Backgrounding and Task Distribution
In Batch Jobs

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

jalupo@cct.lsu.edu

XSEDE
Extreme Science and Engineering
Discovery Environment

Overview

- Description of the Problem Environment
- Quick Review of Command Shell Job Control
- WQ Components
- Simple Serial Example
- Multi-Threaded Example
- MPI Example
- Comments on Monitoring and Statistics

Center for Computation
& Technology

Information Technology
Services

The Problem Environment

XSEDE

Extreme Science and Engineering
Discovery Environment

LSU HPC Environment

- Traditional Moab/Torque (PBS) environment.

- Usage is allocation controlled.

- Large clusters tuned for large-scale parallel processing.

- Small 39 node cluster for serial work.

- Growing interest in bioinformatics causing an explosion in serial job requirements.

High Performance Computing @ Louisiana State University

LSU
Center for Computation
& Technology

LSU
Information Technology
Services

BIG File Counts

- Jobs with 10's of thousands of input files and identical processing tasks.

- 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 is foreign to non-traditional users.

XSEDE
Extreme Science and Engineering
Discovery Environment

Desired Solution

- Avoid detailed scripting requirements - but allow flexibility and adaptability.

- Minimize customization and do most things automagically.

- Make method batch environment aware, particularly wallclock time constraints.

Command Shell Job Control

Shell Job Control

- Shell job control features - manage, background, suspend, foreground.

- Not to be confused with **batch jobs**!

- Typically used interactively.

- Available for use in any shell script.

Common Commands

- `Ctrl-Z . . .` suspends an interactive process.
- `bg %M . . .` sends job #M into background.
- `fg %N . . .` brings job #N into foreground
- `cmd & . . .` starts cmd in the background.
- `kill %L . .` kills job #L.
- `jobs . . .` lists known jobs.

Example

1) Launch EMACS in background (uses an X-11 GUI).
2) VIM and LESS programs started, then suspended with Ctrl-Z.
3) Show jobs.

```
[jalupo@mike1 ~]$ emacs rps.c++ &
[1] 28835
[jalupo@mike1 ~]$ less .bashrc

[2]+  Stopped                 less .bashrc
[jalupo@mike1 ~]$ jobs
[1]-  Running                 emacs rps.c++ &
[2]+  Stopped                 less .bashrc
[jalupo@mike1 ~]$ vim .soft

[3]+  Stopped                 vim .soft
[jalupo@mike1 ~]$ jobs
[1]   Running                 emacs rps.c++ &
[2]-  Stopped                 less .bashrc
[3]+  Stopped                 vim .soft
[jalupo@mike1 ~]$
```

1
2
3

Hands-On Example

1) Run: $ **vimtutor**
2) Suspend with Ctrl-Z.
3) Run: $ **ls -l**
4) Run: $ **jobs**
5) Identify which job is vimtutor.
6) Run: $ **fg %N**
7) Suspend again.
8) Run: $ **vim &**
9) Run: $ **jobs**
10) Run: $ **kill %N %M**

Launching Processes With &

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

- Interactive jobs are suspended.
- Non-interactive jobs start running.
- **stdio*** streams stay as is unless redirected.
- Parent shell determines how jobs are handled when shell terminates.

* stdin, stdout, stderr

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

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

export OMP_NUM_THREADS=2
myprog < infile1 > outfile1 &
myprog < infile2 > outfile2 &

wait
```

Multi-Process MPI Programs

On an 8-core node, 2 4-process MPI tasks can be run.

```
#! /bin/bash
#PBS -l nodes=1:ppn=8
#PBS . . . other settings . . .

NPROCS=4
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.

- 4, 8, 16, 20, and 40 cores per node are available on the various local clusters.

- The types of programs can be mixed, so long as the required number of cores is consistent with what the node provides.

- Scaling up (more cores plus more nodes) will obviously complicate the scripting required.

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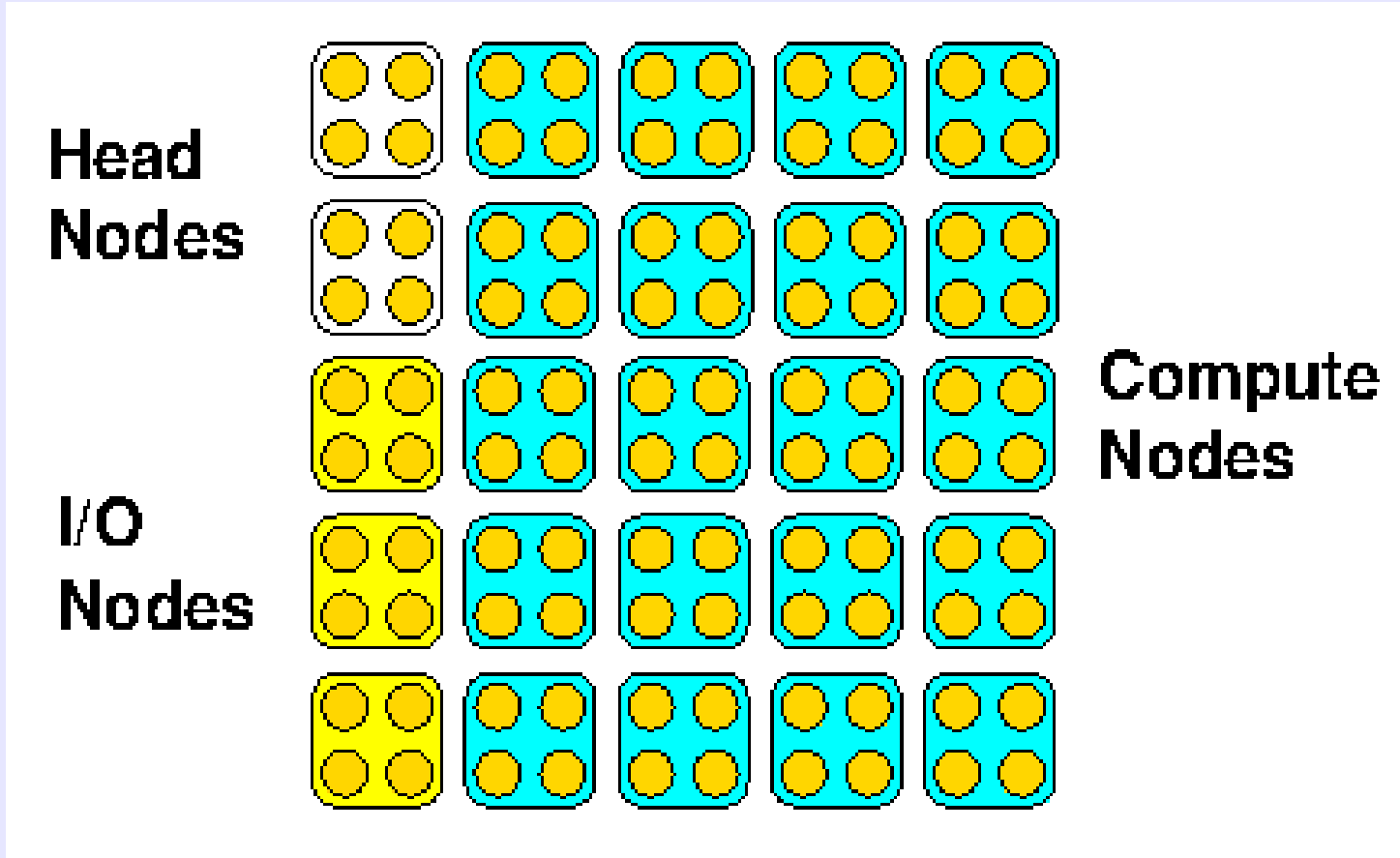
On typical cluster, the work must be done on the compute nodes. We could submit multiple single node job scripts, but how about using more than one node at a time?

Multi-Node Considerations

- The ***mother superior*** node is only one with all the job information, like environment variables.

- The programs on other nodes can be started with remote shell commands, like ssh.

- Must account for shared or local file systems.

- Must assure all programs finish before script exits.

- Must be aware of efficiency (load balancing).

8 Serial Programs on 2 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

- 8 programs on 2 nodes clearly starts to 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

<div style="float:left">

**Node 1
Mother Superior
4 Tasks**

**Node N
Compute Node
4 Tasks**

</div>

```
# 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.

- Involves basic shell programming, and maybe gets involved with fussy quoting rules to get everything passed correctly.

- It 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 wall time 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?

- A work distribution system (distributor/worker model). WQ roughly stands for "Work Queuing".

- Handles **_tasks_** – a task is defined as the work necessary to process one input file.

- Multiple **_workers_** execute the tasks - one worker for each simultaneous task desired all nodes.

- Workers request a task from the **_distributor_**.

- Workers and the distributor share elapsed time information, estimate remaining time, and won't start a new task if insufficient time remains.

Design Assumptions

- ***Task*** viewed as work needed to process one input file.

- The **distributor** manages ***a list of input file names***, and hands out one at a time.

- A ***work request*** includes run time of last task executed.

- A ***work handout*** includes longest known task run time.

WQ Components

- **wq.py** – A Python script that implements the **distributor** and **workers** (as they say, *no user serviceable parts inside*!).

- **wq.pbs** – A PBS batch script template with a few user required variable settings and most of the magic cooked in.

- **wq.sh** – A user created script (could be a program) that accepts an input file name as it's only argument.

- **wq.list** – A user created file containing input file names, one per line (suggest using absolute path names).

The names of files can be changed – just keep consistent with the contents of the PBS script – changing name of **wq.py** would require a bit more work than changing any of the other 3.

wq.pbs : PBS Preamble Section

The PBS script is divided into 2 parts: the WQ prologue (which includes the PBS preamble), and the WQ epilog. Only prologue contains items the user should adjust. The PBS preamble portion should look familiar (not all PBS options are shown here):

```
#! /bin/bash
#################################################
#
# Begin WQ Prologue section
#################################################
#
#PBS -A hpc_myalloc_03
#PBS -l nodes=4:ppn=4
#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

```
# Have the script run 1 worker per core on each node assigned.
# The system to run on has 4 cores per node (i.e. ppn=4):

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
```

1

2

3

4

5

wq.pbs : Epilogue Section

- Serious magic happens in the Epilogue section – so dabble with at your own peril.

- Does some sanity checking of settings.

- Determines if running as mother superior.

- Mother superior preps information exchange process and starts job script on all other compute nodes.

- Mother superior starts dispatcher, and it's workers.

- Compute nodes start required number of workers.

- All workers start the ***request - execute - report*** cycle until time runs out or 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 name. 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 file:

```
$ ./wq.sh filename
```

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

wq.list

This is nothing more than a file containing input file names, one per line. For a really large number of input files, generate it 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).

Some examples are in order.

A Serial Example

A Simple **wq.sh**

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

```
#! /bin/bash

# First, some basic processing of the input file name.

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

# Now just echo the results.

echo "DIR=${DIR}; BASE=${BASE}"
echo "That's all, folks!"
sleep 2
```

⬅ backticks NOT single quotes!

A real-world script can be arbitrarily complex!

XSEDE
Extreme Science and Engineering
Discovery Environment

An Input File List

Let's look for files with .bf extensions:

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

And assume it produces 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**

We can try to run on 32 cores. If the system has 16 cores per node, we would need to request 2 nodes. The PBS preamble option would look like:

#PBS -l nodes=2:ppn=16

Now we just need to make the 5 PBS prologue variables agree:

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

Serial Example STDERR

```
Dispatcher:Start:1
mike044_0:Taking:1:0.00:90.00
. . .
mike045_15:Taking:32:0.00:90.00
Dispatcher:Maxtime:mike044_6:2.03:1393524348.06
. . .
mike045_0:Taking:81:4.03:85.97
Dispatcher:Shutdown:32
Dispatcher:Last:82
mike045_1:Taking:82:4.03:85.97
```

Dispatcher – event, task number involved, event data
Worker ID – action, task number involved, start time, remaining time.

Serial Example STDOUT

Each worker emits a report using 4 groups of lines as so:

```
Task:7:mike044_6:Ran:True:/work/user/wq_timing.sh /work/user/chr13/chr13_720.bf
Timings:7:mike044_6:1393524346.03:1393524348.06:2.03:2.03
Stdout:7:
  DIR=/work/user/chr13; BASE=chr13_720.bf
  That's all, folks!
Stderr:7:
```

- Task – task number, worker ID, action, success, task command
- Timings – task number, worker ID, start time, end time, elapsed, max.
- Stdout – task number, followed by task stdout lines produced, if any.
- Stderr – task number, followed by task stderr lines produced, if any.

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.

XSEDE
Extreme Science and Engineering
Discovery Environment

Multi-Threaded Example **wq.sh**

Shortcut

 # Threads

Keep It Readable

Production vs Testing

```bash
#! /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=4

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

# Build the rather complex command line.

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
```

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
```

Multi-Threaded Example STDERR

```
Dispatcher:Start:1
mike111_0:Taking:1:0.00:252000.00
mike111_1:Taking:2:0.00:252000.00
mike111_2:Taking:3:0.00:252000.00
mike111_3:Taking:4:0.00:252000.00
Dispatcher:Maxtime:mike111_1:64223.42:1393836959.93
mike111_1:Taking:5:64223.42:187776.58
Dispatcher:Maxtime:mike111_2:123837.43:1393896573.95
mike111_2:Skipping:6:123837.44:128162.56
Dispatcher:Timeup:mike111_2:1393896573.95
Dispatcher:Shutdown:3
Dispatcher:Last:6
```

Note Task 6 was last handed out, but mike111_2 skipped it because to little time remained to safely start it and finish.

XSEDE
Extreme Science and Engineering
Discovery Environment

Multi-Threaded Example STDOUT

```
Task:3:mike111_2:Ran:True:/work/jalupo/WQ/BLASTN/wq_blastn.sh
/work/jalupo/WQ/BLASTN/GS049.blast/xai.fna

Timings:3:mike111_2:1393772736.52:1393896573.95:123837.43:123837.43

Stdout:3:

Stderr:3:


Task:6:mike111_2:Skipped:False:/work/jalupo/WQ/BLASTN/wq_blastn.sh
/work/jalupo/WQ/BLASTN/GS049.blast/xaa.fna

Timings:6:mike111_2:-1.00:-1.00:-1.00:123837.44

Stdout:6:

  Insufficient Time

Stderr:6:

  Time left: 128162.56; Max Time: 123837.43; Margin: 1.25
```

An MPI Example

Adjust For MPI

- **wq.sh** – set up for small number 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

```bash
#! /bin/bash

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

# Assume we want 16 processes per task (i.e. 1 worker per node).

PROCS=16
HOSTNAME=`uname -n`
HOSTLIST=""
for i in `seq 1 ${PROCS}`; do
    HOSTLIST="${HOSTNAME},${HOSTLIST}"
done
HOSTLIST=${HOSTLIST%,*}

CMD="mpirun -host ${HOSTLIST} -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**

```bash
#! /bin/bash
###############################################################################
# Begin WQ preamble section.
###############################################################################
#PBS -A hpc_enable02
#PBS ... Other Settings ...

# Set number of workers per node:

WPN=1

# Set the working directory:

WORKDIR=/work/user

# Name of the file containing the list of input files:

FILES=${WORKDIR}/wq.lst

# Name of the task script

TASK=${WORKDIR}/wq_mb.sh

START=1


###############################################################################
# End WQ preamble section.
```

MPI Example STDERR

Only the end of the file looks interesting:

```
. . .
mike333_0:Taking:513:15336.63:2663.37
mike241_0:Taking:514:15367.58:2632.42
mike331_0:Taking:515:15401.42:2598.58
mike322_0:Skipping:516:15459.29:2540.71
Dispatcher:Timeup:mike322_0:1393617821.63
Dispatcher:Shutdown:31
Dispatcher:Last:516
```

Last task handed out was number 516, but looks like mike322_0 skipped it. The Dispatcher expects to order 31 other workers to stop the next time they request a task.

MPI Example STDOUT

The stdout file really doesn't change much, though the file paths are long.

```
Task:513:mike333_0:Ran:True:/work/user/wq_mb.sh
/work/user/PostPredYeast/YHL004W_DNA/SeqOutfiles/YHL004W_DNA.ne
x.run3_2868000/GTRIG.bayesblock
Timings:513:mike333_0:1393617699.01:1393618496.73:797.72:16134.
34
Stdout:513:
Stderr:513:

Task:515:mike331_0:Ran:True:/work/jalupo/WQ/MrBayes/wq_mb.sh
/work/jalupo/WQ/MrBayes/PostPredYeast/YHL004W_DNA/SeqOutfiles/Y
HL004W_DNA.nex.run3_4180000/GTRIG.bayesblock
Timings:515:mike331_0:1393617763.78:1393618516.08:752.30:16153.
72
Stdout:515:
Stderr:515:
```

XSEDE
Extreme Science and Engineering
Discovery Environment

Load Balancing Issues

- The more uniform the runtimes are across all tasks, the more likely a job will end gracefully.

- Take a look at the concepts.

- Illustrate potential problem.

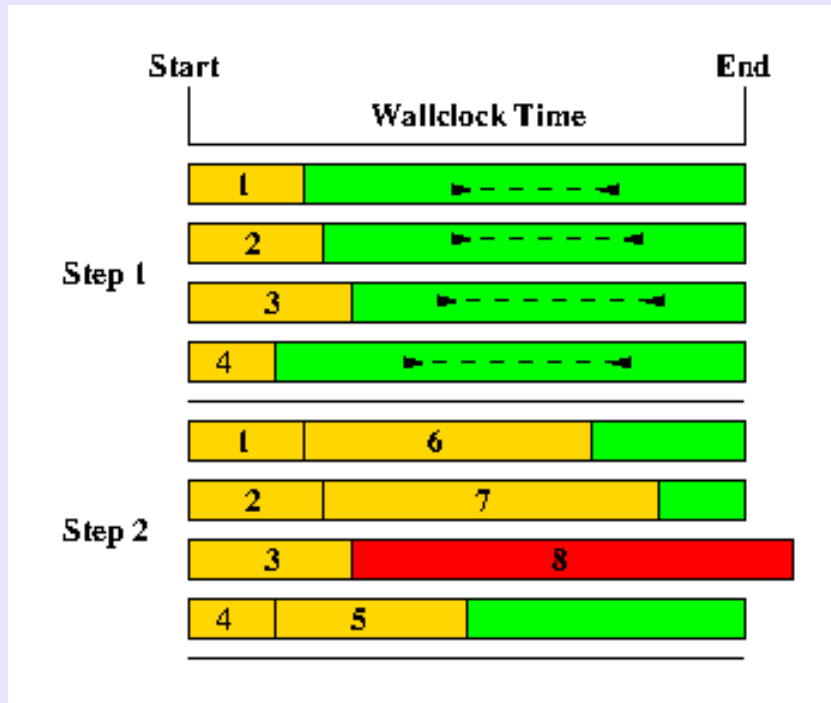- Discuss how to analyze a job's efficiency.

Task Run Times

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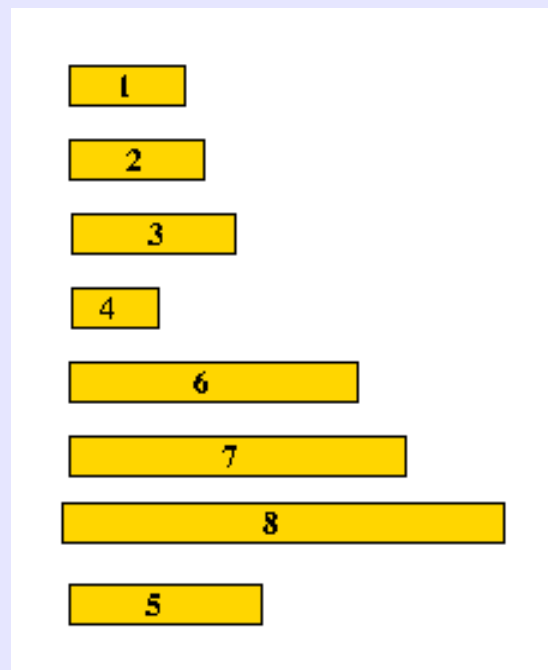
Insufficient Walltime

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 sure jobs complete gracefully.

- Output format allows analysis.

Monitoring and Statistics

Output Format Rational

- The colon-delimited format and seemingly redundant data items was intentional.

- Gawk, Perl, Python, and other scripting languages have easy facilities for parsing lines with character delimited fields.

- A few examples illustrate the idea.

Identifying Failed Tasks

The Task lines in the output file have the following format (fields are numbered):

1   **2**   **3**   **4**   **5**   **6**

**Task:1414:mike104_0:Ran:True:/yada/script
/data/path/filename**

That means we just have to find the "Task" lines, check field 5 for True/False, and output the file name (i.e. could use to generate a new input list). Consider using Gawk:

```gawk
#! /bin/gawk -f

BEGIN { FS = ":"; }

/^Task/ { split($6,task," ");
          if ( $5 == "False" ) failed[++n] = task[2]; }

END {
  print n, "files were not processed.\n" > "/dev/stderr";
  for ( i = 1; i <= n; i++ ) printf( "%s\n", failed[i] );
}
```

XSEDE
Extreme Science and Engineering
Discovery Environment

Timing Jitter

- Use Python to find the timing jitter between the longest and shortest running workers using Timing records. Field 5 is the walltime the task ended, so we just need to find the minimum and maximum end times.

1	2	3	4	5	6	7

`Timings:3:mike243_0:1406673580.51:1406673582.52:2.02:2.02`

```python
#! /bin/env python
import sys

f = open( sys.argv[1], 'r' )
raw = f.readlines()
f.close()
worker = {}
max_end = 0.0
min_end = 1.0e+37

for l in raw:
    u = l.strip().split(':')
    if u[0] == 'Timings' :
        t = float(u[4])
        if t > 0.0 :
            worker[u[2]] = t

for k, v in worker.iteritems() :
    if v < min_end :
        min_end = v
    if v > max_end :
        max_end = v

print( "End times between %.2f and %.2f - %.2f sec spread." %
       ( min_end, max_end, max_end - min_end ) )
```

Efficiency

- Important because job charges are based total walltime (in hours):

$$SU = cores * t_{long}$$

$$Eff_{ave} = \frac{t_{long} + t_{short}}{2 * t_{long}}$$

$$Eff_{actual} = \frac{\sum t_{task}}{cores * t_{long}}$$

- Want uniform end times. If one task keeps many nodes idle waiting to end, not good!

t_{long} = longest worker walltime
t_{short} = shortest worker walltime
t_{task} = individual task walltime

Eff_{ave} = average parallel efficiency
Eff_{actual} = actual parallel efficiency

XSEDE
Extreme Science and Engineering Discovery Environment

Examples

- Consider a job that runs 64 tasks on 64 cores.
- Well balanced, min of 69 hrs, max of 71 hours:
  - 98.6% ave efficiency.
- Highly imbalanced, 63 finish in 1 hour, 1 finishes in 71 hours.
  - 50.7% ave efficiency.
  - **2.9%** actual efficiency.

Resources

- Moodle version at:
  - http://moodle.hpc.lsu.edu
  - HPC121 – Background and Distribute Tasks
  - Distribution zip archive file provided.
- On SuperMike-II:
  - /work/jalupo/WQ/WQ-20140801-r145.zip

Hands-On Test

1. Need +Python-2.7.3-gcc-4.4.6 in .soft

2. Create a directory.

3. unzip /work/jalupo/WQ/WQ-20140801-r145.zip

4. cd Examples/Timing

5. cp ../../wq.py .

6. chmod u+x wq_timing*

7. Edit wq_timing.pbs: set PBS output directory, proper allocation, and work directory.

8. Generate a proper file list:

   1. rm 82_file_list

   2. for i in `seq 1 82` ; do echo "`pwd`/file$i" >> 82_file_list; done

9. Submit the PBS file: qsub wq_timing.pbs > jid

XSEDE
Extreme Science and Engineering
Discovery Environment

Hands-On Test Results

When job completes, the directory will look like this:

```
$ ls
82_file_list     jid           wq.py                 WQ_Timing.o228557
wq_timing.sh
hostlist.228557  pbs.228557  WQ_Timing.e228557  wq_timing.pbs
$
```

jid . . . . Job ID (228557 in this case).
hostlist.228557 . . . Job specific host list.
pbs.228557   . . . . . PBS job script copy.
WQ_Timing.e228557 . . stderr (note use of job name and jobID).
WQ_Timing.o228557 . . stdout (note use of job name and jobID).

XSEDE
Extreme Science and Engineering
Discovery Environment

Final Suggestions

- Organize data files in meaningful way.
  - More than 10,000 files per directory will cause performance degradation.
- Use Linux tools to generate file lists.
- Stay aware of job performance.