WQ — A Task Distributor: The User Manual

Document SVN Version: 265
Document Date: 2016-01-25 16:23:56Z
Program SVN Version: 264

James A. Lupo
jalupo@cct.lsu.edu
Center for Computation & Technology
Louisiana State University
Baton Rouge, LA

Contents

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 (WorkQueuing) is a task distributor for HPC clusters, sort of a private task (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 have 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 serial jobs under a batch submission system can be very monotonous, may require modestly advanced skills at shell scripting, and can cause problems with the job scheduler itself. If you can imagine the difference between setting up 1000 serial jobs with 1 task each versus 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, yet can be used in more complex situations. You might imagine how 1000 tasks could make explicit scripting rather daunting from a repetition viewpoint, yet the effort is fundamentally too simple to turn over to a heavy weight solution, such as SAGA[3] or ManyJobs[1]. From a scripting standpoint, 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 simple serial task, a multi-threaded OpenMP task, a small **MPI** task, and a couple of illustrative alternative handling of the input line provided to the tasks for those with more shell scripting acumen.

2 Some Definitions

The WQ system uses a "dispatcher-worker" model of distributed computing (e.g. hence W(ork)Q(ueuing)). 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 port the scripts 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 exclusively to one user for a specified amount of time. The user may then 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, the latter two being restricted to using the cores of a single node.

walltime Time determined using a conventional time-of-day clock. It is used to track the job elapsed time irrespective of any other system times, such as CPU, USER, 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. One important task is tracking the time taken by the longest running task seen during a job. This allows it to manage a 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 a task. It requests a task from the dispatcher, executes it, sends results back, and 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.

While the discussion are focused on using WQ on clusters with job scheduling systems, it can still be used manually to run multiple tasks on a local desktop or workstation.

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 (nb. easy on a parallel file system). 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.

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 nominally 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 line to the command, and the command script can then do what it will with the information. In the generic use case, the command is a shell script, and each line is a file name. The names could be absolute path names to input data files, and the task executed by the worker would be a shell command line that looks like:

```
$ command filenm
```

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 sends a report containing the execution status code, all text written to standard output (`stdout`), and any text

written to standard error (stderr), back to the dispatcher. If the application produces a high volume (more than 10 lines??) of output on stdout, it would be best to have it written out to a separate output file rather than return it through stdout or stderr. The information returned, along with diagnostics, is written to a file named (wq.log.jobid).

There are 4 files important to the process of running a WQ job. The heavy lifting is done by wq-pbs.sh and wq.py, which provide the PBS interface, and the dispatcher/worker services respectively. These should be installed somewhere in your system PATH. The two other files are wq.pbs, which the user sets up for batch submission, and wq.sh, which is an example of a task command script. These are the files a user would customize for a particular job.

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. Run as the dispatcher, it provides the workers with a couple of special environment variables, and other information needed to run the task. It tracks elapsed wall time, and will stop when it appears time is about to run out. By default, 1 day (86400) seconds is used, or it receives the PBS walltime request from the PBS script. Run as a worker, it first opens a communication channel to the dispatcher and starts cycling through the work request process until work is done or it is told to stop. The file is executable, and should reside somewhere in PATH. It depends on the zmq Python module which is not always included in Python distribution.

wq-pbs.sh This is a shell script that manages the job environment on all the nodes assigned to a job. It takes care of starting the dispatcher and proper number of workers on all the nodes.

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, desired job queue, and other typical settings. The user specifies the number of workers to run per node, which is then used to compute the number of cores/threads each work can use. 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 concerted. The dispatcher does provide workers with two variables that can help. The epilogue section is the WQ specific portion. It consists of one line which calls wq-pbs.sh and passes the settings. The workings of this script is tightly coupled to the behavior of wq.py and so must be used as a pair.

wq.sh This is a shell script that the workers are told to run to actually accomplish a task. Again, the name is not significant, but core usage should be consistent with whatever appears in the PBS batch script. The script (or other specified program) should expect 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 preprocessing 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. The two files that accomplish all the magic (wq.py and wq-pbs.sh, will be covered later on. For now, we'll simply assume they are in a directory somewhere in PATH and are executable. You could check by typing:

```
$ which wq.py
$ which wq-pbs.sh
```

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:

```
$ wq.py --help

Usage: wq.py opt1 [opts ...]

   As Dispatcher:

        wq.py -d[--dispatcher] cmd [-s[--start] task_num] \
              [-t[--time] walltime] [-v[--verbose]] -a[--allworkers] n \
              -i[--input] filenm -j[--jobid] jobid

   As Worker:

        wq.py -w[--workers] n -m[--mothersuperior] ms

   Help:

        wq.py wq.py -h[--help]

   General:

      -v,--verbose ..... Increase STDERR output details.
      -h,--help ........ Display this help message.

   Dispatcher Options:

      -d,--dispatcher cmd .. (Required) Run as the dispatcher for the
                             command cmd, where cmd is the task
                             program or script. cmd will be called
                             with a single file name as its only
                             argument.

      -a,--allworkers n .... (Required) Total workers (workers per
                             node * nodes ).

      -i,--inputs filenm ... (Required) Name of file containing input file
                             names to serve as inputs to cmd, one name
                             per line.

      -j,--jobid ........... (Option) PBS jobnumber. If provided, will
                             be added to the log file name: wq.log.jobid

      -s,--start task_num .. (Optional) Task number to start
                             with. Represents the line number in the
                             input list file. Default is 1.

      -t,--time walltime ... (Optional) 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)


Worker Options:

   -w,--workers n ..........  (Required) Run n workers per node.

   -m,--mothersuperior ms ..  (Required) Host name of mother superior
                              node.

The dispatcher must be started before any of the workers.

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

Revision: $Id: wq.tex 265 2016-01-25 16:23:56Z 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 provided by an environment variable to the mother superior. The dispatcher uses it, coupled with the longest task time it has seen, and estimates whether or not there is sufficient time to complete another task. If there is, it hands out a task. If not, it begins telling the workers to shut down. 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 workers. Before it assigns a new task to a worker, it checks to see if there is sufficient time. It does this by taking the longest task time, multiplying it by a safety factor (1.25) and comparing with the walltime remaining. If there is sufficient time, it sends the worker a new task. If not, it sends the worker a command to terminate. 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. One approach to avoid the problem is to make use of any run time estimates possible based on the inputs and try to order the input to put longest running tasks first. The other is to make sure the walltime requested is longer than the longest expected runtime by at least that amount.

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 out as expected. Namely, the time calculations result in stopping the 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

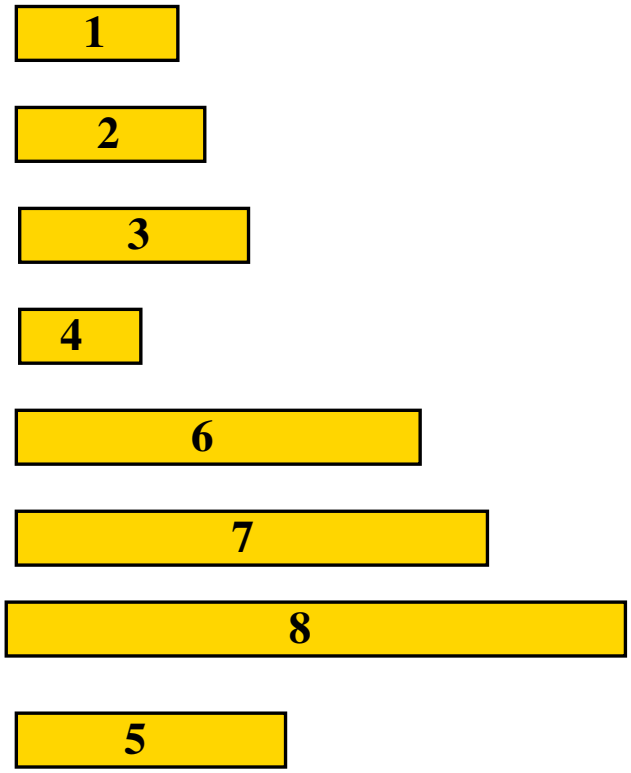

Figure 1: A collection of 8 tasks. The required walltime for each is proportional to the length of each task's bar. The box numbers are meant for identification and not necessarily the order in which the tasks will be assigned.

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

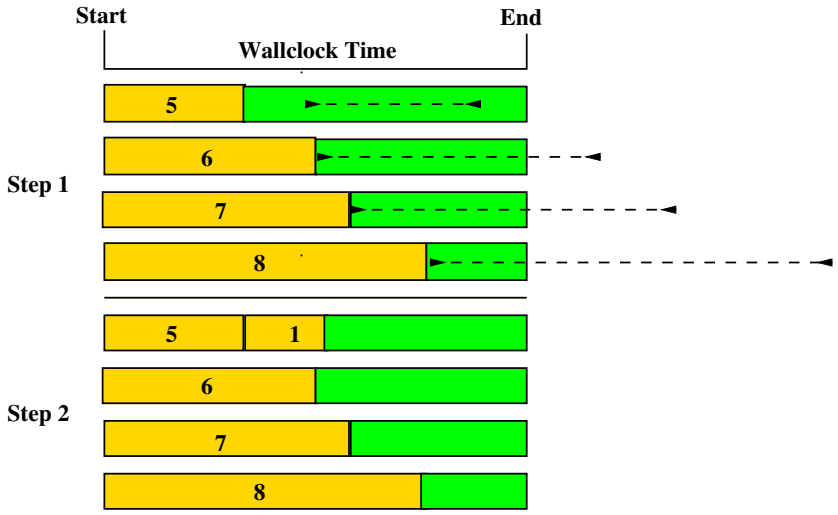

Figure 2: A successful job scenario. Step 1 shows the first set of tasks to be executed. The green boxes indicate the actual walltime remaining. The dashed lines indicates the maximum walltime to allow for the next job, based on what has completed so far.

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. This depends on knowing how run times vary with input parameters, and such information may not be readily available, or possible to determine. The next best thing is to allow more than 2 or 3 times the longest anticipated task time for the entire job. The job will only be charged for the time actually used, so the only downside may be a possible delay in job start due to the length of time requested.

4.2 The PBS Script

The WQ PBS script, which we'll name `wq.pbs` but could be any name, has two main parts. These will be referred to as the *prologue* and *epilogue* sections. 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

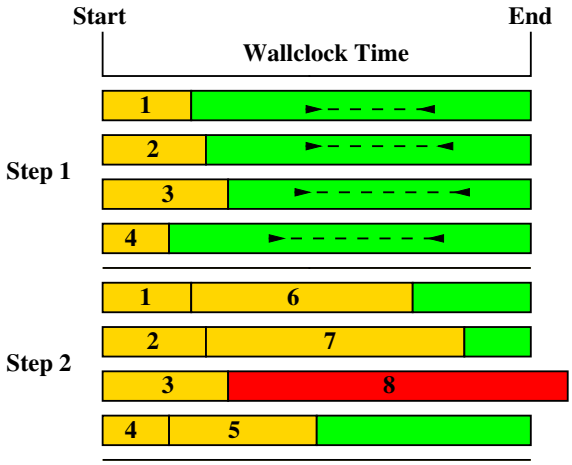

Figure 3: A failed task/job scenario. Step 1 shows the first set of tasks to be executed. The green boxes indicate the actual walltime remaining. The dashed lines indicates the maximum walltime to allow for the next job, based on what has completed so far. The red box indicates a task that was started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

changes. We'll look at each before digging into some examples.

4.2.1 The Prologue

List 1 shows a WQ PBS script, with the prologue section marked by Begin/End comments. Anyone familiar with PBS should recognized the first 12 lines as comments followed by 5 PBS options. In this case, they set an allocation, or account, code; asks for 4 nodes with 16 cores per node (64 cores total); a wall time of 5 minutes; the use of the *workq* queue; and assigns the job name of *WQ_Test*. They should be treated as representative as some are optional and some required - with dependence on local requirements. The setting of several shell variables follow which define the task characteristics. A description of these variables follows the listing.

Listing 1: WQ PBS Script

```
1  #! /bin/bash
2  ##############################################################################
3  # Begin WQ prologue section.
4  #
5  # This section is changed to meet system, user and application specific
6  # requirements, such as allocation codes, PBS options, etc.
7  ##############################################################################
8  #PBS -A hpc_enable04
9  #PBS -l nodes=4:ppn=16
10 #PBS -l walltime=00:05:00
11 #PBS -q workq
12 #PBS -N JobName
13
14 # wq.py and wq-pbs.sh must executable and accessible via PATH
15
16 # Set up 4 workers per node. This will set the variable WQ_CPT=PPN/WPN
17 # (threads or cores per process) for the task script.
18
19 WPN=4
20
21 # Set the working directory. $PBS_O_WORKDIR should NOT be used.
```

```
22
23 WORKDIR=/work/jalupo/Scratch
24
25 # Name of the file containing the list of input files:
26
27 FILES=${WORKDIR}/fna_files
28
29 # Start with task 1 (first line):
30
31 START=1
32
33 # Name of the task script:
34
35 TASK=${WORKDIR}/wq_blastn.sh
36
37 # Turn verbosity off(0)/one(1):
38
39 VERBOSE=1
40
41 ##########################################################################
42 # End WQ prologue section.
43 ##########################################################################
44 # Begin WQ epilogue section.
45 ##########################################################################
46
47 wq-pbs.sh $0 $WPN $WORKDIR $FILES $START $TASK $VERBOSE $1
```

Note that the PBS options are followed by shell commands which set 6 environment variables used elsewhere in the script. These are adjusted by the user depending on the task requirements.

WPN Defines 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= value). The number used here must be consistent with the settings used in the task script file. To help, WQ provides the worker with the number of cores it calculates by detecting the number of cores in the node and dividing it by WPN. This is done by defining a shell variable WQ_CPT. This can be used to guarantee only the proper number of cores will be used.

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 in this directory.

FILES A file containing the list of input data files to be processed. Each line should be treated as the path to one file. But the line contents are passed intact to the task, so with proper handling, arbitrary data such as whole command lines could be passed to the task.

START Indicates the starting line, or record, number 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 pick up were a previous job left off and complete the rest of the file.

TASK Name of the script or program that will be run for each task. It will be called with a single argument, that being the contents of an input line.

VERBOSE Can be used to increase the information output by the dispatcher to the log file. Use a value of 1 to enable it. The default value is 0.

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, and the job number.

4.2.2 The Epilogue

The epilogue section contains a comment and one line which executes `wq-pbs.sh`. `wq-pbs.sh` and `wq.py` are designed to work as a pair, so any changes here might break something — handle with care!

5 Output

WQ sends error information to `stderr` and diagnostic information to a file named `wq.log.jobid`. This is the reason why joining the outputs with the PBS `-j` option is not recommended, and sending output for each task to a separate file is. The output takes the form of colon-delimited lines, each containing specific types of information. A representative format for the lines is:

`<src>:<tag>:<info1>:<info2:...`

where

`<src>` will be `Dispatcher` or `Worker`.

`<tag>` is a data identification tag such as `Task` or `Stderr`.

`<info>` is a single piece of data dependent on the tag.

Two different sorts of time are reported, all in units of seconds. The first is elapsed time, such as how long a task took to process. The other is wallclock time, which is the system time of day reported in seconds since 1 Jan 1970 (i.e. the UNIX epoch). Wallclock time is local to the node reporting it, so while the nodes synchronize their clocks, it should be used as only an approximation when comparing when events occurred between nodes.

In many cases, the worker names are reported. These take the form:

`hostnm_n`

where

`hostnm` – The host, or node, name.

`n` – The worker number on this host.

The amount of output can be controlled by the `wq.py --verbose` option, so we'll describe the default output, and then the added verbose output that can be expected.

5.1 Non-Verbose

There are 8 types of lines which are always generated.

`Dispatcher:Start:N:W` – Dispatcher starting with task `N` at time `W`.

`Dispatcher:Maxtime:W:M:T` – New maximum task time recorded from worker `W` of duration `M` at wall-time `T`.

`Dispatcher:Last:N` — Reports the last task number processed `N`.

`Dispatcher:Shutdown:N:A:Mn:Mx:Wa:Wm:Wx` – Termination message report number of tasks `N`, average task time `A`, minimum task time `Mn`, maximum task time `Mx`, average worker runtime `Wa`, minimum worker runtime `Wm`, max worker runtime (total job time) `Wx`].

`Worker:Startup:W:T` – Worker `W` started at time `T`.

Worker:Starting:W:E – Acknowledging worker W starting at job elapsed time E.

Worker:Stderr:N:W – Stderr output follows from task N run on worker W.

Worker:Stdout:N:W:S – Stdout output follows from task N run on worker W. Task success S is either True or False.

5.2 Verbose

If the VERBOSE flag is set to 1 in the PBS file, then an additional 7 types of lines are generated.

Dispatcher:Task:N:W:L – Task number N was assigned to worker W with line contents L.

Dispatcher:WaitOn:N – Number of workers N still processing.

Dispatcher:WaitFor:W:W:... – list of workers (W) that haven't stopped yet.

Worker:Affinity:W:S – Worker W environment variable WQ_NUMACTL setting S. This is used to constrain the cores the worker can use for it's tasks. More on this below.

Worker:CPT:W:S – Worker W environment variable WQ_CPT setting S. This provides the number of cores/threads a worker can use as calculated by wq.py.

Worker:Task:N:W:C – For task number N, worker W tried to execute command C.

Worker:Timings:N:W:Ts:Te:E – Task number N on worker W started at time Ts and ended at time Te for E elapsed seconds.

5.2.1 WQ_NUMACTL

When the workers call the task script, they define the environment variable WQ_NUMACTL. This is set to a **numactl** command string that restricts whatever it runs to a specific set of cpu cores. It will look something like this:

```
WQ_NUMACTL="numactl --physcpubinding=m-n -- "
```

where m-n represents a range of zero-based core numbers (i.e. they may run from 0 to maxcores-1).

This is likely needed whenever several multi-threaded or multi-process MPI tasks are run at the same time on a node. It makes sure that each thread or process is confined to a separate set of cpu cores. MPI, in particular, has a tendency of starting every MPI process from core 0, and leaving others unused. The use is very straight forward and suggested by the name. Just prefix the task command with the string defined by the variable, and all should be taken care for.

For example, if the desired command is:

```
$ eval ``myprog -nthreads=8 -is the best''
```

then use of WQ_NUMACTL would involve running the command as so:

```
$ eval ``$WQ_NUMACTL myprog -nthreads=8 -is the best''
```

The examples that will follow show deployment in more realistic situations.

5.2.2 WQ_CPT

One other piece of information provided by the workers to the task scripts is the number of cores each task is allowed to use. This is done by setting the environment variable WQ_CPT with the number of cores per tasks. This is computed from the number of cores detected on the compute node divided by the requested number of workers per node.

6 WQ Examples

At this point it is useful to discuss several examples of using WQ. Three of the examples that follow illustrate how purely serial jobs, multiple process **MPI** jobs, and multi-threaded OpenMP jobs are handled. Two example of more advanced use will include passing multiple arguments to the task command line, and treating the entire input line as a fully formed command. At the very end, the possibility of using WQ without PBS is shown.

6.1 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 identified 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 task script working directory can work equally well, and produce smaller files. Chalk this approach up to a certain amount of paranoia from a control freak (one never knows when having the absolute path will come in handy for something unexpected). It really has been tested both with absolute and relative path names. As always, try a manual test before launching a full production effort.

6.2 Serial Processing - Simple Script

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 is 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.

6.2.1 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 2). 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 2: Serial Task Script

```
1  #! /bin/bash
2  #
3  # This script illustrates a basic WQ task script.
4
5  # Treat the first command line argument as a full pathname:
6
7  FILE=$1
8  DIR=`dirname ${FILE}`
9  BASE=`basename ${FILE}`
10
11 # Just echo out the results and WQ variable values.
12
13 echo "DIR=${DIR}; BASE=${BASE}"
14 echo "WQ_NUMACTL=${WQ_NUMACTL}; WQ_CPT=${WQ_CPT}"
15 echo "That's all, folks!"
16
```

```
17 # Simulate a spread in task times:
18
19 T=`expr 1 + $RANDOM % 10`
20 echo "Sleeping for $T seconds."
21 sleep $T
```

If you're wondering about the RANDOM variable, it contains a random integer value provided by BASH and used here to cause some fluctuations in sleep time to simulate different processing times of real tasks. The values of WQ_CPT and WQ_NUMACTL are empty, since they are created at run time by the workers when they call the script. 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
WQ_CPT=; WQ_NUMACTL=
That's all, folks!
Sleeping for 5 seconds.
$
```

6.2.2 The wq_timing.pbs

The PBS script used for the serial example is displayed in Listing 3. The machine it is destined to run on has nodes equipped with two 10-core processors, thus 20 cores total. The example is requesting 1 node, so 20 cores in all. The walltime is set to a very short 2 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, and `VERBOSE` to 0, which are the default values but as used as reminders of what is possible. 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 3: Serial Job Prologue Section

```
1  #! /bin/bash
2  ###############################################################################
3  # Begin WQ prologue section.
4  #
5  # This section is changed to meet system, user and application specific
6  # requirements, such as allocation codes, PBS options, etc.
7  ###############################################################################
8  #PBS -A hpc_enable05
9  #PBS -l nodes=2:ppn=20
10 #PBS -l walltime=00:05:00
11 #PBS -q workq
12 #PBS -N Timing_V1
13
14 # Make sure wq.py and wq-pbs.sh appear executable in PATH.
15
16 # Running 4 workers per node. This will set the variable WQ_CPT=ppn/WPN
17 # (threads or cores per process)
18
19 WPN=4
20
21 # Set the working directory.
22
23 WORKDIR=/work/jalupo/WQ/Examples/Timing
24
25 # Name of the file containing the list of input files:
26
27 FILES=${WORKDIR}/82_file_list
```

```
28
29 # Start with task 1 (first line):
30
31 START=1
32
33 # Name of the task script:
34
35 TASK=${WORKDIR}/wq_timing.sh
36
37 # Turn verbosity off(0)/one(1):
38
39 VERBOSE=1
40
41 ##########################################################################
42 # End WQ prologue section.
43 ##########################################################################
44 # Begin WQ epilogue section.
45 ##########################################################################
46
47 wq-pbs.sh $0 $WPN $WORKDIR $FILES $START $TASK $VERBOSE $1
```

6.2.3 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` (e.g. contains 82 file names). The first few entries are shown in Listing 4.

Listing 4: Serial Job File List

```
1 /work/jalupo/WQ/Examples/Timing/chr13/chr13_710.bf
2 /work/jalupo/WQ/Examples/Timing/chr13/chr13_727.bf
3 /work/jalupo/WQ/Examples/Timing/chr13/chr13_2847.bf
4 /work/jalupo/WQ/Examples/Timing/chr13/chr13_711.bf
5 /work/jalupo/WQ/Examples/Timing/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 whitespace 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 that seen in Listing 5.

Listing 5: Timing Example Directory Before Run

```
1 82_file_list  before  wq_timing.pbs  wq_timing.sh
```

6.2.4 Execution

Everything should be good to go, so the job can be submitted with the usual `qsub` command line:

```
$ qsub wq_timing.pbs > jid
```

The file `jid` will capture the job identifier. When this example was run, job identifier became **426092.mike3**. The job number is 426092 and it is used to create several other job unique files. This is a pretty short example and will likely execute in under 120 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 (Listing 6.

Listing 6: Timing Example Directory After Run

```
1 5_file_list       jid                Timing_V1.o426092  wq_timing.pbs
2 82_file_list      pbs.426092         wq.log.426092      wq_timing.sh
3 hostlist.426092   Timing_V1.e426092  wq.py
```

Notice that the job ID number is used as a file name extension 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 three WQ auxiliary files with base names of `hostlist`, `pbs`, and `wq.log`.

`hostlist` 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` 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.o` 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.e` 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.

`wq.log` This is the diagnostic output produced by `wq.py`.

Let's take a look at the first few lines of the log output file (Listing 8). Refer to the line content descriptions above if you need to:

Listing 7: Serial Example Execution

```
1  Dispatcher:Start:1:1449777983.95
2  Worker:Startup:mike150_0:1449777988.91
3  Worker:Affinity:mike150_0:WQ_NUMACTL="numactl --physcpubind=0-3 -- "
4  Worker:CPT:mike150_0:WQ_CPT=4
5  Worker:Starting:mike150_0:4.97
6  Dispatcher:Task:1:mike150_0:/work/jalupo/WQ/Examples/Timing/chr13/chr13_710.bf
7  Worker:Startup:mike150_3:1449777988.92
8  Worker:Affinity:mike150_3:WQ_NUMACTL="numactl --physcpubind=12-15 -- "
9  Worker:CPT:mike150_3:WQ_CPT=4
10 Worker:Starting:mike150_3:4.97
11 Dispatcher:Task:2:mike150_3:/work/jalupo/WQ/Examples/Timing/chr13/chr13_727.bf
12 Worker:Startup:mike150_1:1449777988.91
13 Worker:Affinity:mike150_1:WQ_NUMACTL="numactl --physcpubind=4-7 -- "
14 Worker:CPT:mike150_1:WQ_CPT=4
15 Worker:Starting:mike150_1:4.97
16 Dispatcher:Task:3:mike150_1:/work/jalupo/WQ/Examples/Timing/chr13/chr13_2847.bf
17 Worker:Startup:mike150_2:1449777988.92
18 Worker:Affinity:mike150_2:WQ_NUMACTL="numactl --physcpubind=8-11 -- "
19 Worker:CPT:mike150_2:WQ_CPT=4
20 Worker:Starting:mike150_2:4.97
21 Dispatcher:Task:4:mike150_2:/work/jalupo/WQ/Examples/Timing/chr13/chr13_711.bf
22 Worker:Startup:mike172_0:1449777989.76
23 Worker:Affinity:mike172_0:WQ_NUMACTL="numactl --physcpubind=0-3 -- "
24 Worker:CPT:mike172_0:WQ_CPT=4
25 Worker:Starting:mike172_0:5.81
26 Dispatcher:Task:5:mike172_0:/work/jalupo/WQ/Examples/Timing/chr13/chr13_696.bf
27 Worker:Startup:mike172_1:1449777989.76
28 Worker:Affinity:mike172_1:WQ_NUMACTL="numactl --physcpubind=4-7 -- "
```

Lets run this again, only this time with `VERBOSE=1`. What is most interesting is the last few lines of `wq.log` which contain the task timing information.

Listing 8: Serial Example wq.log Lines

```
1  Worker:Stderr:81:mike172_3
2  Worker:Task:80:mike150_3:WQ_CPT=4 WQ_NUMACTL="numactl --physcpubind=12-15 -- " /work/
       jalupo/WQ/Examples/Timing/wq_timing.sh /work/jalupo/WQ/Examples/Timing/chr23/
       chr23_707.bf
3  Worker:Timings:80:mike150_3:1449778034.03:1449778042.04:8.01
4  Worker:Stdout:80:mike150_3:True
```

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

6.3 MPI Processing - MrBayes

MrBayes is an application that can make use of **MPI** to speed up processing. Settings in the input files control have an impact on how many **MPI** processes can be used. In this example, the input files used were determined to work well on 16 processes. That means the task could be set up to run with 16 **MPI** processes. Given that the machine used for testing had 16 cores per node, it meant 1 worker could run per node. For the curious, the input files selected to build the input file list all end with `.bf`.

6.3.1 The wq_mb.sh Task Script

The task script requires setting up the environment appropriately for MrBayes (Listing 9). It needs some environment variables to contain paths to directories it relies on for data and library files. Note that towards the end 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 one approach for testing the script to verify the proper command line is generated before turning it loose for real.

Listing 9: MrBayes Task Script

```
1  #!/bin/bash
2
3  # Expect a pathname as the first argument.
4
5  FILE=$1
6  DIR=`dirname ${FILE}`
7  BASE=`basename ${FILE}`
8
9  # Set up to run as many MPI processes per tasks as number of cores
10 # per worker provided by WQ_CPT. Method here is relatively insensitive
11 # to the version of MPI being used.
12
13 PROCS=${WQ_CPT}
14 HOSTNAME=`uname -n`
15 HOSTLIST=""
16 for i in `seq 1 ${PROCS}`; do
17     HOSTLIST="${HOSTNAME},${HOSTLIST}"
18 done
19 HOSTLIST=${HOSTLIST%,*}
20
21 # Here we set the command line to use. It assumes that the executable,
22 # 'mb', is in the shell's PATH. We also use the numactl command
23 # provided in the WQ_NUMACTL variable.
24
25 CMD="${WQ_NUMACTL}"
26 CMD="${CMD} mpirun -host ${HOSTLIST} -np ${PROCS} mb"
27 CMD="${CMD} ${FILE} < /dev/null > ${BASE}.mb.log"
28
29 cd $DIR
30
31 # Clean out any cruft from possible previous run.
32
33 rm -f ${BASE}.[pt] ${BASE}.log ${BASE}.ckp ${BASE}.ckp~ ${BASE}.mcmc
34
35 # For testing purposes, use "if false". For production, use "if true"
36
37
```

```
38  if false ; then
39      eval "${CMD}"
40  else
41      echo "Would have executed the following command:"
42      echo "${CMD}"
43      T=`expr 2 + $RANDOM % 10`
44      echo "Sleeping for $T seconds."
45      sleep $T
46  fi
```

The **MPI** used in this example was 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. Since the **MPI** processes for a task are restricted to run on a common node by WQ, it is relatively easy to build the host list (lines 15–19).

The variable WQ_NUMACTL contains the **numactl** command which assures that the processes will run on unique cores. It is used here to make sure the **MPI** processes of multiple workers on the same nodes use unique cores.

6.3.2 The wq_mb.pbs PBS Script

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 10. This is going to be a heftier example and use 16 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 10: MrBayes PBS Script Prologue

```
1   #! /bin/bash
2   ###############################################################################
3   # Begin WQ prologue section.
4   ###############################################################################
5   #PBS -A hpc_enable05
6   #PBS -l nodes=4:ppn=20
7   #PBS -l walltime=0:10:00
8   #PBS -q workq
9   #PBS -N MrBayes_V0
10
11  # Make sure wq.py and wq-pbs.sh are executable and in PATH.
12
13  # We want 2 workers per node. This would yield 10-core MPI processes
14  # on a 20-core node (ppn/WPN).
15
16  WPN=2
17
18  # Set the working directory:
19
20  WORKDIR=/work/jalupo/WQ/Examples/MrBayes
21
22  # Name of the file containing the list of input files:
23
24  FILES=${WORKDIR}/yeasts.lst
25
26  # Start with task 1 (1st line):
27
28  START=1
29
30  # Name of the worker task script:
31
32  TASK=${WORKDIR}/wq_mb.sh
33
34  # Turn off(0)/on(1) verbose output:
35
36  VERBOSE=0
37
38  ###############################################################################
39  # End WQ prologue section.
```

```
40 ##########################################################################
41 # Begin WQ epilogue section.
42 ##########################################################################
43
44 wq-pbs.sh $0 $WPN $WORKDIR $FILES $START $TASK $VERBOSE $1
```

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.

6.3.3 The yeasts.lst Input List

The sub-directory `PostPredYeast` is assumed to contain all the data files, and will hold the task output files as well. The file `yeasts.lst`, whose name corresponds to the FILES setting, contains a list of 1,414 input file names. The the first few lines are shown in Listing 11.

Listing 11: Sample MrBayes Files Entries

```
1 /work/jalupo/WQ/Examples/MrBayes/PostPredYeast/YDR449C_DNA/SeqOutfiles/YDR449C_DNA.nex.
      run3_1360000/GTRIG.bayesblock
2 /work/jalupo/WQ/Examples/MrBayes/PostPredYeast/YDR449C_DNA/SeqOutfiles/YDR449C_DNA.nex.
      run1_1960000/GTRIG.bayesblock
3 /work/jalupo/WQ/Examples/MrBayes/PostPredYeast/YDR449C_DNA/SeqOutfiles/YDR449C_DNA.nex.
      run2_3960000/GTRIG.bayesblock
4 /work/jalupo/WQ/Examples/MrBayes/PostPredYeast/YDR449C_DNA/SeqOutfiles/YDR449C_DNA.nex.
      run2_2360000/GTRIG.bayesblock
5 /work/jalupo/WQ/Examples/MrBayes/PostPredYeast/YDR449C_DNA/SeqOutfiles/YDR449C_DNA.nex.
      run3_2960000/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!

6.3.4 Execute

Once again `qsub` is used to submit the job.

```
$ qsub wq_mb.pbs > jid
$
```

This example received a job number of 426093, so the after execution, the contents appear as in Listing 13.

Listing 12: MrBayes Directory After Execution

```
1 find_cmd          MrBayes_V0.e426093   PostPredYeast   wq_mb.sh
2 hostlist.426093   MrBayes_V0.o426093   wq.log.426093   wq.py
3 jid               pbs.426093           wq_mb.pbs       yeasts.lst
```

The contents of `wq.log` are similar to the serial example, and the last few lines are shown in Listing **??**.

Listing 13: MrBayes WQ Execution Log

```
1 find_cmd          MrBayes_V0.e426093   PostPredYeast   wq_mb.sh
2 hostlist.426093   MrBayes_V0.o426093   wq.log.426093   wq.py
3 jid               pbs.426093           wq_mb.pbs       yeasts.lst
```

Looking at the Dispatcher:Shutdown line, we see 128 tasks were completed (faked in this case), the average task time was 6.42 seconds, with the shortest and longest being 2.01 and 11.02 seconds, respectively. The workers averaged run times of 108.28 seconds during the job, with the shortest and longest being 105.01 and 114.90 seconds respectively. Since the average is closer to minimum than the maximum run time, it suggests that a particularly long task occurred near the end of the job. In general, if the average appears closer to the maximum time, one can assume the CPU efficiency is better. That is, more cores say in use longer.

6.4 OpenMP Processing - BLASTN

BLASTN is an application that runs multi-threaded with OpenMP. What we'll do here is show how to run 4 workers per node, with each task using 4 threads (4 cores). The OpenMP thread count is specified by setting an appropriate environment variable, and the value will be that given by WQ_CPT.

6.4.1 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 14). 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. Just be aware that other BLASTN input files may require using only 2 or even just 1 thread per task.

Listing 14: BLASTN Task Script

```
#! /bin/bash

# Set a variable as the absolute path to an executable. The alternative
# would be to make sure 'blastn' is in the shell's PATH.

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

# BLASTN uses OpenMP, so we'll set up 1 thread per core allowed for workers.

export OMP_NUM_THREADS=${WQ_CPT}

# Treat the first command line argument as the input file pathname.

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

# Build the rather complex command line, including numactl string.

CMD="${WQ_NUMACTL} ${BLASTN} -task blastn -outfmt 7 -max_target_seqs 1"
CMD="${CMD} -num_threads ${OMP_NUM_THREADS}"
CMD="${CMD} -db /project/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":
# The random times simulate a spread in task times.

if false ; then
    eval "${CMD}"
else
    echo "Would have executed the following command:"
    echo "${CMD}"
    T=`expr 2 + $RANDOM % 10`
    echo "Sleeping for $T seconds."
    sleep $T
fi
```

6.4.2 The wq_blastn.pbs PBS Script

The processing of a single data set takes much longer than our earlier examples, so a real world run would take on the order of 70 hours. There is no reason one couldn't change the nodes=2 PBS option to use more nodes. On a busy cluster, the tradeoff might be a longer wait to get the job started (i.e. a *throughput* tradeoff). The WQ prologue is shown in Listing 15, and the epilogue section remains unchanged.

Listing 15: BLASTN PBS Prologue

```
#! /bin/bash
#############################################################################
# Begin WQ prologue section.
#
# This section is changed to meet system, user and application specific
# requirements, such as allocation codes, PBS options, etc.
#
# Please note the walltime here does not in any way representative
# of a BLASTN job.
#############################################################################
#PBS -A hpc_enable05
#PBS -l nodes=4:ppn=20
#PBS -l walltime=00:05:00
#PBS -q workq
#PBS -N BLASTN_V1

# Set up for 5 workers per node. This would yield 4 cores per worker
# on a 20-core node (ppn/WPN)

WPN=5

# Set the working directory. Could use $PBS_O_WORKDIR

WORKDIR=/work/jalupo/WQ/Examples/BLASTN

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

FILES=${WORKDIR}/fna_files

# Start with task 1 (first line):

START=1

# Name of the task script:

TASK=${WORKDIR}/wq_blastn.sh

# Turn verbosity off(0)/one(1):

VERBOSE=1

#############################################################################
# End WQ prologue section.
#############################################################################
# Begin WQ epilogue section.
#############################################################################

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

6.4.3 Contents of fna_files

The input files selected are those that end in `.fna`. The first few lines are shown in Listing 17

Listing 16: BLASTN Input Files

```
/work/jalupo/WQ/BLASTN/Data/GS049.blast/xab.fna
/work/jalupo/WQ/BLASTN/Data/GS049.blast/xak.fna
/work/jalupo/WQ/BLASTN/Data/GS049.blast/xai.fna
/work/jalupo/WQ/BLASTN/Data/GS049.blast/xag.fna
/work/jalupo/WQ/BLASTN/Data/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` (Listing **??**).

Listing 17: BLASTN Input Files

```
Data  find_cmd  fna_files  wq_blastn.pbs  wq_blastn.sh  wq.py
```

From this point, submit the job with `qsub` and revel in the copious output.

6.5 Multiple Command Line Arguments

It's now time to come clean and admit that requiring a single command line argument was a little white lie to simplify the early goings. The use of an input file with a list of file names is not the only way to use WQ, assuming you're willing to use a little more shell scripting magic. In fact, the line can be treated as containing multiple argumentss. It all depends on how the task script is set up to handle what is handed out by the dispatcher.

As an example, consider how you might approach a parameter sweep task, where each run of an analysis program processes a different set of parameters. If the main configuration is done with an input file, and the control parameters are specified on the command line for the program, you'd have trouble using WQ as described above. A few straight-forward changes can fix that. This is because WQ is really treating the line it reads from the input file as a string of characters and sends the entire line to the worker, which in turn passes the entire string on the command line when it calls the task. It is up to the task script to interpret what it receives.

By way of example, consider a hypothetical program that takes the muzzle elevation, pounds of black power, and cannon specifications data and returns the estimated range for a standard (?) cannon ball. It might be executed as so:

```
./range.sh -e 45.9 -p 1.5 cannon.dat
```

A parameter sweep would look at a range of angles and powder loads. For a particular set of cannon specifications defined in a file named `cannon.dat`, a list of commands covering 5 to 85 degrees of elevation in steps of 5 degrees, and 1.5 to 5.0 pounds of power in steps of 0.1 pounds could be generated by a small script shown in Listing 18:

Listing 18: Command Line Argument Generator

```
1  #! /bin/bash
2  echo "" > cmds2.lst
3  for elevation in `seq  5.0 5 85.0`; do
4     for pounds in `seq 1.5 0.1 5.0`; do
5        echo "$elevation $pounds" >> args.lst
6     done
7  done
```

A total of 612 command lines are generated, with the first shown in Listing 19.

Listing 19: Multiple Argument Input Lines

```
1  5.0 1.5
2  5.0 1.6
3  5.0 1.7
4  5.0 1.8
5  5.0 1.9
```

It is worth pointing out that the script and input file locations use relative paths. That means it is important to start the job in the proper work directory as there is no information provided to compute the work directory name as was done before.

The other important change is in the task script. Instead of using just 1 command line argument, it now expects any number of arguments. This is handled with the shell special variable $* which expands into all of the arguments (e.g. words) on the command line. (See Listing **??**)

Listing 20: Multiple Argument Task Script

```
1  #! /bin/bash
2  #
3  # This script expects two arguments on command line.
4
```

```
5  CANNON=dalhgren
6
7  # The directory must exist. We can't create it here because of a
8  # race condition with all the other workers.
9
10 if [ ! -d ${CANNON} ] ; then
11    echo "This directory must exist before running job: ${CANNON}"
12    exit 1
13 fi
14
15 CMD="./range.sh -e $1 -p $2 ${CANNON}.dat > ${CANNON}/${1}_${2}.dat"
16
17 # 'if true' execute the command. 'if false' just echo it back.
18
19 if true ; then
20    eval "${CMD}"
21 else
22    echo "CMD=${CMD}"
23    T=`expr 2 + $RANDOM % 10`
24    echo "Sleeping for $T seconds."
25    sleep $T
26 fi
```

The PBS script looks very similar to the serial case, with file names adjusted as needed. (See Listing 21)

Listing 21: Full Command Line Task Script

```
1  #! /bin/bash
2  ###############################################################################
3  # Begin WQ prologue section.
4  ###############################################################################
5  #PBS -A hpc_enable05
6  #PBS -l nodes=1:ppn=20
7  #PBS -l walltime=00:02:30
8  #PBS -q workq
9  #PBS -N MultiArg_V0
10
11 # Make sure wq.py and wq-pbs.sh are executable and in PATH.
12
13 # Set up for 20 workers per node
14
15 WPN=20
16
17 # Set the working directory:
18
19 WORKDIR=/work/jalupo/WQ/Examples/MultiArg
20
21 # Name of the file containing the command lines.
22
23 FILES=${WORKDIR}/args.lst
24
25 START=1
26
27 # Name of the task script.
28
29 TASK=${WORKDIR}/wq_multiarg.sh
30
31 # If set to 1, it turns on additional activity messages to help
32 # track the task assignment process.
33
34 VERBOSE=0
```

To make this even more realistic, let's specify a new script that does the range calculations, say `range.sh` (Listing 22).

Listing 22: Range Calculation Script

```
1  #! /bin/bash
2  #
```

```
3  # This does nothing but reflect the command line arguments.
4
5  echo "range.sh called. The arguments provided were:"
6  echo "args: $*"
```

Note that the script will write it's output to seperate files in the Output subdirectory. Since 612 files are created, it's best to put them someplace special so as not to clutter the working directory.

6.6 Full Command Line

One final example would be to pass a complete command line to the task script. This would allow almost any type of independent processing to be done. Using the cannon range parameter sweep idea, the relevent files show only small changes.

Listing 23: Full Command Line Prologue

```
1  #! /bin/bash
2  ##############################################################################
3  # Begin WQ prologue section.
4  ##############################################################################
5  #PBS -A hpc_enable05
6  #PBS -l nodes=1:ppn=20
7  #PBS -l walltime=00:02:30
8  #PBS -q workq
9  #PBS -N CmdLine_V1
10
11 # Make sure wq.py and wq-pbs.sh are executable and in PATH.
12
13 # Set up for 20 works (1 node * 20 cores):
14
15 WPN=20
16
17 # Set the working directory:
18
19 WORKDIR=/work/jalupo/WQ/Examples/CmdLine
20
21 # Name of the file containing the command lines.
22
23 FILES=${WORKDIR}/cmdlines.lst
24
25 START=1
26
27 # Name of the task script.
28
29 TASK=${WORKDIR}/wq_cmdline.sh
30
31 # If set to 1, it turns on additional activity messages to help
32 # track the task assignment process.
```

Listing 24: Generating Full Command Lines

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

Listing 25: First 5 Full Command Lines

```
1  ./range.sh -e 5.0 -p 1.5 cannon.dat
2  ./range.sh -e 5.0 -p 1.6 cannon.dat
3  ./range.sh -e 5.0 -p 1.7 cannon.dat
4  ./range.sh -e 5.0 -p 1.8 cannon.dat
5  ./range.sh -e 5.0 -p 1.9 cannon.dat
```

Listing 26: 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
```

Other approaches are possible, at the expense of more advanced shell scripting.

6.7 Workstation Usage

As alluded to at the very beginning, one can use WQ on a workstation or desktop. All of the PBS discussion can be ignored, and the effort to start processing boils down to 2 steps: starting the dispatcher and starting workers. For instance, the following two lines start a test of the `wq_timing` scripts on 2 workers running in a Debian Linux virtual machine:

```
python wq.py -d ./wq_timing.sh -a 2 -i 82_file_list -j noverbose &
python wq.py -w 2 -m Debian-VM
```

Since there are no PBS variables to play with, and the shell treats the job identifier as a character string, the `-j` option can be any word.

7 Installation

7.1 Single User

For use by a single individual, simply place `wq.py` and `wq-pbs.sh` somewhere in PATH, and make sure they are set executable. The one other requirement is having a Python that supports the `zmq` module also be in PATH. Try executing `wq.py` to check, as it will fail if `zmq` is not present. To run jobs, make copies of `wq.pbs` and `wq.sh`, and modify them as needed.

7.2 System Wide

For a system wide installation, the same general rules apply. If `module` is used, then a module file might look something like this:

Listing 27: Module App File

```
#%Module

proc ModulesHelp { } {
     puts stderr {
WQ (WorkQueuing) is a task distribution system designed to work with
PBS (or SLURM). It allows multiple serial, multi-threaded, and/or
multi-process MPI apps to run simultaneously on available compute nodes.
An input file defines the data files, or command line arguments, that a
a processing application requires. The user specifies this file name,
the name of the command to execute, and 4 other options. When submitted
```

```
11  to PBS, a dispatcher starts workers on the nodes assigned to the job, then
12  hands out one task at a time as they are requested by the workers. It tracks
13  job wallclock time, and stops handing out work if it estimates there is
14  insufficient time for a task to complete. The only constraint is that a
15  task may only use the cores on one node.
16
17  To use, you'll need the PBS template ($WQ_PBS_TEMPLATE). Make a copy and
18  edit to fit your application needs. A sample task script ($WQ_EXAMPLE_TASK)
19  and user documentation ($WQ_DOC) are also available.
20    }
21  }
22
23  module-whatis {
24  Description: WQ is a support tool that creates supports a dispatcher/worker
25  model of distributed computing. Serial, multi-thread OpenMP, and small
26  multi-process MPI are supported. Tasks are defined by input file names,
27  or other command line strings, contained in an input file - one line per
28  task.
29
30  To use, you'll need the PBS template ($WQ_PBS_TEMPLATE). Make a copy and
31  edit to fit your application needs. A sample task script ($WQ_EXAMPLE_TASK)
32  and user documention ($WQ_DOC) are also available.
33  }
34
35  set root /usr/local/packages/wq/257
36
37  conflict wq
38
39  if { [module-info mode load] && ![is-loaded python] } {
40      module load python
41  }
42
43  prepend-path PATH $root
44
45  setenv WQ_HOME $root
46  setenv WQ_PBS_TEMPLATE $root/wq.pbs
47  setenv WQ_EXAMPLE_TASK $root/wq.sh
48  setenv WQ_DOC $root/wq.pdf
```

This example defines a WQ root directory, and makes the PBS template, an example task script, and the user documentation available to copy. It also holds `whatis` and `help` text strings. Finally, it loads Python, which is assumed to have the `zmq` module installed.

The same sort of thing can be done with `softenv`, although it has less room for documentation:

Listing 28: Softenv Key File

```
1  (+wq-264) {
2  {desc:
3      @types: Application/Tools
4      @name: WQ
5      @version: 264
6      @internal:
7      @build:
8      @external:
9      @about: WQ (WorkQueuing) provides a task dispatcher-worker service. To use, you'll
           need the PBS template ($WQ_PBS_TEMPLATE). Just make a copy and edit to fit your
           application needs. A sample task script ($WQ_EXAMPLE_TASK) and user documentation
            ($WQ_DOC) are also available.
10 }
11 [Linux] {
12   PATH /usr/local/packages/WQ/264
13   WQ_HOME /usr/local/packages/WQ/264
14   WQ_PBS_TEMPLATE /usr/local/packages/WQ/264/wq.pbs
15   WQ_EXAMPLE_TASK /usr/local/packages/WQ/264/wq.sh
16   WQ_DOC /usr/local/packages/WQ/264/wq.pdf
17   }
18 }
```

The WQ distribution contains these files to use as starting points.

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.