these states as often as necessary, plus kill or delete.

Common Job Control Commands

- **Ctrl-Z ..** suspends an interactive process.
- **cmd &** ... starts cmd in the background.
- **jobs** .... lists known jobs by number.
- **bg %M** ... backgrounds job **#M**.
- **fg %N** ... foregrounds job **#N.**
- **kill %L** . kills job **#L**.

Example Scripts

Create a couple of small bash scripts, naming them **demo.sh** and **driver.sh,** with the following content:

demo.sh

```
#! /bin/bash
PID=$$
echo "$PID is starting."
for N in $(seq 1 $1); do
   sleep 2
   echo "$PID slept $N times."
done
echo "$PID is ended."
```

driver.sh

```
#! /bin/bash
PID=$$
echo "Driver $PID is starting."
./demo.sh 3
./demo.sh 6
echo "Driver $PID has ended."
```

Make them executable:

chmod u+x demo.sh driver.sh

Run **demo.sh**

- Try: **$ ./demo.sh 3**
- You should see:

```
3432 is starting.
3432 slept 1 times.
3432 slept 2 times.
3432 slept 3 times.
3432 is ended.
```

- 3432 is the process ID assigned when the script started.
- Every running PID is unique.

Run **driver.sh**

- Try: **$ ./driver.sh**

```
Driver 3485 is starting.
3486 is starting.
3486 slept 1 times.
3486 slept 2 times.
3486 slept 3 times.
3486 is ended.
3491 is starting.
3491 slept 1 times.
3491 slept 2 times.
3491 slept 3 times.
3491 slept 4 times.
3491 slept 5 times.
3491 slept 6 times.
3491 is ended.
Driver 3485 has ended.
```

3 process ID's:

3485, 3486, 3491.

They ran ***sequentially***.

Launching in Background With **&**

Syntax: **`$ cmd [-switches] [args] [< stdin] [> stdout] &`**

- Jobs requiring interaction are suspended.
- Non-interactive jobs run in background.
- **`stdio*`** streams stay as is unless redirected.
- Parent shell determines how jobs are handled when shell terminates.

* stdin, stdout, stderr

Try It

- Try: **./demo.sh 5 &**
- Note how output gets mixed up on the screen.
- Try: **./demo.sh 5 > foo &**
- It runs, and output goes to file foo.
- Try a sequence of 3 commands:
  1) **./demo.sh 15 > foo &**
  2) **./demo.sh 20 > bar &**
  3) **jobs**

Modify **driver.sh**

driver.sh

```
#! /bin/bash
PID=$$
echo "Driver $PID is starting."
./demo.sh 3  &
./demo.sh 2  &
echo "Driver $PID has ended."
```

Execution looks a little strange:

```
host:~/Scratch$ ./driver.sh
Driver 4394 is starting.
Driver 4394 has ended.
host:~/Scratch$ 4395 is starting.
4396 is starting.
4395 slept 1 times.
4396 slept 1 times.
4395 slept 2 times.
4396 slept 2 times.
4396 is ended.
4395 slept 3 times.
4395 is ended.
```

Having the driver script complete before the processes it started is not a good thing. If a job script, all user processes will be killed when it ends.

Need one more change.

Modify **driver.sh:** Add **wait**

driver.sh

```
#! /bin/bash
PID=$$
echo "Driver $PID is starting."
./demo.sh 3 &
./demo.sh 2 &
wait
echo "Driver $PID has ended."
```

Execution now looks like:

```
host:~/Scratch$ ./driver.sh
Driver 4473 is starting.
4474 is starting.
4475 is starting.
4475 slept 1 times.
4474 slept 1 times.
4474 slept 2 times.
4475 slept 2 times.
4475 is ended.
4474 slept 3 times.
4474 is ended.
Driver 4473 has ended.
```

The wait is key to making sure sub-processes finish before the caller does.

Keep Output Separate

driver.sh

```
#! /bin/bash
PID=$$
echo "Driver $PID is starting."
./demo.sh 3 > d3.out &
./demo.sh 2 > d2.out &
wait
echo "Driver $PID has ended."
```

Execution now looks like:

```
host:~/Scratch$ ./driver.sh
Driver 4473 is starting.
Driver 4473 has ended.
host:~/Scratch$ ls
   driver.sh demo.sh
   d3.out  d2.out
```

Driver output is displayed on the terminal. Application output goes into individual files.

PBS Script Running 4 Serial Programs

```
#! /bin/bash
#PBS -l nodes=1:ppn=4
#PBS . . . other settings . . .

myprog < infile1 > outfile1 &
myprog < infile2 > outfile2 &
myprog < infile3 > outfile3 &
myprog < infile4 > outfile4 &

wait
```

The **wait** makes sure all 4 tasks (shell jobs) have completed, else when the script ends, the job manager will kill all the user's running programs in preparation for the next job.

Running 2 Multi-Threaded Programs

- With 16 cores there is the ability to run 2 8-thread programs - almost as easy as running serial programs.

```
#! /bin/bash
#PBS -l nodes=1:ppn=16
#PBS . . . other settings . . .

export OMP_NUM_THREADS=8
myprog < infile1 > outfile1 &
myprog < infile2 > outfile2 &

wait
```

Multi-Process MPI Programs

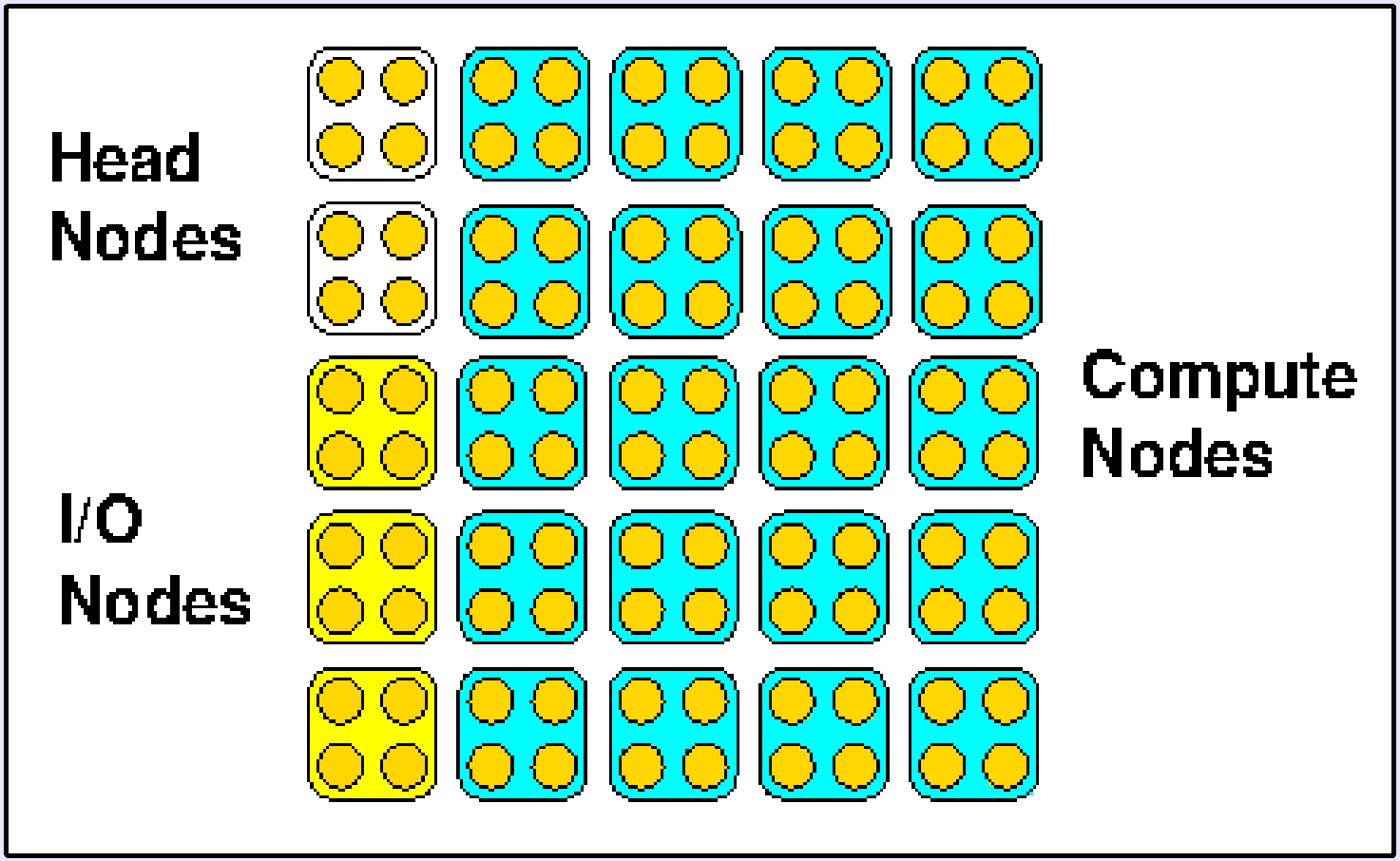
20-core node? Consider running 2 10-process MPI apps.

```
#! /bin/bash
#PBS -l nodes=1:ppn=20
#PBS . . . other settings . . .

NPROCS=10
mpirun -np $NPROCS -machinefile $PBS_NODEFILE \
   mprog < infile1 > outfile1 &
mpirun -np $NPROCS -machinefile $PBS_NODEFILE mprog \
   < infile2 > outfile2 &

wait
```

Higher Core Counts

- Multiple multi-threaded or multi-process tasks allows one script to take advantage of all cores in a node.
- Depends on specs for a given clusters.
- Program types can be mixed, so long as the required number of cores is consistent with what the node provides.
- Scaling up (more nodes plus more cores) will complicate the scripting required.

Use Multiple Nodes?



On typical clusters, the work must be done on the compute nodes. We could submit multiple single node job scripts, but how about using more than one node at a time?

Multi-Node Considerations

- The ***mother superior*** node is only one given all the job information, like environment variables, list of node names, etc.
- Start programs on other nodes with remote shell commands, like **ssh**.
- Account for shared and local file systems.
- Assure all programs finish before script exits.
- Be aware of efficiency (load balancing).

8 Serial Programs on Two 4-core Nodes

8 Cores Total

Node 1
Mother Superior
4 Tasks

Node 2
Compute Node
4 Tasks

```
#! /bin/bash
#PBS -l nodes=2:ppn=4
#PBS . . . other settings . . .

export WORKDIR=/path/to/work/directory
cd $WORKDIR

myprog < infile1 > outfile1 &
myprog < infile2 > outfile2 &
myprog < infile3 > outfile3 &
myprog < infile4 > outfile4 &

# Discover second host name, somehow, then

ssh -n $HOST2 “cd $WORKDIR; myprog < infile5 > outfile5” &
ssh -n $HOST2 “cd $WORKDIR; myprog < infile6 > outfile6” &
ssh -n $HOST2 “cd $WORKDIR; myprog < infile7 > outfile7” &
ssh -n $HOST2 “cd $WORKDIR; myprog < infile8 > outfile8” &

wait
```

-n suppresses reading from stdin and just starts the program.
The path to myprog is assumed known (.bashrc?)

Some Real Scripting Required

- 40 programs on 2 nodes would clearly make life complicated.
- Real shell magic needed to figure out host names - maybe a little opaque:

```
NAMES=($(uniq $PBS_HOSTFILE))
HOST2=NAMES[1]
```

Assumes host names are assigned starting with the mother superior, and in sorted order. More work if this is not the case!

Automating Multiple Nodes

Node 1
Mother Superior
4 Tasks

Node N
Compute Node
4 Tasks

```
# Get the node names
NODES=($(uniq $PBS_NODEFILE ))
# Get the number of names
NUMNODES= $(uniq $PBS_NODEFILE ) | wc -l | awk '{print $1-1}')
# Do commands on first node:
cmd stuff &
. . . start as many as desired (but customize each line!). . . .
cmd stuff &
# Loop over all the nodes, starting with the second name:
for i in $(seq 1 $((NUMNODES-1)) ); do
   ssh -n ${NODES[$i]} cmd stuff &
   ... start as many as desired (but customize each line!). . . .
   ssh -n ${NODES[$i]} cmd stuff &
done
wait
```

Really not fun if you don't like shell scripting, yet it gets worse!

Consider Multi-Threaded / MPI Task Requirements

- Have to pass the thread count.
- Have to construct partial host name lists.
- Have to worry about ***CPU affinity***!
- Involves basic shell programming, and maybe gets involved with fussy quoting rules to get everything passed correctly.
- Manual scripting doesn't really SCALE!

Solution Requirements

- Isolate the things that change with each task.
- Make user setup as simple as possible.
- Automate most of the magic.
- Try to deal with batch job walltime issues.

Questions?

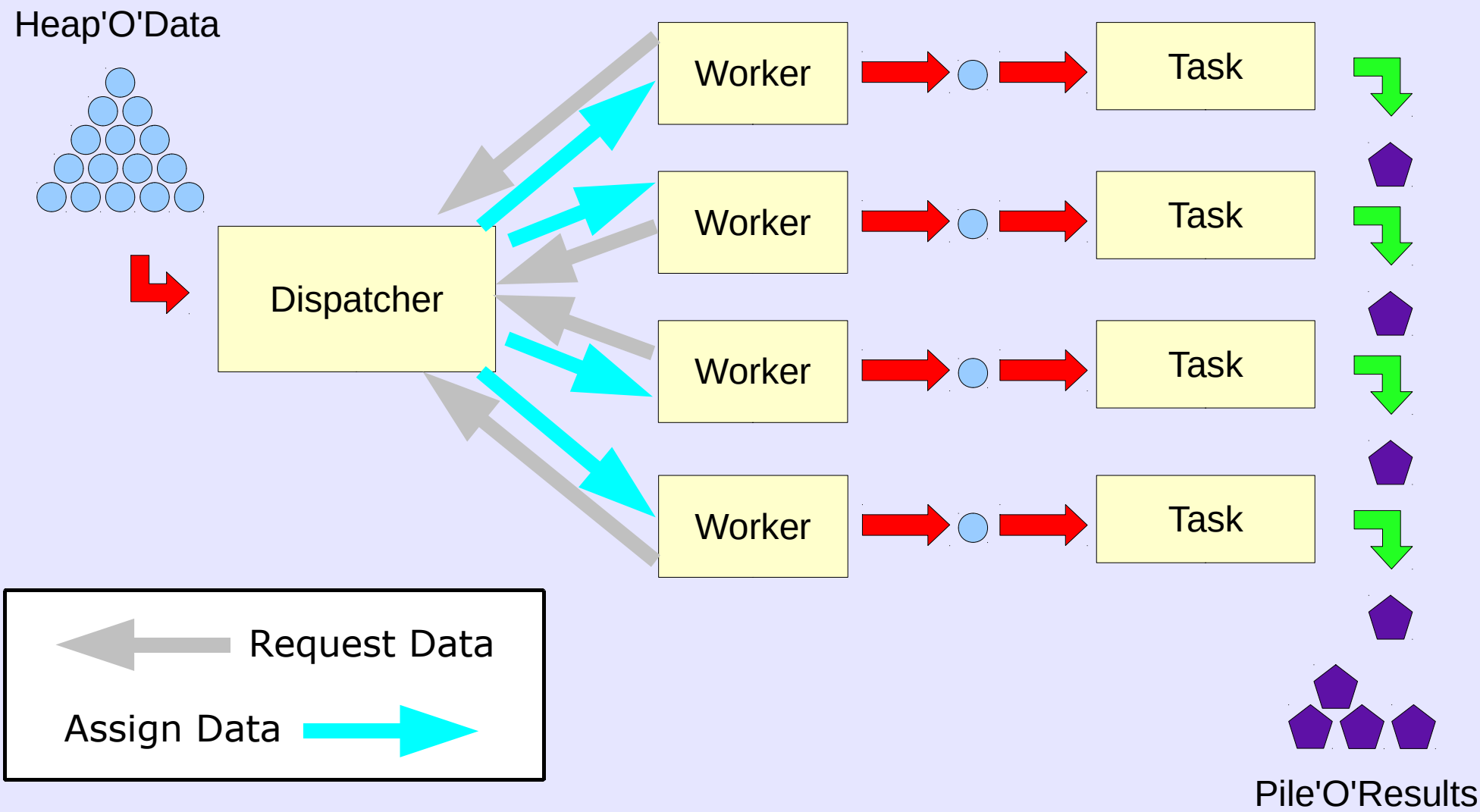
- Before we move on, any further clarifications of the basic shell scripting concepts needed?
- Any concerns over difference between a shell script and a PBS job script?

WQ and It's Components

What Is WQ?

- Dispatcher/Worker model - WQ roughly stands for *Work Queueing*.
- Handles ***tasks*** – defined as the work necessary to process one line from an input file.
- Multiple ***workers*** execute the tasks - one worker per simultaneous task on all nodes.
  - ***Workers*** request a ***task*** from the ***Dispatcher***.
  - ***Workers*** share task times with ***Dispatcher***.
  - ***Dispatcher*** won't assign a new task if it estimates that insufficient time remains to complete it.

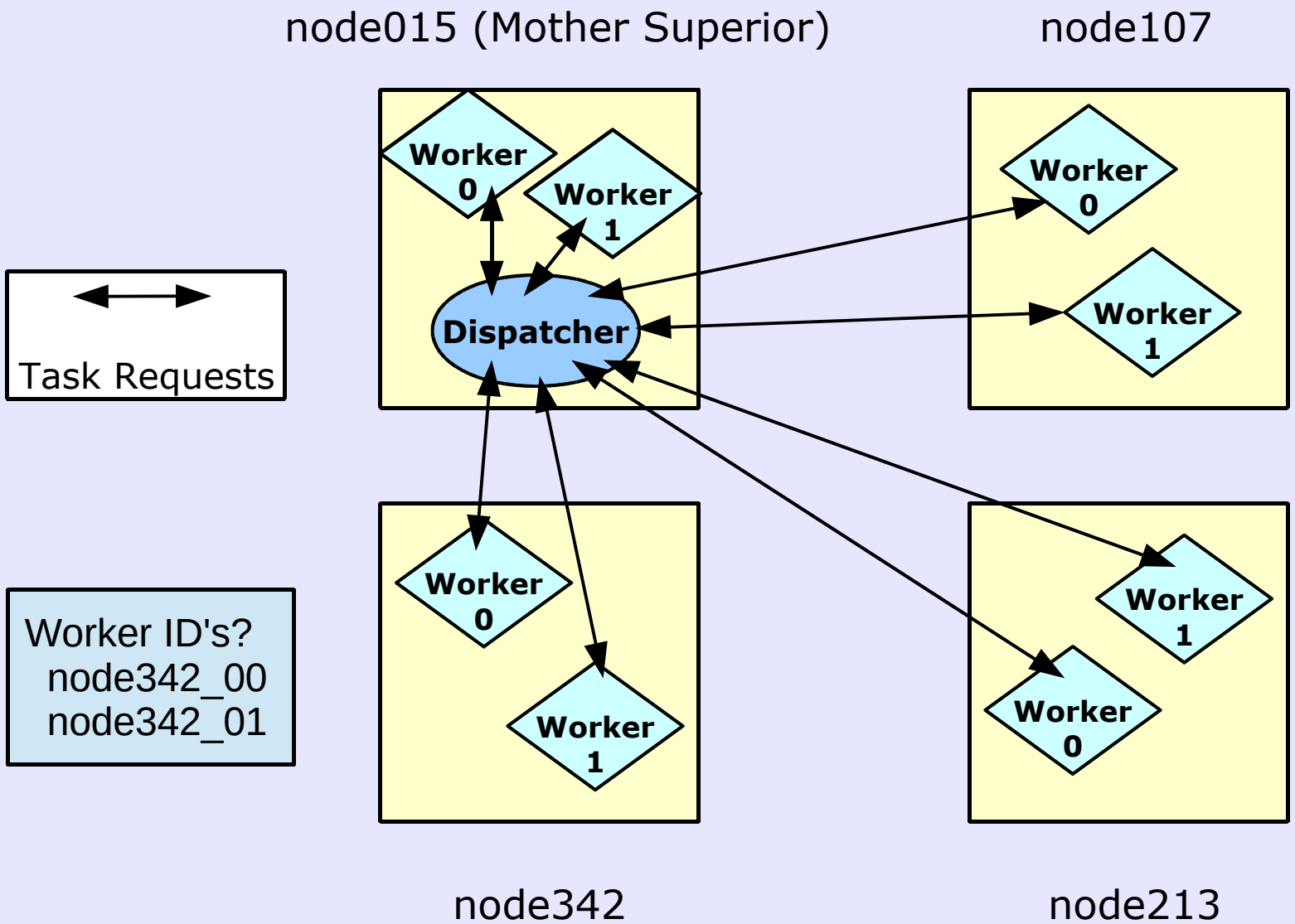
WQ Schematic



Design Assumptions

- **Task** viewed as *application* + *data*
  - Consider one app, many input file names.
- **Dispatcher** manages *an input file with file names*, handing out one per request.
- A **Worker** submits a *request*, and reports run time of last task executed.
- **Dispatcher** tracks longest task time. Uses it, and a safety margin of 1.25, to decide if there is sufficient time before handing out another task.

WQ Runtime Schematic



WQ Components

- **wq.py** – A Python script that implements the **dispatcher** and **workers -** *no user servicable parts inside!***.**
- **wq-pbs.sh** - The PBS specific part of WQ - *no user servicable parts inside!*
- **wq.pbs** – A PBS batch script template with a few user required variable settings and call to wq-pbs.sh built in.
- **wq.sh** – A user created script (could be a program) that accepts an input file line as it's argument.
- **wq.list** – A user created file containing input file names, one per line (suggest using absolute path names).
- **wq.log.N** - Output file of WQ actions. N = PBS job number.

The names of files can be changed – just keep consistent with the contents of the PBS script – wq.py and wq-pbs.sh must be executable and in PATH.

wq.pbs : PBS Preamble Section

The PBS script is divided into 2 parts: the WQ prologue (which includes the PBS preamble), and the WQ epilogue. Only the first two contain items the user changes:

```
#! /bin/bash
##########################################
#
# Begin WQ Prologue section
##########################################
#
#PBS -A hpc_myalloc_03
#PBS -l nodes=4:ppn=16
#PBS -l walltime=00:30:00
#PBS -q workq
#PBS -N WQ_Test
```

The PBS script itself must be set executable, as it will be run by nodes other than the mother superior, if necessary.

wq.pbs : Prologue Section

```
# “Workers Per Node” - WPN * processes = cores (PPN)

WPN=4

# Set the working directory:

WORKDIR=/work/user

# Use a file with 82 names listed:

FILES=${WORKDIR}/82_file_list

# Name the task script each worker is expected to run on the file
# name provided as it's only argument.

TASK=${WORKDIR}/wq_timing.sh

START=1

VERBOSE=0
```

1

2

3

4

5

6

wq.pbs : Epilogue Section

- Serious magic happens in the Epilogue section – it consists of a single incantation:

  wq-pbs.sh $0 $WPN $WORKDIR $FILES $START $TASK $VERBOSE $1

- What does **wq-pbs.sh** do? In summary:
  - Some sanity checking of settings.
  - Determines if running on mother superior node.
    - Preps information exchange process
    - Starts job script on all other compute nodes.
    - Starts dispatcher, and local workers.
  - Compute nodes start their workers.
  - All workers start the ***request - execute*** cycle until walltime runs out or there are no more tasks to assign.

wq.sh

This name represents an actual shell script, program, or any other type of executable which works on the provided input file line. What it does should be consistent with the settings (i.e. multi-threaded, multi-process, serial) in wq.pbs.

Before launching, it can/should be tested with a single line*:

```
$ ./wq.sh line of text from file
```

If it works manually, it should function correctly when called by a worker.

*A shell programmer would see this as a command and 5 arguments!

wq.list

This is nothing more than a file containing lines of data, let's say file names. Could generate one with the **find** command:

```
$ find `pwd` -name '*.dat' -print > wq.list
```

In many cases, using absolute paths for the file names is best since the script can extract information about the location from the name (hence the use of `pwd` to get the current working directory).

Sample:

```
/work/jalupo/WQ/Examples/Timing/chr13/chr13_710.bf
/work/jalupo/WQ/Examples/Timing/chr13/chr13_727.bf
/work/jalupo/WQ/Examples/Timing/chr13/chr13_2847.bf
/work/jalupo/WQ/Examples/Timing/chr13/chr13_711.bf
```

wq.log.N

wq.py provides a record of activity in wq.log.N (N is job number). All lines have the form:

<src>:<tag>:<data1>:...:<dataM>

where:

<src> : Dispatcher or Worker

<tag> : Data indicator

<data> : Specific pieces of data

What gets displayed is controlled by the setting of VERBOSE.

See user manual on how to decode all lines.

A Serial Task Example

A Simple **wq.sh**

Let's not try to do much except waste some time and show what can be done with a file name:

```
#! /bin/bash

# Argument 1 is assumed to be a path name.

FILE=$1
DIR=`dirname ${FILE}`
BASE=`basename ${FILE}`

# Now just echo the important vars, and sleep.

echo "DIR=${DIR}; BASE=${BASE}"
echo "WQ_NUMACTL=${WQ_NUMACTL}; WQ_CPT=${WQ_CPT}"
echo "That's all, folks!"
T=`expr 2 + $RANDOM % 10`
echo "Sleeping for $T seconds."
sleep $T
```

backticks NOT single quotes!

An Input File List

Let's look for files with .bf extensions:

```
$ find /work/user -name '*.bf' -print > file_list
```

And assume it produces 82 names like so:

```
/work/user/chr13/chr13_710.bf
/work/user/chr13/chr13_727.bf
/work/user/chr13/chr13_2847.bf
/work/user/chr13/chr13_711.bf
/work/user/chr13/chr13_696.bf
. . .
```

A Serial **wq.pbs**

Assume system has 16 cores per node, we could request 2 nodes to run 32 tasks at a time. The PBS preamble option would look like:

```
#PBS -l nodes=2:ppn=16
```

(2x16=32!) Now we just need to set the 6 PBS prologue variables accordingly:

```
WPN=16
WORKDIR=/work/user
FILES=${WORKDIR}/file_list
TASK=${WORKDIR}/wq.sh
START=1
VERBOSE=0
```

Serial Example wq.log Lines

```
Worker:Timings:80:mike150_3:1449778034.03:1449778042.04:8.01
Worker:Stdout:80:mike150_3:True
  DIR=/work/jalupo/WQ/Examples/Timing/chr23; BASE=chr23_707.bf
  WQ_NUMACTL=numactl --physcpubind=12-15 -- ; WQ_CPT=4
  That's all, folks!
  Sleeping for 8 seconds.
Worker:Stderr:80:mike150_3
Dispatcher:Last:82
Dispatcher:Shutdown:82:4.81:1.01:10.01:54.74:52.06:58.09
```

Worker:Timings -- task 80 timing information.
Worker:Stdout -- task 80 was successful, followed by stdout lines.
Worker:Stderr -- stderr lines (none in this case).
Dispatcher:Last -- last task was task 82.
Dispatcher:Shutdown -- number of tasks and runtime info.

A Multi-Threaded Example

Adjust For Multi-Threading

- **wq.sh** – set up for multi-threading. We'll use OpenMP for this example.
- **wq.pbs** - adjust so number of threads and number of workers is consistent with number of cores on the nodes.

Multi-Threaded Example **wq.sh**

Shortcut

Threads

Keep It Readable

Production vs Testing

```
#! /bin/bash

# Set a variable as the absolute path to the blastn executable:

BLASTN=/usr/local/packages/bioinformatics/ncbiblast/2.2.28/gcc-4.4.6/bin/blastn

export OMP_NUM_THREADS=${WQ_CPT}

FILE=$1
DIR=`dirname ${FILE}`
BASE=`basename ${FILE}`

# Build the rather complex command line.

CMD="${WQ_NUMACTL}
CMD="${CMD} ${BLASTN} -task blastn -outfmt 7 -max_target_seqs 1"
CMD="${CMD} -num_threads ${OMP_NUM_THREADS}"
CMD="${CMD} -db /project/special/db/img_v400_custom/img_v400_custom_GENOME"
CMD="${CMD} -query ${FILE}"
CMD="${CMD} -out ${DIR}/IMG_genome_blast.${BASE}"

# For testing purposes, use "if false". For real runs, use "if true":

if true ; then
   eval "${CMD}"
else
   echo "${CMD}"
   # This just slows things way down for testing.
   sleep 1
fi
```

???

WQ_NUMACTL & WQ_CPT

- OpenMP, and MPI, should be told what cores to run on if multiple processes share a node.
- Each worker determines the cores it should use and sets two environment variables for the benefit of the task scripts:
  - **WQ_NUMACTL** - **numactl** string to set *CPU affinity*. Add to front of the task command!
  - **WQ_CPT** - the number of "cores per task" that are available. Use to set MPI process or OpenMP thread counts.

Multi-Threaded Example **wq.pbs**

Assume the system has 16 cores per node. That means we could run 4 4-thread tasks per node. On 2 nodes we could run 8 tasks at a time, so let's set that up in the PBS preamble:

```
#PBS -l nodes=2:ppn=16
```

Now we just need to make the PBS prologue variables agree:

```
WPN=4
WORKDIR=/work/user
FILES=${WORKDIR}/file_list
TASK=${WORKDIR}/wq.sh
START=1
VERBOSE=0
```

An MPI Example

Adjust For MPI

- **wq.sh** – set up for small number (same node only) of MPI processes per task.
- **wq.pbs** - adjust so number of processes and number of workers is consistent with number of cores on the nodes.

MPI Example **wq.sh**

Build Host Lists

```
#! /bin/bash

FILE=$1
DIR=`dirname ${FILE}`
BASE=`basename ${FILE}`

PROCS=${WQ_CPT}
HOSTNAME=`uname -n`
HOSTLIST=""
for i in `seq 1 ${PROCS}`; do
   HOSTLIST="${HOSTNAME},${HOSTLIST}"
done
HOSTLIST=${HOSTLIST%,*}

CMD="${WQ_NUMACTL} mpirun -host ${HOSTLIST}"
CMD="${CMD} -np ${PROCS} mb < ${FILE} > ${BASE}.mb.log"

cd $DIR

# Clean out any previous run.

rm -f *.[pt] *.log *.ckp *.ckp~ *.mcmc

# For testing purposes, use "if false". For production, use "if true"

if false ; then
   eval "${CMD}"
else
   echo "${CMD}"
   echo "Faking It On Hosts: ${HOSTLIST}"
   sleep 2
fi
```

MPI Example **wq.pbs**

Assume the system has 16 cores per node. That means we could run 2 8-process tasks per node. On 2 nodes we could run 4 tasks at a time, so let's set that up in the PBS preamble:

```
#PBS -l nodes=2:ppn=16
```

Now we just need to make the PBS prologue variables agree:

```
WPN=2
WORKDIR=/work/user
FILES=${WORKDIR}/wq.lst
TASK=${WORKDIR}/wq_mb.sh
START=1
VERBOSE=0
```

Advanced Usage

Advanced Usage Possibilities

- Many tasks do not depend on only file names.
- Not a problem - use a line for anything!
  - Multiple arguments!
  - Complete command lines!
- The dispatcher sends the entire line to the worker. The task just has to know how to handle it.

Parameter Sweep

- Parameter sweeps adjust input variables over some range of values to see how the output changes.
- One example: for given cannon ball weight, the range of a cannon depends on powder load and elevation.
- Assume: range.sh -e X -p Y
  - X is elevation in degrees.
  - Y is pounds of powder load.

range.sh

```
#! /bin/bash
#
# This does nothing but reflect the
command line arguments.

echo "range.sh called. The arguments
provided were:"
echo "args: $*"
```

Generate Parameters

```
#! /bin/bash
echo "" > args.lst
for elevation in `seq  5.0 5 85.0`; do
   for pounds in `seq 1.5 0.1 5.0`; do
      echo "$elevation $pounds" >> args.lst
   done
done
```

args.lst

- Generated 612 lines:

```
5.0 1.5
5.0 1.6
5.0 1.7
5.0 1.8
5.0 1.9
5.0 2.0
5.0 2.1
5.0 2.2
5.0 2.3
5.0 2.4
5.0 2.5
... and many more ...
```

- Treat as two arguments per command line.

Multiple Argument Task Script

```
#! /bin/bash

CANNON=dalhgren

# Make sure directory exists.

if [ ! -d ${CANNON} ] ; then
   echo "This directory must exist before running job: ${CANNON}"
   exit 1
fi

# Expect elevation as 1st argument, poundage as 2nd argument:

CMD="./range.sh -e $1 -p $2 ${CANNON}.dat > ${CANNON}/${1}_${2}.dat"

if true ; then
   eval "${CMD}"
else
   echo "CMD=${CMD}"
   T=`expr 2 + $RANDOM % 10`
   echo "Sleeping for $T seconds."
   sleep $T
fi
```

Sample Result:

- dalgren/5.0_1.5.dat contains:

range.sh called. The arguments provided were:
args: -e 5.0 -p 1.5 dalhgren.dat

Full Command Line Task Script

```
#! /bin/bash
#
# This script treats all args as a complete command line.

CMD="$*"

# 'if true' execute the command. 'if false' just echo it back.

if false ; then
   eval "${CMD}"
else
   echo "CMD=${CMD}"
   T=`expr 2 + $RANDOM % 10`
   echo "Sleeping for $T seconds."
   sleep $T
fi
```

Generate Full Command Lines

```
#! /bin/bash
echo "" > cmdlines.lst
for elevation in `seq  5.0 5 85.0`; do
   for pounds in `seq 1.5 0.1 5.0`; do
      echo "./range.sh -e $elevation -p $pounds \
            cannon.dat" >> cmdlines.lst
   done
done
```

cmdlines.lst

- Generated 612 lines:

```
./range.sh -e 5.0 -p 1.5 cannon.dat
./range.sh -e 5.0 -p 1.6 cannon.dat
./range.sh -e 5.0 -p 1.7 cannon.dat
./range.sh -e 5.0 -p 1.8 cannon.dat
./range.sh -e 5.0 -p 1.9 cannon.dat
./range.sh -e 5.0 -p 2.0 cannon.dat
./range.sh -e 5.0 -p 2.1 cannon.dat
./range.sh -e 5.0 -p 2.2 cannon.dat
./range.sh -e 5.0 -p 2.3 cannon.dat
./range.sh -e 5.0 -p 2.4 cannon.dat
... and many more ...
```

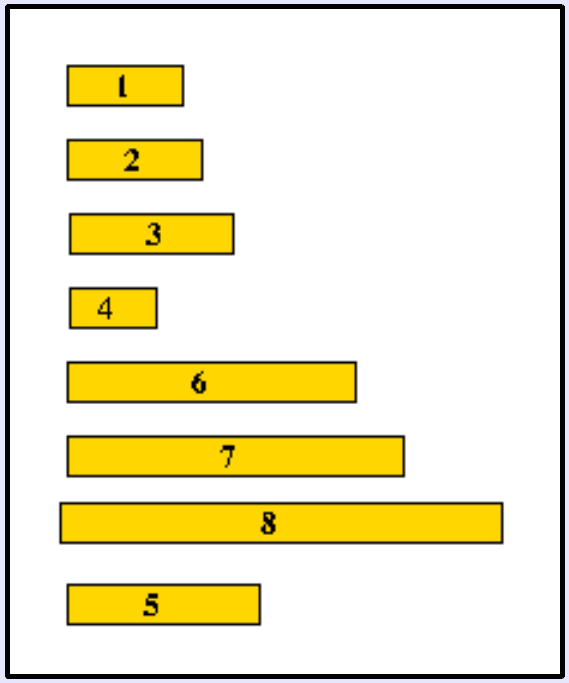
An Aside on Load-Balancing

Load Balancing Issues

- The more uniform the task times are across all tasks, the more likely a job will end gracefully.
- Take a look at the concepts.
- Illustrate potential problem.
- Discuss how to analyze a job's efficiency.

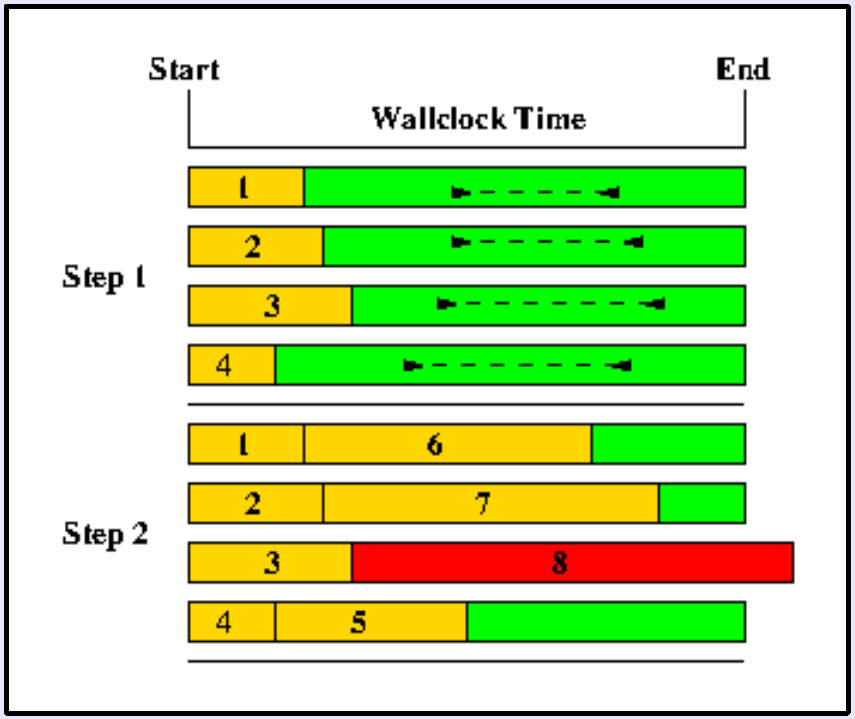
Task Run Times

Imagine a set of 8 tasks, number for identification only, and represented by bars propotional to their run times.



Insufficient Walltime

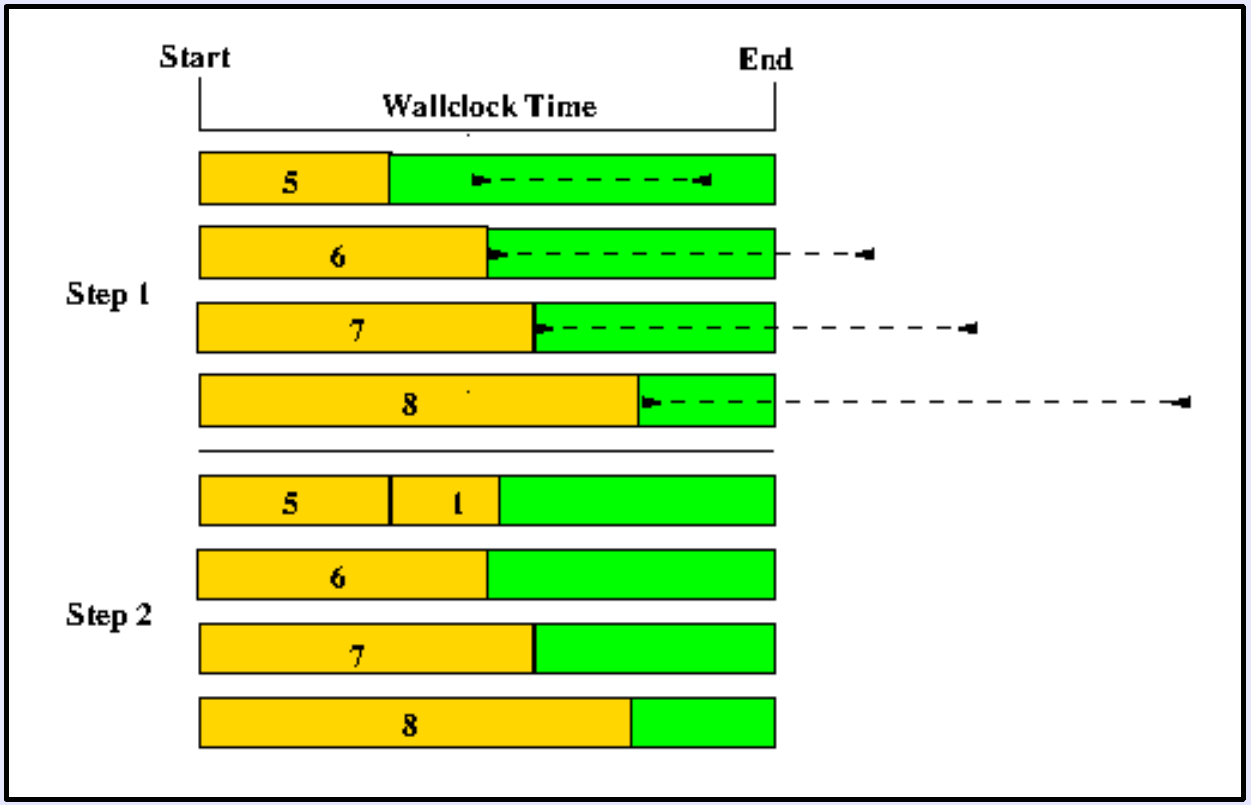
PBS walltime sets the maximum wallclock time a job is allowed. Imagine the tasks get assigned in the following order:



Dashed bars show estimated times for next task - they all appear to fit remaining time.

Bad estimate for Task 8!

Sufficient Walltime



5 estimates sufficient time remains, but 6-8 do not!

5 requests 1 more task, but 6-8 stop!

Load Balance Implications

- Order by longest running first, if possible.
- Run many tasks so representative times are seen early in the job.
- If range of times not known, there is no good way to make absolutely sure jobs complete gracefully.
- Output format allows analysis.