

## WQ — A Task Distributor: The User Manual

Document Date: 6 Aug 2014

Program Version: SVN Revision 145

James A. Lupo

jalupo@cct.lsu.edu

Center for Computation & Technology

Louisiana State University

Baton Rouge, LA

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/us/>.

## Acknowledgments

WQ was motivated by the needs of Jeremy Brown and Cameron Thrash, two professors in the LSU Biology Dept. They had to process large numbers of input files through the same analysis tool and required a way to easily distribute the work across multiple nodes with possibly varying core counts per task. Vinson Doyle, a post-doc, helped considerably in refining the tool as his file counts per run began to exceed 25000 - most effective for stress testing. Their willingness to act as friendly users (alpha/beta testers?) to shake out quirks and motive refinements in the ease of use is greatly appreciated.

## WQ Introduction

### 1 Motivation

WQ is a task distributor for HPC clusters, sort of a private work queuing system under user control. The types of task supported may be serial or parallel, with the restriction that parallel tasks are required to use only the processing cores on one node. The motivation is due to recent trends in HPC cluster architectures which has been towards ever higher core counts per node. At the same time, there is an increasing number of non-traditional users who still rely on serial jobs, and in many cases a great number of them. Launching many independent jobs under a batch submission system can be very monotonous, and may require modestly advanced skills at shell scripting. If you can imagine the difference between setting up 1000 serial jobs with 1 task each version 1 job working on 1000 tasks, you can see the potential difficulties. WQ is designed to facilitate a common use case: run the same application with many different input files. You might imagine how such a case makes explicit scripting rather daunting from a repetition viewpoint, yet the effort is too simple to turn over to a heavy weight solution, such as SAGA[3] or ManyJobs[1], though GNU Parallel[2] would be a close competitor.

The WQ tool set is designed for simplicity, but not at the expense of flexibility. There are 3 pieces to master, only 2 of which require customization by the user to meet specific requirements. Most of the magic involved in multi-tasking is hidden by the tools, but some understanding of the workings of job batch systems and basic shell scripting is still required to make the pieces work. What follows describes the 3 files that make up the WQ tool set, along with examples involving a toy serial task, a multi-threaded OpenMP task, and a small MPI task.

### 2 Some Definitions

The WQ system uses a “dispatcher-worker” model of distributed computing. The current implementation is targeted at systems running PBS (Portable Batch System) under Maui/Torque, so much of the nomenclature and the script internals are slanted towards that environment. It’s important to understand the definition of terms used to facilitate usage and support porting to other job managers:

**job** All of the processing on computational resources assigned by the PBS scheduler under the control of a master script. Normally one PBS script defines one job in the system. PBS assigns the job to a portion of a machine for exclusive use by a user for a specific amount of time. The user may have the nodes do any (well, almost any) type of processing desired.

**task** A unit of work involving a single command or input file that is completely independent of any other processing.

**process** Formally, the running instance of an application. This may be the application used by a single task. The task may be a purely serial program running on 1 core, a multi-threaded program using several cores, or a small MPI program restricted to the cores of a single node.

**walltime** Time determined using a conventional time-of-day clock. It is used to track elapsed time irrespective any system times, such as CPU time, or I/O time, that is recorded by the operating system.

**dispatcher** The process which hands out one task at a time to a worker upon request. It supports sharing of other information, such as the walltime taken by the longest running task so far, to aid in time management and allow graceful shutdown if there appears to be insufficient time to start and complete a task before the job’s walltime runs out.

**worker** The process which controls the execution of tasks, one at a time. A worker requests a task from the dispatcher, executes it, then requests another task. This continues until all tasks are completed, or there appears to be insufficient time to start and complete a task before the job walltime runs out.

**head node** The front end, management, or log-in node of a HPC cluster.

**mother superior node** The node, which is one of several assigned to a job, on which execution of the job script begins. It is special in the sense that much of the job information is made available only to it, and the information must be explicitly passed to the other nodes if they require it. The WQ PBS script takes pains to make sure the dispatcher process is started on the mother superior.

**compute node** A node assigned by the scheduler to a user's job. Technically, the mother superior is a special compute node. WQ runs 1 or more workers on each of the compute nodes (including the mother superior), depending on user requirements and task configuration.

### 3 Overview

When a job is started under PBS, the requested number of nodes are assigned to the user, and the PBS script starts executing on the mother superior node. The mother superior is special among the compute nodes because it has access to environment variables created for the job by the scheduler. In particular, all the assigned node names are listed in a file pointed to by the shell variable `PBS_NODEFILE`, a unique job identifier is provided in `PBS_JOBID`, and the amount of wall time requested is provided in `PBS_WALLTIME`. Should the other compute nodes need access to this information, it is up to the mother superior to provide it to them.

Because the mother superior is the only compute node aware of the job script, the WQ script actively detects if it running on the mother superior and does some setup work. This setup includes making copies of the job script and the host file so they are visible to all workers. The next step involves starting up the dispatcher so it is ready to respond to worker requests. Only then does it start up workers on all compute nodes, including on the mother superior. When the WQ script executes on a worker node, it simply gathers the information prepared by the mother superior and starts executing its workers. The workers request and execute tasks until the tasks are exhausted, or they are told to stop by the dispatcher because time is running out and any new tasks may not complete.

The information provided to the dispatcher at start-up includes two file names. The first is treated as the command to be run by the worker. The command is expected to process exactly 1 command line argument. The command can be an application or script following the usual command line conventions. The examples shown below all use shell scripts because of the pre- and post-processing capabilities and extra processing control they enable over a single application command.

Each line of the second file is treated as an argument to be passed to the command. Basically each task involves handing out 1 file name to the command, and the command script can then do what it will with the name. In the generic use case, the command is a shell script, and the file names would be absolute path names to input data files. Thus, the task executed by the worker would be a shell command line that looks like:

```
$ command filename
```

In fact, this command line should work correctly if one types it manually and provides a proper file name. Such testing is highly recommended before committing to a production job. Once the worker completes a task, it emits a report containing the execution status code, all text written to standard output (`stdout`), and any text written to standard error (`stderr`). If the application produces a high volume (more than 10 lines??) of output on `stdout`, it may be best to have it written out to a separate output file rather than return it through `stdout` or `stderr`.

There are just 3 files important to the process of running a job. The heavy lifting is done by `wq.py`, which provides for the dispatcher and worker services, depending on how its run; `wq.pbs` is the job submission script; and `wq.sh` is the task command script.

**wq.py** The main python script which is used to create workers, or the dispatcher, depending on how it is executed. As they say, there are no user serviceable parts in this file. It does provide the workers with one special bit of information, and that is how much time was requested for the job. The workers share the

longest task times amongst themselves (with the help of the dispatcher), and do not start new tasks if the time remaining is less than the maximum time plus a small safety margin. (So, okay, one could go in and change the safety margin from 25% to something else.) If the requested job time can't be determined, 1 day (86400 secs) is used (Okay, okay. There are two items a user might change).

**wq.pbs** This is a typical PBS batch script. The name is not significant, and the contents could be revised to fit a different mode of using WQ. It begins with a prologue section with the usual PBS options, such as user and allocation information, desire job queue, and other typical PBS settings. The user does have to pay attention to a few shell variables that control the number of workers started per node. Basically, the number of cores used by all the workers on a node should be less than or equal to the number of cores available. The user is responsible for assuring consistency between user settings in this script and the task execution script as far as core utilization is concerned. Below the prologue is the WQ specific section that takes care of launching the task dispatcher and all of the workers.

**wq.sh** This is the task script that the workers run to actually accomplish the work. Again, the name is not significant, but core usage should be consistent with whatever appears in the PBS batch script. It is a shell script (or other program) that expects a single command line argument, and controls the processing for a single task. A typical use case would treat the argument as an absolute path name to an input file. The path could then be processed to find the working directory, create output names, or any other pre-processing the user finds necessary. The number of threads or number of MPI processes started by this script, times the number of workers per node, should equal the number of cores available on the node.

Since the contents of `wq.pbs` and `wq.sh` tend to vary with the application run, it is easiest to discuss them in the form of examples later on. The main file is a Python script, `wq.py`, and is responsible for dispatcher and worker services, it will be discussed in detail next.

## 4 Running wq.py

Like most good user applications, the `-h` or `--help` command line options can be used to output a short help message on how to use `wq.py`. There are 3 different modes to run the program in, and that is reflected below by 3 different usage lines. Note that all of the available options come in either a short, 1 letter form, or in longer word form:

```
$ python wq.py --help
```

```
Usage:  python wq.py -h[--help]
        python [-s[--start] task_num] -d[--dispatcher] cmd \
            -a[--allworkers] n -i[--input] filenm
        python -w[--workers] n -m[--mothersuperior] ms [-t[--time] walltime]

Help display:
    -h,--help ..... Display this help message.

Run as dispatcher:
    -s,--start task_num .. Task number to start with. Represents the line
                           number in the input list file. Default is 1.
    -d,--dispatcher cmd .. Run as the dispatcher for the command cmd, where
                           cmd is the command or script to use for the task.
                           cmd will be called with a single file path as
                           it's only argument.
    -i,--inputs filenm ... Name of file containing input file names to
                           serve as inputs to cmd, one per task.
    -a,--allworkers n .... Total workers (workers per node * nodes ).

Run as worker:
    -w,--workers n ..... Run n workers per node.
```

```
-m,--mothersuperior ms ... Host name of mother superior node.
-t,--time walltime ..... Wallclock time to allow for entire job.
                           May be expressed as one of the following:
                           ss, mm:ss, hh:mm:ss, or d:hh:mm:ss.
                           (note: Torque sets env variable PBS_WALLTIME)
```

The dispatcher must be started before any of the workers.

The default worker jobtime is hardwired to 86400 secs - 1 day.

Revision: \$Id: wq.py 145 2014-08-06 18:30:30Z jalupo \$

Note that the same Python script, `wq.py`, is used to start either the dispatcher (which should be done first) or the workers. The `wq.pbs` script takes care of this by starting the job dispatcher on the mother superior first. It then sees to starting workers on all the compute nodes, and finally starts a set of workers on the mother superior. The name of the mother superior is passed to all the workers so they know which node to contact in order to talk with the dispatcher. That's pretty straight forward.

The requested job walltime is also passed to the workers. This allows them to track how long each task takes, and the information about the longest task run time is shared via the dispatcher with all the workers. The workers use this time in an attempt to be smart about the remaining job time and thus avoid starting new tasks if they estimate there is insufficient time remaining to complete it. This is worth talking about in more detail in the next section.

This document, and WQ itself, is under active development and will continue to evolve. Thus you may have noticed that the script reports its version number. This will make it easier to track what is being used if problems are reported.

## 4.1 Runtime Considerations

The ability to stop processing before job time runs out is predicated on relatively uniform run time, and knowledge of how long the tasks are taking. Each time a worker requests a task, it sends along the longest task time it is aware of to the dispatcher. The dispatcher keeps track of the longest time it sees from the various works, and sends it along with the task assignment when answering requests. This way workers can decide if there have sufficient time to complete a task before the job time runs out. If a worker estimates there is insufficient time, it notifies the dispatcher that it is skipping the assignment because of insufficient time. The dispatcher then stops handing out new work, and instead responds to new work requests with a terminate command. This isn't perfect, as a worst case ordering of the tasks may place the longest running task at the very end of mostly very short ones. The only protection is to make use of any run time estimates possible based on the input and try to order the input to put longest running tasks first.

To see what this all means, consider a simple case of running 4 tasks at a time. Assume there are 8 tasks total, represented in Figure 1, where the length of each bar corresponds to how much walltime the task will take. Further assume that the walltime needs aren't known in advance.

The first execution scenario is one in which things work as expected. Namely, the time calculations stop processing before the job runs out of walltime. Figure 2 shows how this scenario plays out. Tasks 5–8 are the first to be assigned (this assumes their input files happened to be the first 4 in the input file list). Lets assume Task 5 completes first. As far as the worker knows, the time it took is the maximum task time, so it computes how much time to allow for the next task based on this info. The time it comes up with is shown as the dotted bar. Since it fits within the remaining wallclock time, the worker requests another task, and so starts working on Task 1. Task 6 is the next to finish, since it took longer, its time becomes the maximum task time to use to estimate remaining time. In fact, the dotted bar shows the estimated time exceeds available time, and the worker stops processing. Things only get worse, as far as available time is concerned, so in fact, no other worker starts any other tasks. The job finishes properly, with 5 tasks completed, and 3 left to process in a second run. The

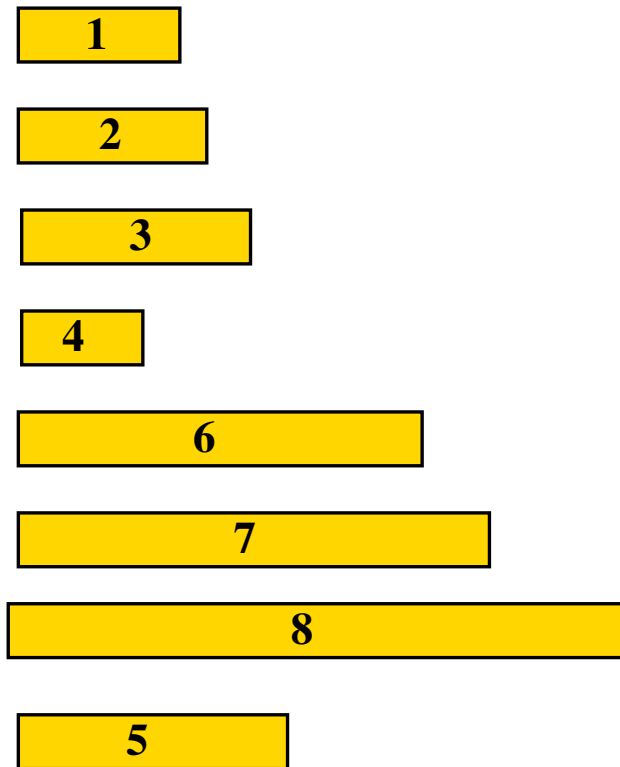


Figure 1: A collection of 8 tasks. The required walltime for each is proportional to the length of each task's bar. The box numbers are meant for identification and not necessarily the order in which the tasks will be assigned.

lesson here is that longer running jobs should be among the first to execute, if that's possible to determine. In reality a large set of tasks with reasonable spread in run times are likely to have sufficient long running task complete so process works as planned.

What might go wrong? Let's look at the scenario shown in Figure 3. In this case, the 4 shortest running tasks (1–4) are started first. By the time the worker completes Task 3, the only task left for it is Task 8, which happens to be the longest running. However, it sees the time it took on Task 3 to be the longest task time, so even with the safety margin built in, it estimates there will be time left so starts Task 8. Task 8 actually exceeds the available walltime, and will be killed by the system job manager. More tasks are done, but the non-graceful exit will result in incomplete output, and may require manual examination of the output files to determine what really succeeded and what did not.

The best recommendation is to know how long the tasks will take based on values from the input file and apply some judicious reordering. The next best thing is to allow more than 2 or 3 times the longest anticipated task for the entire job. The job will only be charged for the time actually used, so the downside may be a possible delay in job start due to the length of time requested.

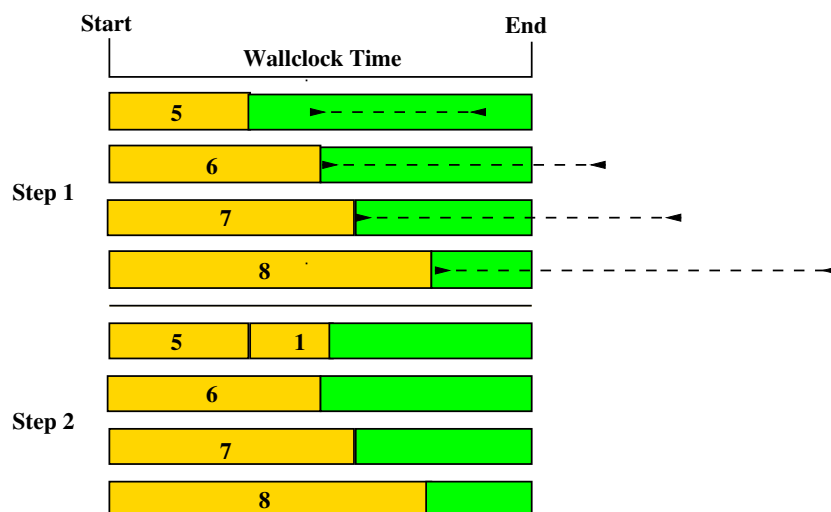


Figure 2: A successful job scenario. Step 1 shows the first set of tasks to be executed. The green boxes indicate the actual walltime remaining. The dashed lines indicates the maximum walltime to allow for the next job, based on what has completed so far.

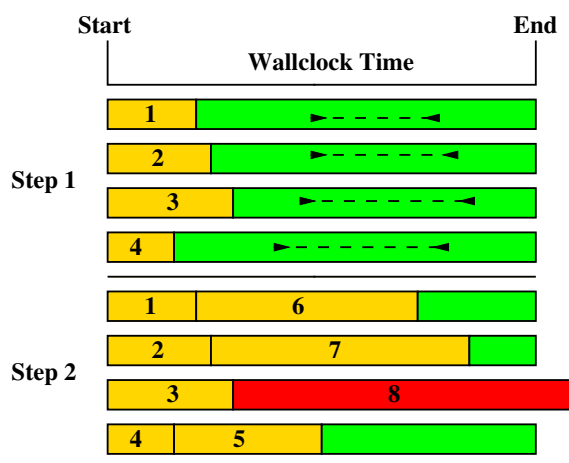


Figure 3: A failed task/job scenario. Step 1 shows the first set of tasks to be executed. The green boxes indicate the actual walltime remaining. The dashed lines indicates the maximum walltime to allow for the next job, based on what has completed so far. The red box indicates a task that was started based on a too-short estimate of remaining walltime.



## 4.2 The PBS Script

The WQ PBS script, `wq.pbs`, has two main parts which will be referred to as the *prologue* and *epilogue*. The prologue, or beginning, section contains things the user should configure, such as job options and work directories. The epilogue, or remaining, section contains all the scripting needed to setup and start the dispatcher and workers. Normally, the epilogue section requires no changes. We'll look at each before digging into some examples.

### 4.2.1 The Prologue

List 1 shows the prologue lines from the PBS script. Anyone familiar with PBS should recognize the first 5 lines which set the allocation, or account, code, asks for 16 nodes with 16 cores per node, a wall time of 30 minutes, use of the *workq* queue, and with a job name of *WQ\_Test*. They should be treated as representative as some are optional and some required - with dependence on local requirements.

Listing 1: WQ PBS Script Prologue

```
1  #! /bin/bash
2  #####
3  # Begin WQ prologue section.
4  #####
5  #PBS -A my_allocation
6  #PBS -l nodes=16:ppn=16
7  #PBS -l walltime=00:30:00
8  #PBS -q workq
9  #PBS -N WQ_Test
10
11 # Things that should be customized, carefully of course.
12
13 # Set the desired number of workers per node. This is basically the
14 # number of cores available on a node divided by the number of
15 # processes/threads that will be used per task (i.e. 4 MPI processes
16 # per task on a 16-core node would allow for 4 workers). Let's assume
17 # 2 threads per task, so:
18
19 WPN=8
20
21 # Set the working directory:
22
23 WORKDIR=/work/someone/important/data
24
25 # Name of the file containing the list of input files (not real
26 # imaginative):
27
28 FILES=${WORKDIR}/wq.lst
29
30 # Set the starting line in the file. This allows you to skip over
31 # previously completed tasks. The default is 1 (i.e. start from the
32 # beginning).
33
34 START=1
35
36 # Name of the task script each worker is expected to run to process
37 # the files sent to it as the only command line argument.
38
39 TASK=${WORKDIR}/wq.sh
40
41 #####
42 # End WQ prologue section.
```

The PBS section is followed by shell commands which set 4 environment variables used elsewhere in the script.

**WPN** How many workers to start per node. The number of workers per node times the number of cores per task

should equal the number of cores on the node (i.e. PPN). The number used here must be consistent with the settings used in the task script file.

**WORKDIR** The working directory to use while the job runs. Several files are created in the process of setting up for the run, and all must be accessible to every node.

**FILES** A file containing the list of input data files to be processed. Each line should be treated as the path to one file.

**START** Indicates the starting line, or record, in the input file. Since a job that stops because it ran out of time may not complete all tasks, this allows one to quickly run a following up job to complete the rest.

**TASK** Name of the script or program that will be run for each task. It will be called with a single argument, that being one of the input file names.

As a final note, given the way `stderr` and `stdout` are handled, it is best to allow them to appear in separate files rather than use the PBS `-j` option to merge them. It is okay to assign names, if you wish, or just accept the default names created using the job name assigned.

## 4.2.2 Epilogue Section

The bulk of the job setup is done in the epilogue section. Listing 2 shows how the launching of the dispatcher and workers is managed. It relies on the fact that only the mother superior is running the script when the job starts, making it easy to detect what role the node should play. The mother superior is also the only node to have access to the PBS environment variables, thus some important values must be explicitly passed to the other nodes when starting the workers.

Listing 2: WQ PBS Script Epilogue Section

```
1 #
2 # Begin WQ epilogue section.
3 # What follows is the main WQ script. It should be considered powerful
4 # magic. Dabbled with at your own peril.
5 #####
6
7 # Drop into the working directory after making sure it exists.
8
9 if [ ! -d ${WORKDIR} ] ; then
10     echo "WQ.PBS.Error: _WORKDIR=_\"${WORKDIR}\" _does_not_exist!"
11     exit 1
12 fi
13
14 cd ${WORKDIR}
15
16 # Only the mother superior has PBS_JOBID defined, so we will be
17 # passing it to the other nodes as $2. Use this fact to decide if
18 # we are running on the mother superior or a compute node:
19
20 if [ "${2}" = "x" ] ; then
21
22     # Must be running on the mother superior. Do some basic sanity
23     # checking just to be safe.
24
25     if [ ! -r ${FILES} ] ; then
26         echo "WQ.PBS.Error: _FILES=_\"${FILES}\" _does_not_exist_or_can't_be_read!"
27         exit 1
28     fi
29
30     if [ $(wc -l ${FILES} | cut -d ' ' -f 1) -lt 1 ] ; then
31         echo "WQ.PBS.Warning: _FILES=_\"${FILES}\" _is_empty. _No_work_to_do!"
32         exit 0
33     fi
```

```

34
35  if [ ! -x ${TASK} ] ; then
36      echo "WQ.PBS_Error: TASK=\`${TASK}\` does not exist or isn't executable!"
37      exit 1
38  fi
39
40  if [ ${START} -lt 1 ] ; then
41      echo "WQ.PBS_Error: START can't be less than 1! Quitting!"
42      exit 1
43  fi
44
45  # Remember our host name.
46
47  MS=`uname -n`
48
49  # Use a bit of magic to strip off the trailing host name and
50  # leave only the job number from PBS_JOBID:
51
52  JOBNUM=${PBS_JOBID%.*}
53  HOSTLIST=${WORKDIR}/hostlist.${JOBNUM}
54
55  # We want the mother superior host name first. So, take the host
56  # list provided, sort it into a unique list of names, with MS first.
57  # This assures it's node ID, or position in the hostlist, is 1.
58
59  echo ${MS} > ${HOSTLIST}
60  grep -v ${MS} ${PBS_NODEFILE} | uniq | sort >> ${HOSTLIST}
61
62  # Compute the number of nodes assigned.
63
64  export NODES=`wc -l ${HOSTLIST} | gawk '/{ print $1}`'
65
66  # Make a local copy of the PBS script since only the mother superior
67  # can see it at job start.
68
69  JOBFILE=${WORKDIR}/pbs.${JOBNUM}
70  cp $0 $JOBFILE
71  chmod a+x ${JOBFILE}
72
73  # Mother superior must start up the dispatcher, so:
74
75  python ${WORKDIR}/wq.py --start $START --dispatcher ${TASK} \
76      --inputs ${FILES} --allworkers $(( $WPN * $NODES )) &
77
78  # Give it a chance to spin up since the dispatcher must be ready
79  # to accept connections from the workers upon request.
80
81  sleep 5
82
83  # Ready to start the script on all compute nodes. This will fire up
84  # workers. We'll pass PBS_WALLTIME and the job number as arguments.
85  # They'll connect to the dispatcher and start work immediately.
86
87  for H in `cat ${HOSTLIST}` ; do
88      if [ ${H} != ${MS} ] ; then
89          ssh -n ${H} ${JOBFILE} ${PBS_WALLTIME} ${JOBNUM} &
90      fi
91  done
92
93  # Finally, mother superior can also start workers:
94
95  python ${WORKDIR}/wq.py --workers ${WPN} --mothersuperior ${MS} \
96      --time ${PBS_WALLTIME}
97
98  # Make sure to wait until all the processes are done!
99
100  wait
101
102  else
103

```

```

104  # Must be running on a compute node. The job number was passed by the
105  # mother superior (see above).
106
107  HOSTLIST=${WORKDIR}/hostlist.$2
108
109  # Now, we have to get the name of mother superior from the host
110 # list. Thats so we know where the dispatcher is running. Simply
111 # grab the first entry from the hostlist file and press on.
112
113  MS='head -1 ${HOSTLIST}'
114
115  # Ready to go. Spin up the workers. The mother superior passed
116 # the job wall time as argument 1 when the script is called, so
117 # we have all the values needed for workers:
118
119  python ${WORKDIR}/wq.py --workers ${WPN} --mothersuperior ${MS} \
120      --time $1
121
122  fi
123
124  # Give a bit of time to make sure dispatcher has shut down cleanly.
125  # The WAIT above should allow for this, but coming down on the side of
126 # paranoia:
127
128  sleep 2

```

If you read through the epilogue, notice that the mother superior starts the dispatcher and remote workers in the background, but starts its own workers in foreground. When the workers are done, the script is essentially complete. But a wait is added at the end of the mother superior section because the remote workers may not be done yet, hence the dispatcher is still running. It makes sure all background tasks have completed before the script proceeds.

At this point it is useful to discuss several examples of using `wq.py`. The 3 examples that follow illustrate how purely serial jobs, small process count MPI jobs, and multi-threaded OpenMP jobs are handled. Since only the prologue must change as user information and tasks change, it will be the only PBS script section discussed in detail in the following examples.

## 5 WQ Examples

### 5.1 Creating The Input File List

Many applications, such as two of those we are about to discuss, require their input files to be of a particular type through the use of file extensions. That is, they append a period and a set of characters to the end of the file name (e.g. `.txt`, `.fna`, `.ini`, and so on). This makes it very easy to create a file holding a list of input file names. A simple shell command can create the list. The absolute path to the files can be included if desired by using the `pwd` command with `find`, like so:

```
$ find `pwd`/*.*fna > input_files
```

After `input_files` is created, it can be edited as needed.

It is not absolutely necessary to use absolute path names. Pathnames relative to the working directory the scripts are started in can work equally well. Chalk this up to a certain amount of paranoia from a control freak. It really has been tested both with absolute and relative path names. As always, try a manual test before launching a full production effort.

### 5.2 Serial Tasks

A very simple example of a serial task would be a script that does nothing but echo back the values of some environment variables and then sleeps a short time to provide some simulated processing time. It takes no real

actions so is safe to run interactively without any preparations. Let's examine the task script first, then set up the PBS file, create the input file list, and finally execute.

### 5.2.1 The `wq_timing.sh` Task Script

The serial example is pretty simply as all it does is a little shell processing magic and echos a few environment variable values (Listing 3). This script hints at some of what can be done with the absolute path of the input file name, and very little else. A shell script can do any sort of processing desired through normal shell programming methods, which is one reason it's used here. The example is based on the **bash** shell, but any script or command could be used. The only critical part of the process is the fact there is one command line argument to process.

Listing 3: Timing Task Script

```
1  #!/bin/bash
2  #
3  # This script does little but provide a way to illustrate the workings
4  # of WQ.
5
6  # First, some basic processing of the file name provided as argument 1.
7
8  FILE=$1
9  DIR='dirname ${FILE}'
10 BASE='basename ${FILE}'
11
12 # The command will just echo the results.
13
14 echo "DIR=${DIR}; BASE=${BASE}"
15 echo "That's all, folks!"
16 sleep 2
```

Note that no attempt is made to actually do anything with the file named in argument 1. It simply shows how the directory and base name parts can be extracted. The file itself doesn't even have to exist for the script to work correctly. In fact, it can be tested manually with any made-up file name, like this:

```
$ ./wq_timing.sh /test/file/path/foobar.txt
DIR=/test/file/path; BASE=foobar.txt
That's all, folks!
$
```

### 5.2.2 The `wq_timing.pbs` Prologue

The PBS script prologue section used for the serial example is displayed in Listing 4. The machine it is destined to run on has nodes equipped with two 8-core processors, or 16 cores total. The example is requesting 2 nodes, totaling 32 cores. The walltime is set to a very short 1 minute 30 seconds - it really doesn't do much! **WPN** is set to the number of cores on a node, and a work directory name is specified. The file `82_file.lst` just happens to have 82 file names swiped from another job. The serial task itself is defined in the file with the incredibly original name of `wq_serial.sh`. While not absolutely necessary, it also sets **START** to 1, which is the default value. This is all that must be changed in the PBS script. The PBS epilogue section should stay unmodified unless you really, really, really want to mess with it.

Listing 4: Serial Job Prologue Section

```
1  #!/bin/bash
2  #
3  #####
4  # Start of WQ PBS Prologue section.
5  #
6  # It is best not to use "-j oe" - the output gets a bit confusing.
7  #
```

```

8  #PBS -A hpc_enable02
9  #PBS -l nodes=2:ppn=16
10 #PBS -l walltime=00:01:30
11 #PBS -q workq
12 #PBS -o /work/jalupo/WQ/Timing
13 #PBS -N wq_timing
14
15 # Have the script run 1 worker per core on each node assigned.
16 # The system to run on has 16 cores per node (i.e. ppn=16):
17
18 WPN=16
19
20 # Set the working directory:
21
22 WORKDIR=/work/jalupo/WQ/Timing
23
24 # Use a file with 82 names listed:
25
26 FILES=${WORKDIR}/82_file_list
27
28 # Name of the task script each worker is expected to run to process
29 # the files sent to it as the only command line argument.
30
31 TASK=${WORKDIR}/wq_timing.sh
32
33 START=1
34
35 #####
36 # End WQ prologue section.

```

### 5.2.3 The Input File List

To run a proper demonstration, a file containing pathnames is needed. Let's use a list of 82 file names, which according to the prologue settings, should be found in a file named `82_file_list`. The first few entries look like:

```

/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

```

The file was created from an actual input file that was very much longer. Note that there really isn't any white-space at the beginning or end of each line. Let's assume we have a copy of `wq.py` in our current directory. The directory contents should then look something like the following prior to running the job:

```

$ ls
82_file_list  wq.py  wq_timing.pbs  wq_timing.sh

```

### 5.2.4 Execute, Execute, Execute

Everything should be good to go, so the job can be submitted:

```

$ qsub wq_timing.pbs > jid

```

The file `jid` will capture the job identifier. When this example was run, job identifier became **119123.mike3**. The job number is 119123 and it is used to create several other job unique files. This is a pretty short example and will execute in under 10 seconds, but you may have to wait a bit before it actually starts if the system is busy. When it completes, you should see a few more files in the directory:

```
$ls
82_file_list  hostlist.119123  jid  pbs.119123  wq.py  wq_timing.e119123
wq_timing.o119123  wq_timing.pbs  wq_timing.sh
```

Notice that the job ID number is used to mark files produced by the job. This is the default naming convention used by PBS for the `stdout` and `stderr` streams so is used to create names for the two auxiliary files called `hostlist.119123` and `pbs.119123`.

**hostlist.119123** This is a lists of the nodes assigned to the job. Since it is basically a serial job, each host name is listed just once.

**pbs.119123** This is a copy of the PBS script as processed by `qsub`. All blank lines were removed by `qsub` during processing, but nothing else changed. It is created so the workers have access to the script which otherwise is seen only by the mother superior.

**wq\_timing.o119123** This is the standard output from the run. It used the job name set in the PBS script and appended `.o` and the job number.

**wq\_timing.e119123** This is the standard error from the run. It used the job name set in the PBS script and appended `.e` and the job number.

Let's take a look at the contents of the standard output file (Listing 5). There are 4 important lines for each task, each written with colon-delimited fields. In all cases, the second field is the task number. The line starting with `Task` displays information about the task: the number, the worker that requested it, if the worker `Ran` or `Skipped` the task, the exit status of the task, and the full command line that worker executed. The worker name is the node host name and the sequence number of the worker on that node, starting with 0 (all good C programmers know you start counting from 0!).

The second important line begins with `Timings`. This line again show the task number followed by 4 times, all expressed in seconds. The first two are the system times at which the task started and ended expressed in seconds (this simplifies any time arithmetic you might want to do). The third is the task's elapsed time, and the final time is the job walltime when the task ended (e.g. in Task 1 below, the task ended 4.11 seconds from the start of the job). To fit the output nicely on the page `/...` represents omission of elements in the path.

#### Listing 5: Timing Example Execution

```
1 Task:1:mike067_0:Ran:True: /.../ wq_timing.sh /.../ chr13_710.bf
2 Timings:1:mike067_0:1392928642.80:1392928646.92:4.11:4.11
3 Stdout:1:
4   DIR=/work/user/chr13; BASE=chr13_710.bf
5   That's all, folks!
6 Stderr:1:
7
8 Task:3:mike067_2:Ran:True: /.../ wq_timing.sh /.../ chr13_2847.bf
9 Timings:3:mike067_2:1392928642.81:1392928646.92:4.11:4.11
10 Stdout:3:
11   DIR=/work/user/chr13; BASE=chr13_2847.bf
12   That's all, folks!
13 Stderr:3:
14
15 Task:12:mike067_11:Ran:True: /.../ wq_timing.sh /.../ chr13_187.bf
16 Timings:12:mike067_11:1392928642.81:1392928646.92:4.11:4.11
17 Stdout:12:
18   DIR=/work/user/chr13; BASE=chr13_187.bf
19   That's all, folks!
20 Stderr:12:
```

The `Stdout` lines verify the task number and are followed by any standard output produced by the task. The same treatment is given to standard error (`Stderr`).

The timing information is provided to help determine the order in which the tasks were executed. For instance, we might use `grep` to extract all the tasks executed by worker `mike067_0`:

```
$ grep Timings wq_timing.ol19123 | grep mike067_0
Timings:1:mike067_0:1392928642.80:1392928646.92:4.11:4.11
Timings:33:mike067_0:1392928647.02:1392928649.04:2.02:6.23
Timings:65:mike067_0:1392928649.14:1392928651.15:2.01:8.35
```

This shows that `mike067_0` executed tasks 1, 33, and 65. It started task 33 0.1 seconds after finishing task 1, and started task 65 0.1 seconds after finishing task 33.

The standard error file contains several different output lines which can also help figure out how the processing went. The first few near the beginning are shown in Listing 6. Pay attention to the two `Dispatcher`.

Listing 6: Timing Stderr Top Lines

```
1 Dispatcher:Start:1
2 mike044_0:Taking:1:0.00:90.00
3 mike044_1:Taking:2:0.00:90.00
4 mike044_2:Taking:3:0.00:90.00
5 mike044_3:Taking:4:0.00:90.00
6 mike044_4:Taking:5:0.00:90.00
7 . . . skipping . . .
8 mike045_13:Taking:30:0.00:90.00
9 mike045_14:Taking:31:0.00:90.00
10 mike045_15:Taking:32:0.00:90.00
11 Dispatcher:Maxtime:mike044_6:2.03:1393524348.06
12 Dispatcher:Maxtime:mike044_2:2.03:1393524348.07
13 Dispatcher:Maxtime:mike044_0:2.03:1393524348.07
14 mike044_6:Taking:33:2.04:87.96
15 mike044_2:Taking:34:2.05:87.95
16 \end{itemize}
```

The line with `Dispatcher:Start` reports the first task assigned from the input file. In this case, they were assigned from the beginning of the file. This should match the `START` setting in the PBS script. The lines with `Dispatcher:Maxtime` indicating that the named worker reported a maximum task time that was longer than any seen by the dispatcher, with the last two fields being the task time and the wall time at which the report was made. Moving to the bottom of the file, some other types of information become evident (Listing 7)

Listing 7: Timing Stderr Top Lines

```
1 . . . many skipped lines . . .
2 mike044_14:Taking:79:4.06:85.94
3 mike044_10:Taking:80:4.06:85.94
4 mike045_0:Taking:81:4.03:85.97
5 Dispatcher:Shutdown:32
6 Dispatcher>Last:82
7 mike045_1:Taking:82:4.03:85.97
```

The line with `Dispatcher:Shutdown` indicates that the workers will be told to stop working instead of having more tasks handed out, and that all 32 requested have not yet been communicated with. The line with `Dispatcher>Last` reports the last task number that was completed. Here it indicates task 82, which just happened to be given to worker `mike045_1`.

### 5.3 MPI Processing - MrBayes

MrBayes is an application that can make use MPI to speed up processing. Settings in the input files control how many MPI processes are a good number, and the set of input files used to test this example were found to work well on 2 processes. That means setting up a task to run with 2 MPI processes. The machine used for testing had 16 cores, which means 8 tasks can be run per node. For the curious, the input files selected to build the input file list all end with `.bf`.



### 5.3.1 The wq\_mb.sh Task Script

The task script requires setting up the environment appropriately for MrBayes (Listing 8). It needs some environment variables to contain paths to directories it relies on for data and library files. Note that towards the end (lines 57–63) it has an **if** block which can be used to either execute the real command by using **if true**, or just echo the command line **if false**. This provides a way of testing the script to verify the proper command line is generated.

Listing 8: MrBayes Task Script

```
1  #!/bin/bash This script executes a task under the control of the Omq
2  #worker. It is past an absolute file name as the only argument. The
3  #script should be able to function stand-alone for testing, or under
4  #worker control. What follows is an example of what might be done,
5  #but is really limited only to the scripting included. Any other
6  #scripting language could be used, even a binary program.
7
8  # Capture the input file from the argument list, and split it into
9  # the directory path part, and the basename part (name less extension).
10
11  FILE=$1
12  DIR='dirname ${FILE}'
13  BASE='basename ${FILE}'
14
15  # Execute the desired processing steps. Again, more than one is
16  # possible. This example runs MrBayes. The specifics here are for
17  #
18  # +mrbayes-3.2.2-Intel-13.0.0-openmpi-1.6.2-CUDA-4.2.9.
19  #
20  # The settings are hardwired based on what the softenv has set. Try:
21  #
22  # $ soft-dbg +mrbayes-3.2.2-Intel-13.0.0-openmpi-1.6.2-CUDA-4.2.9.
23  #
24  # to verify. soft-dbg can be used to examine other softenv keys in
25  # the same way.
26
27  # Likely do not need the INCLUDE path, but it won't hurt.
28
29  BEAGLELIB=/usr/local/packages/beaglelib/1.0/Intel-13.0.0-openmpi-1.6.2
30  export LD_INCLUDE_PATH=${LD_INCLUDE_PATH}:${BEAGLELIB}/include
31  export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${BEAGLELIB}/lib
32  MRBAYES=/usr/local/packages/mrbayes
33  export PATH=${PATH}:${MRBAYES}/3.2.2/Intel-13.0.0-openmpi-1.6.2-CUDA-4.2.9/bin
34  export mrbayes_HOME=${MRBAYES}/3.2.2/Intel-13.0.0-openmpi-1.6.2-CUDA-4.2.9
35
36  # Maybe run multiprocess (on one node only!). Here we are going to run 2
37  # per task, so create a list with the local hostname appearing 2 times.
38  # Just append the name, with a separating ",", then trim off any final ",".
39
40  PROCS=2
41  HOSTNAME='uname -n'
42  HOSTLIST=""
43  for i in `seq 1 ${PROCS}`; do
44      HOSTLIST="${HOSTNAME},${HOSTLIST}"
45  done
46  HOSTLIST=${HOSTLIST%,*}
47
48  # Here we set the command line to use. May have to be sensitive to
49  # "quoting hell" issues if it gets too fancy.
50
51  CMD="mpirun -host ${HOSTLIST} -np ${PROCS} -mb< ${FILE} > ${BASE}.out"
52
53  cd $DIR
54
55  # For testing purposes, use "if false". For production, use "if true"
56
57  if true ; then
58      eval "${CMD}"
```

```

59 else
60     echo "${CMD}"
61     echo "Faking _It_On_Hosts: _${HOSTLIST}"
62     sleep 2
63 fi

```

The MPI used in this example is OpenMPI. A different version of MPI may require the use of a launcher other than `mpirun`, and/or a different set of command line options. What is important to note is the MPI processes of a task are all on the same node, so it is relatively easy to build a host list (see lines 42–46).

### 5.3.2 The `wq_mb.pbs` Prologue

The important thing for this example is that we are using 16 MPI processes per task, which means the job should run only 1 worker per node given 16 cores per node. We'll assume the PBS file name is `wq_mb.pbs`, and the prologue portion is displayed in Listing 9. This is going to be a heftier example and use 32 nodes total. It will be allowed to run for 5 hours to illustrate what happens when time runs out before all tasks are completed.

Listing 9: MrBayes PBS Script Prologue

```

1  #!/bin/bash
2  #####
3  # Begin WQ prologue section.
4  #####
5  #PBS -A hpc_enable02
6  #PBS -l nodes=32:ppn=16
7  #PBS -l walltime=5:00:00
8  #PBS -q workq
9  #PBS -N WQ_MrBayes
10
11 # Set number of workers per node:
12
13 WPN=1
14
15 # Set the working directory:
16
17 WORKDIR=/work/jalupo/WQ/MrBayes
18
19 # Name of the file containing the list of input files:
20
21 FILES=${WORKDIR}/wq.lst
22
23 # Name of the task script each worker is expected to run to process
24 # the files sent to it as the only command line argument.
25
26 TASK=${WORKDIR}/wq_mb.sh
27
28 START=1
29
30 #####
31 # End WQ prologue section.

```

As before, we leave the PBS epilogue section well enough alone. Before we can execute, we'll need to create a list of input files, this time full of real, live file names.

### 5.3.3 The `wq.lst` Input List

Before the job was started, the contents of the `WORKDIR` looked like:

```

$ ls
PostPredYeast  wq.lst  wq_mb.pbs  wq_mb.sh  wq.py

```

The sub-directory `PostPredYeast` contains all the data files, and will hold the task output files as well. The file `wq.lst`, which is the `FILES` setting, contains a list of 1,414 input file names. After abbreviating the names for display purposes, the first four lines look like:

```

/.../PostPredYeast/YDR449C_DNA/SeqOutfiles/YDR449C_DNA.nex.run3_1360000/GTRIG.bayesblock
/.../PostPredYeast/YDR449C_DNA/SeqOutfiles/YDR449C_DNA.nex.run1_1960000/GTRIG.bayesblock
/.../PostPredYeast/YDR449C_DNA/SeqOutfiles/YDR449C_DNA.nex.run2_3960000/GTRIG.bayesblock
/.../PostPredYeast/YDR449C_DNA/SeqOutfiles/YDR449C_DNA.nex.run2_2360000/GTRIG.bayesblock

```

We now have the task script (check), the input file list (check), and the PBS script (check). Guess we're good to go!

### 5.3.4 Make It So

There is nothing mysterious about launching this job:

```

$ qsub wq_mb.pbs > jid
$

```

Examining the file **jid**, we find that this particular job was assigned the job name 119880.mike3. Hence, when it completed, the contents of the WORKDIR looked as so:

```

$ ls
PostPredYeast  hostlist.119880 pbs.119880 wq.lst  wq_mb.pbs  wq_mb.sh
wq.py WQ_MrBayes.e119880  WQ_MrBayes.o119880

```

The contents of WQ\_MrBayes.e119880 is much longer than the serial example simply because more tasks were completed (515 in this case). The first few beginning and ending lines look like:

```

Dispatcher:Start:1
mike241_0:Taking:1:0.00:18000.00
mike309_0:Taking:2:0.00:18000.00
mike308_0:Taking:3:0.00:18000.00
mike302_0:Taking:4:0.00:18000.00
mike299_0:Taking:5:0.00:18000.00
. . . (removed for brevity) . . .
mike303_0:Taking:512:15323.72:2676.28
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:515

```

It appears that once worker mike322\_0 took task 516, it realized there might not be enough time to complete it, hence it reported that it had to skip the task because time was up. The dispatcher reports shutting down the other 31 workers, and that the last completed task was 515. The remainder of the contents show of the workers completed 16 tasks, and a few did more. Looking over the timing data, the workers all stopped with 695 seconds of each other (see Section 6.2 for details), so the work was fairly balanced. Of course that represents this particular set of data and may not be representative of what can be expected with a arbitrary data set. Since we know that 515 tasks completed, we could now resubmit the PBS script after making two changes: `START=516` will skip over the completed tasks, and `walltime=15:00:00` should give sufficient time for the remaining 899 tasks to complete. If not, just run a 3rd job to pick up the strays.

## 5.4 BLASTN

BLASTN is an application that runs multi-threaded with OpenMP. What we'll do here is run 4 workers per node, with each task using 4 threads on 4 of the 16 cores available. The OpenMP thread count is specified by setting an appropriate environment variable.

### 5.4.1 The `wq_blastn.sh` Task Script

What is different in the task script from the MPI example is the use of the variable `OMP_NUM_THREADS` to set the number of threads a process can use (Listing 10). Notice that the BLASTN command line is the most complex of all the examples, and is built up step-wise to keep things readable. The decision to use 4 threads per task is based on the characteristics of the input files, other types of input may require using only 2 or 1 thread per task.

Listing 10: BLASTN Task Script

```
1  #!/bin/bash
2
3  # Set a variable as the absolute path to the BLASTN executable.
4
5  BLASTN=/usr/local/packages/bioinformatics/ncbiblast/2.2.28/gcc-4.4.6/bin/blastn
6
7  # Specify use of 4 threads.
8
9  export OMP_NUM_THREADS=4
10
11 # Capture the input file from the argument list, and split it into
12 # the directory path part, and the basename part (name less extension).
13
14 FILE=$1
15 DIR='dirname ${FILE}'
16 BASE='basename ${FILE}'
17
18 # Build up the rather complex command line.
19
20 CMD="${BLASTN} -task blastn -outfmt 7 -max_target_seqs 1"
21 CMD="${CMD} -num_threads ${OMP_NUM_THREADS}"
22 CMD="${CMD} -db /project/jcthrash/db/img_v400_custom/img_v400_custom_GENOME"
23 CMD="${CMD} -query ${FILE}"
24 CMD="${CMD} -out ${DIR}/IMG_genome_blast.${BASE}"
25
26 # Execute the desired processing steps. Again, more than one is
27 # possible. This example runs BLASTN against a particular database.
28 # For testing purposes, use "if false". For real runs, use "if true":
29
30 if true ; then
31     eval "${CMD}"
32 else
33     echo "${CMD}"
34     # This just slows things way down for testing.
35     sleep 1
36 fi
```

### 5.4.2 The `wq_blastn.pbs` Prologue

The processing of a single data set takes much longer than our earlier examples, so this example is set up to run on 1 node for 70 hours. That means a total of 4 works will run at any given time.

Listing 11: BLASTN PBS Script

```
1  #!/bin/bash
2  #####
3  # Begin WQ prologue section.
4  #####
5  #PBS -A hpc_enable02
```

```

6  #PBS -l nodes=1:ppn=16
7  #PBS -l walltime=70:00:00
8  #PBS -q workq
9  #PBS -o /work/jalupo/WQ/BLASTN
10 #PBS -N wq_blastn
11
12 # The task script is going to run BLASTN via OpenMP with 4 processes ,
13 # so we'll divide PPN by 4, which gives 4 workers on a 16 core node.
14
15 # Set number of workers per node:
16
17 WPN=4
18
19 # Set the working directory:
20
21 WORKDIR=/work/jalupo/WQ/BLASTN
22
23 # Name of the file containing the list of input files:
24
25 FILES=${WORKDIR}/fna_files
26
27 # Name of the PBS script file:
28
29 JOBFILE=${WORKDIR}/wq_blastn.pbs
30
31 # Name of the task script each worker is expected to run to process
32 # the files sent to it as the only command line argument.
33
34 TASK=${WORKDIR}/wq_blastn.sh
35 START=1
36
37 #####
38 # End WQ prologue section.

```

Again, the epilogue section of the script (Listing 2) is unchanged.

### 5.4.3 Contents of fna\_files

The input files selected are those that end in .fna. The first few lines contain:

```

/work/jalupo/WQ/BLASTN/GS049.blast/xab.fna
/work/jalupo/WQ/BLASTN/GS049.blast/xak.fna
/work/jalupo/WQ/BLASTN/GS049.blast/xai.fna
/work/jalupo/WQ/BLASTN/GS049.blast/xag.fna
/work/jalupo/WQ/BLASTN/GS049.blast/xaj.fna

```

The file names are a bit shorter than the MrBayes example because the directory tree isn't near as deep. Instead of one data directory, there are multiple directories in the WORKDIR. In fact, before the job is executed, the contents look like:

```

$ls
fna_files  GS036.blast  GS037.blast  GS038.blast  GS039.blast
GS040.blast  GS041.blast  GS042.blast  GS043.blast  GS044.blast
GS045.blast  GS046.blast  GS047.blast  GS048a.blast  GS049.blast
wq_blastn.pbs  wq_blastn.sh  wq.py

```

Anything missing? Nope we're all set.

### 5.4.4 Engage

Hopefully, you have this job submission thingy committed to memory:

```
$ qsub wq_blastn.pbs > jid
$
```

This time, the job name assigned was 121166.mike3, so you should be able to predict what WORKDIR contained after it completed, right?

```
$ls
fna_files  GS036.blast  GS037.blast  GS038.blast  GS039.blast
GS040.blast  GS041.blast  GS042.blast  GS043.blast  GS044.blast
GS045.blast  GS046.blast  GS047.blast  GS048a.blast  GS049.blast
hostlist.121166  pbs.121166  wq_blastn.e121166  wq_blastn.o121166
wq_blastn.pbs  wq_blastn.sh  wq.py
```

The job completed in 48 hours and a look at the contents of wq\_blastn.e121166 indicates what happened:

```
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:5
```

The worker mike111\_1 finished its first task in 64,223.42 seconds (17.84 hrs). Thus when it requests more work, it estimated there was plenty of time to complete it and so pressed on. It took worker mike111\_2 123837.44 seconds (34.4 hrs) to complete its first. That means when it computed the time remaining, and included the runtime safety factor of 1.25, it detected that time was up, and skipped the task. The dispatcher shut down the other workers when they requested more work, and notes that 5 tasks were completed.

This is a good example of wildly varying run times. With a limit of 70 hours total, there is much leeway to smooth out run times that vary by a factor of 2. The impact is that 12 cores were idle for about 14 hours as the 5<sup>th</sup> task was completed.

## 6 Some Sanity Checks

### 6.1 Worker End Times

We can make use of the formatted output to check on how well the workers performed. For instance, we'd like to be sure that there were no delays between the end of a task and the start of a new one. To do this, we could pull out the Timing lines for a particular worker. Here's is how it might be down with the MrBayes example:

Listing 12: MrBayes Example Timing Data

```
1 $ grep Timings WQ_MrBayes.o119880 | grep mike322_0
2 Timings:11:mike322_0:1393602362.34:1393603386.53:1024.19:1024.19
3 Timings:33:mike322_0:1393603386.53:1393604526.32:1139.78:2163.97
4 Timings:76:mike322_0:1393604526.32:1393605575.58:1049.26:3213.24
5 Timings:103:mike322_0:1393605575.58:1393606075.38:499.79:3713.03
6 Timings:130:mike322_0:1393606075.38:1393606575.01:499.63:4212.67
```

```

7 Timings:158:mike322_0:1393606575.01:1393607098.21:523.20:4735.87
8 Timings:191:mike322_0:1393607098.21:1393607639.23:541.02:5276.89
9 Timings:226:mike322_0:1393607639.23:1393608302.66:663.43:5940.32
10 Timings:258:mike322_0:1393608302.66:1393609006.48:703.82:6644.14
11 Timings:292:mike322_0:1393609006.48:1393609652.26:645.78:7289.92
12 Timings:322:mike322_0:1393609652.26:1393611663.55:2011.29:9301.21
13 Timings:365:mike322_0:1393611663.56:1393613379.13:1715.57:11016.79
14 Timings:394:mike322_0:1393613379.13:1393615136.07:1756.94:12773.72
15 Timings:422:mike322_0:1393615136.07:1393615999.23:863.16:13636.89
16 Timings:450:mike322_0:1393615999.23:1393616912.33:913.10:14549.98
17 Timings:483:mike322_0:1393616912.33:1393617821.63:909.30:15459.29
18 Timings:516:mike322_0:-1.00:-1.00:-1.00:15459.29
19 $

```

Pay attention to field 5, the end time of one line and field 4, the start time, of the following line. It is not until you get down to lines 10–11 that you notice a 0.01 sec delay between the end of a task (322) and start of the next assignment (365). Things were chunking along at a pretty regular clip with minimal overhead time between tasks.

Alternatively, one could import the data into a spreadsheet, using the colon as the field separator, and do more exotic calculations.

## 6.2 Load Balancing Jitter

Jitter is a fancy technical term which represents how much variance there is in timing values for a bunch of related activities. In distributed processing, this would be how uniform the execution times are for each task, and how close the workers come to the same end time of all work. This was discussed above, and in short, we'd want all the works to complete at the same time, or within an acceptably short interval.

The same output file is used, and similar methods of selecting what information to track. The script in Listing 13 determines the earliest and latest worker end times.

Listing 13: Worker End Time Script

```

1 import sys
2
3 f = open( sys.argv[1], 'r' )
4 raw = f.readlines()
5 f.close()
6 worker = {}
7 max_end = 0.0
8 min_end = 1.0e+37
9
10 for l in raw:
11     u = l.strip().split(':')
12     if u[0] == 'Timings' :
13         t = float(u[4])
14         if t > 0.0 :
15             worker[u[2]] = t
16
17 for k, v in worker.iteritems() :
18     if v < min_end :
19         min_end = v
20     if v > max_end :
21         max_end = v
22
23 print( "End times between %.2f and %.2f - %.2f sec spread." %
24        ( min_end, max_end, max_end - min_end ) )

```

Let's apply this script to the output of the MrBayes example:

```

$ python timing.py WQ_MrBayes.o119880
End times between 1393617821.63 and 1393618516.08 - 694.45 sec spread.
$

```

Given that the maximum task time was 2045.15 sec, having all workers finish within about 1/3 of the max time is pretty good.



## References

### References

- [1] Tom Bishop, Shantenu Jha, and Hideki Fujioka. ManyJobs. <http://dna.engr.latech.edu/ManyJobs/>, Nov 2013.
- [2] GNU Parallel. <http://www.gnu.org/software/parallel/>.
- [3] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Andre Merzky, John Shalf, and Christopher Smith. A Simple API for Grid Applications (SAGA). Technical Report GFD-R-P.90, Open Grid Forum, 15 Jan 2008.