WQ — A Task Distributor: The User Manual

Document Date: 11 August 2015
Program Version: SVN Revision 192

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 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 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 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 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. 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 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 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, 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. 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 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 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 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 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 concerted. 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 it is responsible for creating the dispatcher and worker services. That is what 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 opt1 [opts ...]

   As Dispatcher:

        python -d[--dispatcher] cmd [-s[--start] task_num] \
               [-t[--time] walltime] [-v[--verbose]] -a[--allworkers] n \
               -i[--input] filenm

   As Worker:

        python -w[--workers] n -m[--mothersuperior] ms \\
               [-v[--verbose]]

   Help:
```

```
     python 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.

   -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)

   -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.

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.py 171 2015-01-29 21:26:04Z 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 long 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, 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 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 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.

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.

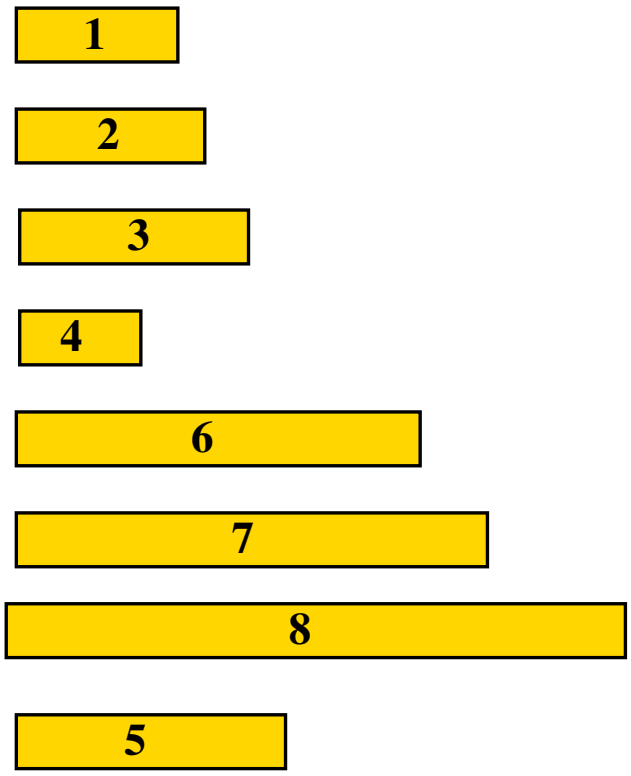

Figure 1: A collection of 8 tasks. The required walltime for each is proportional to the length of each task's bar. The box numbers are meant for identification and not necessarily the order in which the tasks will be assigned.

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 recognized 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 ##############################################################################
```

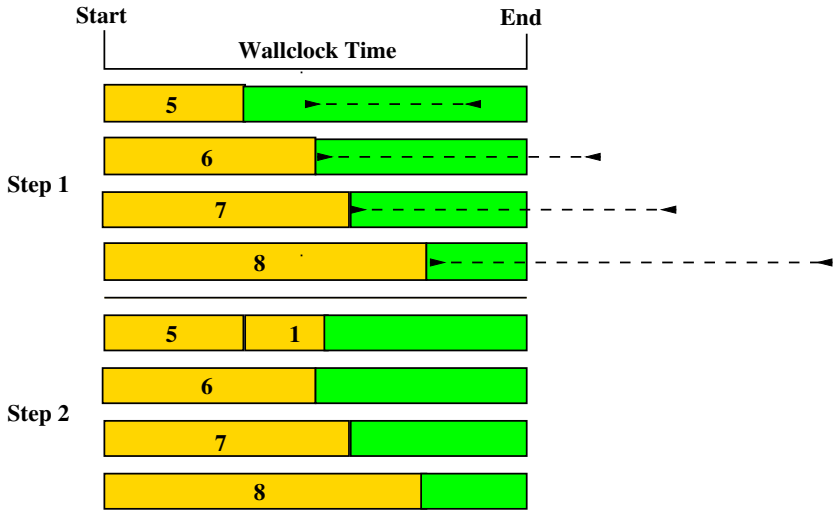

Figure 2: A successful job scenario. Step 1 shows the first set of tasks to be executed. The green boxes indicate the actual walltime remaining. The dashed lines indicates the maximum walltime to allow for the next job, based on what has completed so far.

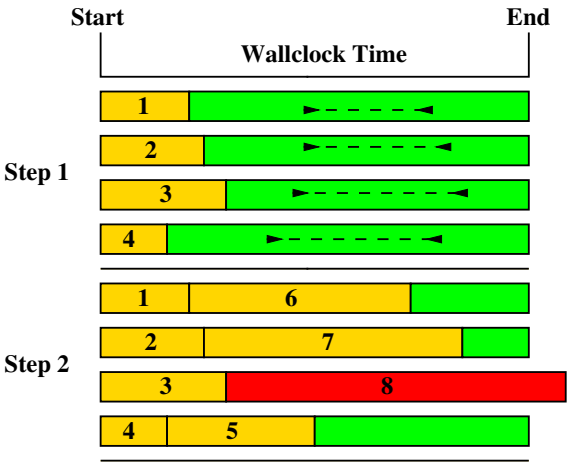

Figure 3: A failed task/job scenario. Step 1 shows the first set of tasks to be executed. The green boxes indicate the actual walltime remaining. The dashed lines indicates the maximum walltime to allow for the next job, based on what has completed so far. The red box indicates a task that was started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

```
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 # If set to 1, it turns on additional activity messages to help
42 # track the task assignment process.
43
44 VERBOSE=0
45
46 ###############################################################################
47 # End WQ prologue section.
```

The PBS section is followed by shell commands which set 6 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.

VERBOSE Can be used to increase the information output by the dispatcher and workers. 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.

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

```
# End WQ prologue section.
#
# Begin WQ epilogue section.
# What follows is the main WQ script.  It should be considered powerful
# magic. Dabbled with at your own peril.
##############################################################################

# Set verbosity flag, if desired.

if [ "$VERBOSE" = "1" ] ; then
   VERBOSE=--verbose
else
   VERBOSE=
fi

# Drop into the working directory after making sure it exists.

if [ ! -d ${WORKDIR} ] ; then
   echo "WQ.PBS Error: WORKDIR = \"${WORKDIR}\" does not exist!"
   exit 1
fi

cd ${WORKDIR}

# Only the mother superior has PBS_JOBID defined, so we will be
# passing it to the other nodes as $2. Use this fact to decide if
# we are running on the mother superior or a compute node:

if [ "${2}x" = "x" ] ; then

   # Must be running on the mother superior. Do some basic sanity
   # checking just to be safe.

   if [ ! -r ${FILES} ] ; then
      echo "WQ.PBS Error: FILES = \"${FILES}\" does not exist or can't be read!"
      exit 1
   fi

   if [ $(wc -l ${FILES} | cut -d ' ' -f 1) -lt 1 ] ; then
      echo "WQ.PBS Warning: FILES = \"${FILES}\" is empty. No work to do!"
      exit 0
   fi

   if [ ! -x ${TASK} ] ; then
      echo "WQ.PBS Error: TASK = \"${TASK}\" does not exist or isn't executable!"
      exit 2
   fi

```

```
49      if [ ${START} -lt 1 ] ; then
50         echo "WQ.PBS Error: START can't be less than 1! Quiting!"
51         exit 3
52      fi
53
54      # Make sure the dispatcher port is available.
55
56      PORT=54321
57      P=`netstat -lt | gawk 'BEGIN{x=0;}/'$PORT'/{x=1;}END{print x;}'`
58      if [ $P = '1' ] ; then
59          echo "WQ.PBS Error: Port $PORT is busy?!? Quiting!"
60          exit 4
61      fi
62
63      # Remember our host name.
64
65      MS=`uname -n`
66
67      # Use a bit of magic to strip off the trailing host name and
68      # leave only the job number from PBS_JOBID:
69
70      JOBNUM=${PBS_JOBID%.*}
71      HOSTLIST=${WORKDIR}/hostlist.${JOBNUM}
72
73      # We want the mother superior host name first. So, take the host
74      # list provided, sort it into a unique list of names, with MS first.
75      # This assures it's node ID, or position in the hostlist, is 1.
76
77      echo ${MS} > ${HOSTLIST}
78      grep -v ${MS} ${PBS_NODEFILE} | uniq | sort >> ${HOSTLIST}
79
80      # Compute the number of nodes assigned.
81
82      export NODES=`wc -l ${HOSTLIST} |gawk '//{print $1}'`
83
84      # Make a local copy of the PBS script since only the mother superior
85      # can see it at job start.
86
87      JOBFILE=${WORKDIR}/pbs.${JOBNUM}
88      cp $0 $JOBFILE
89      chmod a+x ${JOBFILE}
90
91      # Mother superior must start up the dispatcher, so:
92
93      python ${WORKDIR}/wq.py --start $START --dispatcher ${TASK} \
94          $VERBOSE --inputs ${FILES} --allworkers $(( $WPN * $NODES )) &
95
96      # Give it a chance to spin up since the dispatcher must be ready
97      # to accept connections from the workers upon request.
98
99      sleep 5
100
101     # Ready to start the script on all compute nodes. This will fire up
102     # workers. We'll pass PBS_WALLTIME and the job number as arguments.
103     # They'll connect to the dispatcher and start work immediately.
104
105     for H in `cat ${HOSTLIST}` ; do
106        if [ ${H} != ${MS} ] ; then
107           ssh -n ${H} ${JOBFILE} ${PBS_WALLTIME} ${JOBNUM} &
108        fi
109     done
110
111     # Finally, mother superior can also start workers:
112
113     python ${WORKDIR}/wq.py --workers ${WPN} --mothersuperior ${MS} \
114         $VERBOSE --time ${PBS_WALLTIME}
115
116     # Make sure to wait until all the processes are done!
117
118     wait
```

```
119
120 else
121
122     # Must be running on a compute node. The job number was passed by the
123     # mother superior (see above).
124
125     HOSTLIST=${WORKDIR}/hostlist.$2
126
127     # Now, we have to get the name of mother superior from the host
128     # list. Thats so we know where the dispatcher is running. Simply
129     # grab the first entry from the hostlist file and press on.
130
131     MS=`head -1 ${HOSTLIST}`
132
133     # Ready to go. Spin up the workers. The mother superior passed
134     # the job wall time as argument 1 when the script is called, so
135     # we have all the values needed for workers:
136
137     python ${WORKDIR}/wq.py --workers ${WPN} --mothersuperior ${MS} \
138         $VERBOSE --time $1
139
140 fi
141
142 # Give a bit of time to make sure dispatcher has shut down cleanly.
143 # The WAIT above should allow for this, but coming down on the side of
144 # paranoia:
145
146 sleep 2
```

The epilogue section starts by performing some sanity check to make sure everything is consistent and runnable. You'll then 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 is allowed to terminate.

At this point it is useful to discuss several examples of using `wq.py`. The 3 examples that follow illustrate how purely serial jobs, multiple process **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 Output

WQ sends information to both `stderr` and `stdout`, which is the reason 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. `stderr` output pretty much contains timing and task distribution information (i.e. who was assigned what when), while STDOUT contains results from each task. If more than a few lines of output is expected from the tasks, it is strongly recommended that the script direct that output to task specific output files. The amount of output can be controlled by the `wq.py --verbose` option.

5.1 Workers

Only workers send anything to `stdout`. The information they write out has to do primarily with task execution and any results. Each type of line is described below. In some cases, the worker identification is provided, which takes the form: `hostname_worker#` (e.g. `mike013_14`, or node `mike013`, worker number 14). Any system time is reported in seconds. This is usually elapsed time *since the epoch*, or 1 Jan 1970.

5.1.1 stdout

- There are 2 types of lines written out by default (`VERBOSE=0`):

  `Worker:Stdout:N:mode:status` — Reports the task number (`N`), the task handling mode (`Ran` or `Skipped`), and an indication if the task was successful or not (`True` or `False`). This is then followed by any lines the task wrote to its `stdout`.

  `Worker:Stderr:N` — Reports just the task number (`N`), and is followed by any lines the task wrote to its `stderr`.

- There are two additional optional lines that appear if verbose output is requested (`VERBOSE=1`):

  `Worker:Task:N:W:task` — Reports the task number (`N`), the ID of the worker (`W`) who handled it, and the full task command line that was executed.

  `Worker:Timings:N:W:ts:te:tt:wt` — Reports the task number (`N`), the ID of the worker (`W`) who handled it, the task start system time (`te`), task end system time (`tt`), and the elapsed walltime (`wt`).

5.1.2 stderr

- There is 1 type of line written out by default (`VERBOSE=0`):

  `Worker:Shutdown:W:T` — Reports the worker ID (`W`) and the system time (`T`) at which it stopped processing.

- There are 4 additional optional lines produced if verbose output is enabled (`VERBOSE=1`):

  `Worker:Startup:W:T` — Reports the worker ID (`W`) and the system time (`T`) at which it started processing.

  `Worker:LastTask:W:N` — Reports the worker ID (`W`) and the last task number (`N`) it processed. A task number of 0 indicates the worker just started, hence no task has been processed.

  `Worker:Taking:W:N:T` — Reports the worker ID (`W`) and that it accepted task number `N` at system time `T`.

  `Worker:Affinity:W:NUMACTL_PREFIX="numactl --physcpubinding=m-n -- "` — Reports the NUMA control setting to be used to assure unique core access by multi-threaded and MPI based tasks. Refer to the **numactl** man page for details.

5.1.3 NUMACTL_PREFIX

When the workers call the task script, they define the environment variable NUMACTL_PREFIX. This is set to a **numactl** command that restricts whatever it runs to a specific set of cpu cores. This is likely needed when ever 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 NUMACTL_PREFIX would involve just:

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

The examples that follow show deployment in more realistic situations.

5.2 Dispatcher

The dispatcher only sends output to `stderr`. It all has to do with task assignments and time tracking.

- There are 4 types of lines written out by default (`VERBOSE=0`):

  `Dispatcher:Start:N:T` — Reports the first task number (`N`) handed out (should equal `START` value), and the system time (`T`) at which it started processing.

  `Dispatcher:Timeup:T` — Reports the system time (`T`) at which it estimated that there was insufficient walltime remaining to start other tasks.

  `Dispatcher:Last:N` — Reports the last successful task processed (`N`).

  `Dispatcher:Shutdown:T` — Reports the system time (`T`) at which it stopped processing.

- There are 4 types of optional lines written if verbose is enabled (`VERBOSE=1`):

  `Dispatcher:File:F:N:W` — Reports the file name (`F`), task number (`N`), and ID of the worker (`W`) it was assigned to.

  `Dispatcher:Maxtime:W:E:T` — Reports the worker ID (`W`), the elapsed task time (`E`), and the system time (`T`) when a new maximum elapsed task time is noted.

  `Dispatcher:WaitOn:N` — Reports the dispatcher will have to wait for `N` workers to complete at the end of a job.

  `Dispatcher:WaitFor:W1:...:W6` — Worker ID's (`W's`) being waited on. Will list all workers, 6 names per line.

6 WQ Examples

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 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.

6.2 Serial Task - Simple Timing

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 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: Serial 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=${BASE}"
15 echo "NUMACTL_PREFIX=${NUMACTL_PREFIX}"
16 echo "That's all, folks!"
17 T=`expr 1 + $RANDOM % 10`
18 echo "Sleeping for $T seconds."
19 sleep $T
```

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
NUMACTL_PREFIX=
  That's all, folks!
  Sleeping for 5 seconds.
  $
```

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 value of NUMACTL_PREFIX is empty, since this is a variable created at run time by the workers when they call the script.

6.2.2 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, thus 16 cores total. The example is requesting 1 node, so 16 cores. 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 4: Serial Job Prologue Section

```
#! /bin/bash
#############################################################################
# Begin WQ prologue section.
#############################################################################
#PBS -A hpc_enable03
#PBS -l nodes=1:ppn=16
#PBS -l walltime=00:02:30
#PBS -q workq
#PBS -o /work/jalupo/WQ/Timing
#PBS -N WQ_Timing

# Things that should be customized, carefully of course.
# First, number of workers per node:

WPN=16

# Set the working directory:

WORKDIR=/work/jalupo/WQ/Timing

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

FILES=${WORKDIR}/82_file_list

# Set the starting line in the file.  The default is 1 (i.e. start
# from the beginning), but just to be explicit.

START=1

# Name of the task script:

TASK=${WORKDIR}/wq_timing.sh

# Turn verbose output off(0) or on(1):

VERBOSE=0

#############################################################################
# End WQ prologue section.
```

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`. 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 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 the following prior to running the job:

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

6.2.4 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 **296158.mike3**. The job number is 296158 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:

```
$ls
82_file_list  hostlist.296158  jid  pbs.296158  wq.py  wq_timing.e296158
wq_timing.o296158  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.296158` and `pbs.296158`.

`hostlist.296158` 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.296158` 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.o296158` 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.e296158` 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 first few lines of the standard output file (Listing 5). Refer to the line content descriptions above if you need to:

Listing 5: Serial Example Execution

```
1  Worker:Stdout:13:Ran:True
2    DIR=/work/jalupo/DCL/0MQ/Brown/chr13; BASE=chr13_707.bf
3          NUMACTL_PREFIX=numactl --physcpubind=12-12 --
4    That's all, folks!
5    Sleeping for 1 seconds.
6  Worker:Stderr:13
7
8  Worker:Stdout:16:Ran:True
9    DIR=/work/jalupo/DCL/0MQ/Brown/chr17; BASE=chr17_2984.bf
10         NUMACTL_PREFIX=numactl --physcpubind=15-15 --
11   That's all, folks!
12   Sleeping for 1 seconds.
13 Worker:Stderr:16
14
15 Worker:Stdout:10:Ran:True
16   DIR=/work/jalupo/DCL/0MQ/Brown/chr13; BASE=chr13_2821.bf
17         NUMACTL_PREFIX=numactl --physcpubind=9-9 --
18   That's all, folks!
19   Sleeping for 1 seconds.
20 Worker:Stderr:10
```

The standard error contains timing information that may be useful. With verbose set of, the file is rather short (Listing 6)

Listing 6: Serial `stderr VERBOSE=0` Example

```
1  Dispatcher:Start:1:1422475181.25
2  Worker:Shutdown:mike305_0:1422475215.27
```

```
Worker:Shutdown:mike305_13:1422475216.28
Worker:Shutdown:mike305_4:1422475216.28
Worker:Shutdown:mike305_12:1422475217.27
Worker:Shutdown:mike305_7:1422475218.27
Worker:Shutdown:mike305_1:1422475218.27
Worker:Shutdown:mike305_3:1422475221.27
Worker:Shutdown:mike305_14:1422475221.28
Worker:Shutdown:mike305_8:1422475221.29
Worker:Shutdown:mike305_15:1422475221.29
Worker:Shutdown:mike305_6:1422475221.29
Worker:Shutdown:mike305_9:1422475222.27
Worker:Shutdown:mike305_2:1422475223.27
Worker:Shutdown:mike305_5:1422475223.28
Worker:Shutdown:mike305_10:1422475223.28
Dispatcher:Last:82
Dispatcher:Shutdown:1422475225.28
Worker:Shutdown:mike305_11:1422475225.28
```

Notice that even for this small job, we're able to see some skew in the ending times. Worker `mike305_0` stopped almost 8 seconds before worker `mike305_10`.

If the verbose option is turned on, more detail is provided. The first few line near the beginning and end are shown in Listings 7 and 8.

Listing 7: Serial `stderr` `VERBOSE=1` Top Lines

```
Dispatcher:Start:1:1422475179.48
Worker:Startup:mike156_0:1422475184.07
Worker:Startup:mike156_1:1422475184.07
Worker:LastTask:mike156_0:0
Worker:LastTask:mike156_1:0
Worker:Startup:mike156_2:1422475184.07
Worker:LastTask:mike156_2:0
Worker:Startup:mike156_3:1422475184.07
Worker:LastTask:mike156_3:0
Worker:Startup:mike156_4:1422475184.07
Worker:LastTask:mike156_4:0
Worker:Startup:mike156_5:1422475184.07
Dispatcher:File:/work/jalupo/DCL/0MQ/Brown/chr13/chr13_710.bf:1:mike156_0
Worker:LastTask:mike156_5:0
Dispatcher:File:/work/jalupo/DCL/0MQ/Brown/chr13/chr13_727.bf:2:mike156_4
...
    skipping
        ...
```

Listing 8: Serial `stderr` `VERBOSE=1` Bottom Lines

```
...
    many skipped lines
                ...
Dispatcher:File:/work/jalupo/DCL/0MQ/Brown/chr24/chr24_1019.bf:82:mike156_3
Dispatcher:WaitOn:16
Dispatcher:WaitFor:mike156_14:mike156_12:mike156_15:mike156_13:mike156_2:mike156_3
Dispatcher:WaitFor:mike156_0:mike156_1:mike156_6:mike156_7:mike156_4:mike156_5
Dispatcher:WaitFor:mike156_10:mike156_8:mike156_9:mike156_11
Worker:Taking:mike156_3:82:1422475215.15
Worker:LastTask:mike156_8:61
Worker:Shutdown:mike156_8:1422475215.16
Worker:LastTask:mike156_4:71
Worker:Shutdown:mike156_4:1422475215.16
Worker:LastTask:mike156_13:76
Worker:Shutdown:mike156_13:1422475215.17
...
    more lines skipped
                ...
Worker:Shutdown:mike156_0:1422475221.17
Worker:LastTask:mike156_5:79
Dispatcher:Last:82
Dispatcher:Shutdown:1422475221.17
Worker:Shutdown:mike156_5:1422475221.17
```

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 (lines 32–38) 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  FILE=$1
4  DIR=`dirname ${FILE}`
5  BASE=`basename ${FILE}`
6
7  # We are going to run 16 processes per task, so create a list with the
8  # local hostname appearing 16 times.  Just append the name, with a
9  # separating ",", then trim off any final ",".
10
11 PROCS=16
12 HOSTNAME=`uname -n`
13 HOSTLIST=""
14 for i in `seq 1 ${PROCS}`; do
15    HOSTLIST="${HOSTNAME},${HOSTLIST}"
16 done
17 HOSTLIST=${HOSTLIST%,*}
18
19 # Here we set the command line to use, and include NUMACTL_PREFIX
20
21 CMD="${NUMACTL_PREFIX}"
22 CMD="${CMD} mpirun -host ${HOSTLIST} -np ${PROCS} mb < ${FILE} > ${BASE}.mb.log"
23
24 cd $DIR
25
26 # Clean out files from any previous run.
27
28 rm -f *.[pt] *.log *.ckp *.ckp~ *.mcmc
29
30 # For testing purposes, use "if false". For production, use "if true"
31
32 if true ; then
33    eval "${CMD}"
34 else
35    echo "${CMD}"
36    echo "Faking It On Hosts: ${HOSTLIST}"
37    sleep 2
38 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 12–17).

The variable NUMACTL_PREFIX contains a **numactl** command which assures that the processes will run on unique cores.

6.3.2 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 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

```
#! /bin/bash
###############################################################################
# Begin WQ prologue section.
###############################################################################
#PBS -A hpc_enable03
#PBS -l nodes=16:ppn=16
#PBS -l walltime=5:00:00
#PBS -q workq
#PBS -N WQ_MrBayes

# We want 1 worker running 16 \mpi processes, so:

WPN=1

# Set the working directory:

WORKDIR=/work/jalupo/WQ/MrBayes

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

FILES=${WORKDIR}/yeasts.lst

# Start with task 1 (1st line):

START=1

# Name of the worker task script:

TASK=${WORKDIR}/wq_mb.sh

# Turn off(0)/on(1) verbose output:

VERBOSE=1

###############################################################################
# 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.

6.3.3 The yeasts.lst Input List

The sub-directory `PostPredYeast` contains all the data files, and will hold the task output files as well. The file `yeasts.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!

6.3.4 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 `296682.mike3`. Hence, when it completed, the contents of the `WORKDIR` looked as so:

```
$ ls
PostPredYeast  hostlist.296682 pbs.296682 wq.lst  wq_mb.pbs  wq_mb.sh
wq.py WQ_MrBayes.e296682  WQ_MrBayes.o296682
```

The contents of `WQ_MrBayes.e296682` are similar to the serial example, but there are a few differences, especially since there wasn't enough walltime allowed for all the tasks to run. Note that only 65 completed:

```
Dispatcher:Start:1:1422568712.63
Worker:Startup:mike055_0:1422568716.99
Worker:LastTask:mike055_0:0
Dispatcher:File:/work/.../run3_1360000/GTRIG.bayesblock:1:mike055_0
Worker:Taking:mike055_0:1:1422568717.00
Worker:Startup:mike124_0:1422568718.11
Worker:LastTask:mike124_0:0
. . .
     (removed for brevity)
                            . . .
Worker:Taking:mike124_0:65:1422572903.44
Worker:LastTask:mike107_0:54
Dispatcher:Timeup:mike107_0:1422572914.62
Dispatcher:WaitOn:15
Dispatcher:WaitFor:mike133_0:mike109_0:mike166_0:mike073_0:mike057_0:mike074_0
Dispatcher:WaitFor:mike055_0:mike124_0:mike101_0:mike108_0:mike113_0:mike118_0
Dispatcher:WaitFor:mike069_0:mike107_0:mike136_0:mike075_0
Worker:Shutdown:mike107_0:1422572914.62
. . .
     (removed for brevity)
                            . . .
Dispatcher:Last:65
Dispatcher:Shutdown:1422573963.89
Worker:Shutdown:mike124_0:1422573963.89
```

It appears that once worker `mike107_0` completed a task and asked for another, the dispatcher realized there might not be enough time to complete it, hence it started telling workers to shut down. The dispatcher reports that 15 workers are still processing, along with the work ID's. It also reports that the last task completed was number 65.

The difference in end times between the first worker to shut down (`mike107_0`) and the last (`mike124_0`) is about 1049 secs, which illustrates how the individual task times can impact the overall job time. Taking a peek in the job's `stdout` file (`WQ_MrBayes.o296682`), one sees the line:

```
Resources Used:  cput=19:37:14,mem=426328kb,vmem=6085376kb,walltime=01:27:36
```

The average CPU time consumed per node was about 1:12:57, suggesting this run used 83% of the available CPU time. Not too bad for a short job.

6.4 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 on 4 of the 16 cores available. The OpenMP thread count is specified by setting an appropriate environment variable.

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 11). 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 11: BLASTN Task Script

```
#! /bin/bash

# Set a variable as the absolute path to the BLASTN executable.

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

# Specify use of 4 threads.

export OMP_NUM_THREADS=4

# Capture the input file from the argument list, and split it into
# the directory path part, and the basename part (name less extension).

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

# Build up the rather complex command line.

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

# Execute the desired processing steps. Again, more than one is
# possible. This example runs BLASTN against a particular database.
# For testing purposes, use "if false". For real runs, use "if true":

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

6.4.2 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 2 node for 70 hours. That means a total of 8 workers will run at any given time. If more nodes are desired, just change the nodes=2 PBS option.

Listing 12: BLASTN PBS Script

```
#! /bin/bash
#################################################################################
```

```
3  # Begin WQ prologue section.
4  ##############################################################################
5  #PBS -A hpc_enable03
6  #PBS -l nodes=2:ppn=16
7  #PBS -l walltime=00:05:00
8  #PBS -q workq
9  #PBS -N WQ_Blastn
10 #PBS -o /work/jalupo/WQ/BLASTN
11
12 # We want 4 4-threaded tasks (workers) per node.
13
14 WPN=4
15
16 # Set the working directory:
17
18 WORKDIR=/work/jalupo/WQ/BLASTN
19
20 # Name of the file containing the list of input files:
21
22 FILES=${WORKDIR}/fna_files
23
24 # Start with task 1 (first line):
25
26 START=1
27
28 # Name of the task script:
29
30 TASK=${WORKDIR}/wq_blastn.sh
31
32 # Turn verbosity off(0)/one(1):
33
34 VERBOSE=1
35
36 ##############################################################################
37 # End WQ prologue section.
```

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

6.4.3 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. The job is executed just as we did with the previous 2 examples. Since there's nothing new to see in the output files, we'll end the discussion of the examples here.

6.5 Advanced Uses

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. 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. With a few straight-forward changes, it is possible to send complete command lines to a task script! This is because WQ really sends the entire line to the worker, so the line can contain anything. Of course, the script must know what to do with the line when it receives it.

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 degress, 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 13:

Listing 13: Command Line Generator

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

A total of 612 command lines are generated, with the first few looking like:

```
./range.sh -e 5.0 -p 1.5 cannon.dat
./range.sh -e 5.0 -p 1.6 cannon.dat
./range.sh -e 5.0 -p 1.7 cannon.dat
```

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 gives all of the arguments. The task script then looks like:

```
#! /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
```

The PBS script looks very similar to the serial case, with file names adjusted as needed. After execution, the verbose STDOUT output would look like List 14:

Listing 14: Verbose Advanced STDOUT

```
1  . . . skipping . . .
2  Worker:Task:3:mike007_1:/work/jalupo/WQ/Advanced/wq_advanced.sh ./range.sh -e 5.0 -p 1.7 cannon.dat
3  Worker:Timings:3:mike007_1:1423584398.22:1423584401.24:3.02:3.02
4  Worker:Stdout:3:Ran:True
5    CMD=./range.sh -e 5.0 -p 1.7 cannon.dat
6     Sleeping for 3 seconds.
7  Worker:Stderr:3
8
9  Worker:Task:1:mike007_0:/work/jalupo/WQ/Advanced/wq_advanced.sh ./range.sh -e 5.0 -p 1.5 cannon.dat
10 Worker:Timings:1:mike007_0:1423584398.22:1423584401.24:3.02:3.02
11 Worker:Stdout:1:Ran:True
12   CMD=./range.sh -e 5.0 -p 1.5 cannon.dat
13    Sleeping for 3 seconds.
14 Worker:Stderr:1
15 . . . skipping . . .
```

Just to hint at what else might be possible, one could generate only the elevation and poundage arguments and use them explicitly in a script. The argument list generation script would look more like:

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

And the task script would become:

```
#! /bin/bash
#
# This script expects the elevation angle as argument 1, and the
# powder poundage as argument 2.

# Specify the cannon name. The file name.dat is expected to have
# the specifications for the cannon.

CANNON=dalhgren

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

# '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.

7 Some Sanity Checks

7.1 Task Times

We saw above that there was a spread of 1049 seconds in the worker end times. We could write a script to analyze the `Worker:Timings` lines and see what sort of spread there was in the task run times. A Python script to do that would look like:

Listing 15: Task Run Time Script

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

The script demonstrates how to use the colon delimited fields, in this case for lines found in the `stdout` file. Running this script produces:

```
$ ./tasktimes.py WQ_MrBayes.o296682
Task times between 983.38 and 1229.86 - 246.48 sec spread.
```

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

7.2 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 16 determines the earliest and latest worker end times from the `stderr` file, similar to what was pointed out manually in the discussion of the MrBayes example.

Listing 16: Worker End Time Script

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

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

for l in raw:
   u = l.strip().split(':')
   if u[0] == 'Worker' and u[1] == 'Shutdown' :
      t = float(u[3])
      if t > 0.0 :
         worker[u[2]] = t

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

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

Let's apply this script (make sure it is set executable!) to the `stderr` of the MrBayes example:

```
$ ../timing.py WQ_MrBayes.e296682
End times between 1422572914.62 and 1422573963.89 - 1049.27 sec spread.
$
```

Given that the maximum task time was 1229 sec, having all workers finish within less than the maximum task time is pretty good.

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.