Accessing Outside Computing Resources

Overview

Tools are hosted within a "tool session" running within the hub environment. The tool session supports the graphical interface, which helps the user set up the problem and visualize results. If the underlying calculation is fairly light weight (e.g., runs in a few minutes or less), then it can run right within the same tool session. But if the job is more demanding, it can be shipped off to another machine via the "submit" command, leaving the tool session host less taxed and more responsive.

This chapter describes the "submit" command, showing how it can be used at the command line within a workspace and also within Rappture-based tools.

Submit Command

Overview

submit takes a user command and executes it remotely. The objective is to allow the user to issue a command in the same manner as a locally executed command. Multiple submission mechanisms are available for run dissemination. A set of steps are executed for each run submission:

- Destination site is selected
- A wrapper script is generated for remote execution
- If needed a batch system description file is generated.
- Input files for a run are gathered and transferred to the remote site. Transferred files include the wrapper script and batch description scripts.
- The wrapper script is executed remotely.
- Progress of the remote run is monitored until completion.
- Output files from the run are returned to the dissemination point.

Command Syntax

submit command options can be determined by using the help parameter of the submit command.

```
$ submit --help
Usage: submit [options]

Options:
  -h, --help            Report command usage. Optionally request listi
ng of
                        managers, tools, or venues.
  -l, --local           Execute command locally
  -v, --venue           Remote job destination
  -i, --inputfile       Input file
  -p, --parameters      Parameter sweep variables. See examples.
  -d, --data            Parametric variable data - csv format
  -s SEPARATOR, --separator=SEPARATOR
                        Parameter sweep variable list separator
  -n NCPUS, --nCpus=NCPUS
                        Number of processors for MPI execution
  -N PPN, --ppn=PPN     Number of processors/node for MPI execution
  -w WALLTIME, --wallTime=WALLTIME
                        Estimated walltime hh:mm:ss or minutes
  -e, --env             Variable=value
  -m, --manager         Multiprocessor job manager
  -r NREDUNDANT, --redundancy=NREDUNDANT
```

```
                        Number of indentical simulations to execute in
                        parallel
  -M, --metrics         Report resource usage on exit
  -W, --wait            Wait for reduced job load before submission
  -Q, --quota           Enforce local user quota on remote execution h
ost
  -q, --noquota         Do not enforce local user quota on remote exec
ution
                        host

Parameter examples:

submit -p @@cap=10pf,100pf,1uf sim.exe @:indeck

        Submit 3 jobs. The @:indeck means "use the file indeck as a te
mplate
file." Substitute the values 10pf, 100pf, and 1uf in place of @@cap wi
thin the
file. Send off one job for each of the values and bring back the resul
ts.

submit -p @@vth=0:0.2:5 -p @@cap=10pf,100pf,1uf sim.exe @:indeck

        Submit 78 jobs. The parameter @@vth goes from 0 to 5 in steps
of 0.2,
so there are 26 values for @@vth. For each of those values, the parame
ter
@@cap changes from 10pf to 100pf to 1uf. 26 x 3 = 78 jobs total. Again
@:indeck is treated as a template, and the values are substituted in p
lace of
@@vth and @@cap in that file.

submit -p params sim.exe @:indeck

        In this case, parameter definitions are taken from the file na
med
params instead of the command line. The file might have the following
contents:

                # paramters for my job submission
                parameter @@vth=0:0.2:5
                parameter @@cap = 10pf,100pf,1uf

submit -p "params;@@num=1-10;@@color=blue" job.sh @:job.data

        For someone who loves syntax and complexity... The semicolon s
```

```
eparates
the parameters value into three parts. The first says to load parame
te
rs from
a file params. The next part says add an additional parameter @@num
 th
at goes
from 1 to 10. The last part says add an additional parameter @@colo
r w
ith a
single value blue. The parameters @@num and @@color cannot overrid
e an
ything
defined within params; they must be new parameter names.

submit -d input.csv sim.exe @:indeck

        Takes parameters from the data file input.csv, which must be
 i
n comma-
separated value format. The first line of this file may contain a
 seri
es of
@@param names for each of the columns. If it doesn't, then the col
umns
 are
assumed to be called @@1, @@2, @@3, etc. Each of the remaining lin
es
represents a set of parameter values for one job; if there are 100
 suc
h lines,
there will be 100 jobs. For example, the file input.csv might look
 lik
e this:

                @@vth, @@cap
                1.1, 1pf
                2.2, 1pf
                1.1, 10pf
                2.2, 10pf

        Parameters are substituted as before into template files
 such
as
@:indeck.

submit -d input.csv -p "@@doping=1e15-1e17 in 30 log" sim.exe @:infi
le

        Takes parameters from the data file input.csv, but also adds
 a
nother
parameter @@doping which goes from 1e15 to 1e17 in 30 points on a
 log
scale.
For each of these points, all values in the data file will be execu
ted
. If the
data file specifies 50 jobs, then this command would run 30 x 50 =
 150
0 jobs.
```

```
submit -d input.csv -i @:extra/data.txt sim.exe @:indeck

        In addition to the template indeck file, send along another fi
le
extra/data.txt with each job, and treat it as a template too.

submit -s / -p @@address=23 Main St.,Hometown,Indiana/42
Broadway,Hometown,Indiana -s , -p @@color=red,green,blue job.sh @:job.
data

        Change the separator to slash when defining the addresses, the
n change
back to comma for the @@color parameter and any remaining arguments. W
e
shouldn't have to change the separator often, but it might come in han
dy if
the value strings themselves have commas.

submit @@num=1:1000 sim.exe input@@num

        Submit jobs 1,2,3,...,1000. Parameter names such as @@num are
recognized not only in template files, but also for arguments on the c
ommand
line. In this case, the numbers 1,2,3,...,1000 are substituted into th
e file
name, so the various jobs take their input from "input1", "input2", ..
.,
"input1000".

submit @@file=glob:indeck* sim.exe @:file

        Look for files matching indeck* and use the list of names as t
he
parameter @@file. Those values could be substituted into other templat
e files,
or used on the command line as in this example. Suppose the directory
contains
files indeckA, indeckB, and indeck-123. This example would launch thre
e jobs
using each of those files as input for the job.
```

Additional information is available by requesting user specific lists of choices for some

command options. The available option lists are generated for a user based on configured restrictions and availability. The values listed here are for example only and may not be available on all HUBs.

```
$ submit --help tools

Currently available TOOLs are:
   pegasus-plan

$ submit --help venues

Currently available VENUES are:
   DiaGrid
   WF-DiaGrid

$ submit --help managers

Currently available MANAGERs are:
   mpi
   mpich
   parallel
```

By specifying a suitable set of command line parameters it is possible to execute commands on configured remote systems. The simple premise is that a typical command line can be prefaced by submit and its arguments to execute the command remotely.

```
$ submit -v clusterA echo Hello world!
Hello world!
```

In this example the echo command is executed on the venue named clusterA where runs are executed directly on the host. Execution of the same command on a cluster using PBS would be done in a similar fashion

```
$ submit -v clusterB echo Hello world!
(2586337) Simulation Queued Wed Oct  7 14:45:21 2009
(2586337) Simulation Done Wed Oct  7 14:54:36 2009
$ cat 00577296.stdout
Hello world!
```

submit supports an extensible variety of submission mechanisms. HUBzero supported submission mechanisms are

- local - use batch submission mechanisms available directly on the submit host. These include PBS, condor, and Pegasus batch queue submission.
- ssh - direct use of ssh. Submit manages access to a common ssh key, essentially serving as a proxy for the HUB user.
- ssh + remote batch submission - use ssh to do batch run submission remotely. Again methods for PBS, condor, and Pegasus batch queue submission are provided.

In addition to single site submission the -r/--redundancy option provides the option to simultaneously submit runs to multiple remote venues. In such cases the successful completion of a run at one venue cancels runs at all other venues. If none of the runs are successful results from one of the runs are returned to the user. Redundant submission is not allowed when performing parametric sweeps.

A site for remote execution is selected in one of the following ways, listed in order of precedence:

- Execute the command within the user tool session, -l/--local option
- User specified on the command line with -v/--venue option.
- Randomly selected from remote sites associated pre-staged application.
- Select randomly from all configured sites

Any files specified by the user plus internally generated scripts are packed into a tarball for delivery to the remote site. Individual files or entire directory trees may be listed as command inputs using the -i/--inputfile option. Additionally command arguments that exist as files or directories will be packed into the tarball. If using ssh based submission mechanisms the tarball is transferred using scp.

The job wrapper script is executed remotely either directly or submitted to a batch queue. The job is subject to all remote queuing restrictions and idiosyncrasies.

Remote batch jobs are monitored for progress. Methods appropriate to the batch queuing system are used to check job status at a configurable frequency. A typical frequency is on the order one minute. Job status changes are reported to the user. The maximum time between reports to the user is set on the order of five minutes even in the absence of change. The job status is used to detect job completion.

The same methods used to transfer input files are applied in reverse to retrieve output files. Any

files and directories created or modified by the application are be retrieved. A tarball is retrieved and expanded to the home base directory. It is up to the user to avoid the overwriting of files.

In addition to the application generated output files additional files are generated in the course of remote run execution. Some of these files are for internal bookkeeping and are consumed by submit, a few files however remain in the home base directory. The remaining files include RUNID.stdout and RUNID.stderr, it is also possible that a second set of standard output/error files will exist containing the output from the batch job submission script. RUNID represents unique job identifier assigned by submit.

Pegasus Workflow Submission

Overview

With this version of submit new functionality has been introduced to support workflow management using Pegasus. Two use cases are available: automatic workflow generation for parametric sweeps on one or more variables, or user constructed workflows. In both instances submit is used to configure access to one or more computational resources eliminating the need for a user to supply a site catalog thereby simplifying use of the workflow management system.

Parametric Sweeps

submit command options -p/--parameters and -d/--data have added to provide support for specifying parameter sweeps in a compact general way. The user is relieved of the chore of generating entire sets of input files and command arguments comprising a parameter sweep. Substitutable parameters are declared on the submit command line. Values of these parameters can then be systematically substituted into data files or application command line parameters. submit performs the necessary substitutions to cover all parameter combination. Each combination of parameters is abstractly represented as a node in a workflow and concretely executed as a job on the designated computational resource. A simple curses interface is provided to monitor progress of the simulation run.

User Constructed Workflows

Parameter sweeps are represented as a simple workflow consisting of many individual independent nodes. That is data is not shared between nodes or jobs in the run. There are cases where this simple approach is not sufficient to describe a workflow required to achieve a developer's or user's objective. Under these circumstances a developer may create a workflow and build an application around where the user supplies values for selected inputs. In such cases the Pegasus API's may be used to generate the abstract workflow description in the form of a dax file. The dax file can then executed by a simple submit command.

```
submit pegasus-plan --dax daxFile
```

In cases where more than one venue is capable of executing Pegasus runs a specific venue can be requested on the command line, otherwise submit will choose a venue at random.

```
submit -v DiaGrid pegasus-plan --dax daxFile
```

There are several additional options to pegasus-plan command that are supplied by submit. A few of the command options may be provided on the command line. submit reserves the option to silently ignore options as it sees fit.

In addition to remote execution of Pegasus runs it is also possible to do the execution locally with in the tool session. Simply use the submit -l/--local option.

```
submit --local pegasus-plan --dax daxFile
```

The use command can be employed to put pegasus-plan and all other Pegasus commands in the PATH environment variable. In additional to setting PATH, other environment variables are set allowing use of the python and java dax generation API's.

Rappture Integration with Submit

Overview

It is possible to use the submit command to execute simulation jobs generated by Rappture interfaces remotely. A common approach is to create a shell script which can exec'd or forked from an application wrapper script. This approach has been applied to TCL, Python, Perl wrapper scripts. To avoid consumption of large quantities of remote resources it is imperative that the submit command be terminated when directed to do so by the application user (Abort button).

TCL Wrapper Script

submit can be called from a TCL Rappture wrapper script for remote batch job submission. An example of what code to insert in the wrapper script is detailed here.

An initial code segment is required to catch the Abort button interrupt. Setting execctl to 1 will terminate the process and any child processes.

```
package require Rappture
Rappture::signal SIGHUP sHUP {
   puts "Caught SIGHUP"
   set execctl 1
}
Rappture::signal SIGTERM sTERM {
   puts "Caught SIGTERM"
   set execctl 1
}
```

A second code segment is used to build an executable script that can executed using Rappture::exec. The trap statement will catch the interrupt thrown when the wrapper script execution is Aborted. Putting the submit command in the background allows for the possibility of issuing multiple submit commands from the script. The wait statement forces the shell script to wait for all submit commands to terminate before exiting.

```
   set    submitScript "#!/bin/sh\\n\\n"
   append submitScript "trap cleanup HUP INT QUIT ABRT TERM\\n\\n"
   append submitScript "cleanup()\\n"
   append submitScript "{\\n"
   append submitScript "   kill -TERM `jobs -p`\\n"
   append submitScript "   exit 1\\n"
```

```
   append submitScript "}\\n\\n"

   append submitScript "cd [pwd]\\n"
   append submitScript "submit -v cluster -n $nodes -w $walltime\\\\\\
n"
   append submitScript "       COMMAND ARGUMENTS &\\n"
   append submitScript "sleep 5\\n"
   append submitScript "wait\\n"

   set submitScriptPath [file join [pwd] submit_script.sh]
   set fid [open $submitScriptPath w]
   puts $fid $submitScript
   close $fid
   file attributes $submitScriptPath -permissions 00755
```

The standard method for wrapper script execution of commands can now be used. This will stream the output from all submit commands contained in submit_script.sh to the GUI display. The same output will be retained in the variable out.

```
set status [catch {Rappture::exec $submitScriptPath} out]
```

Each submit command creates files to hold COMMAND standard output and standard error. The file names are of the form JOBID.stdout and JOBID.stderr, where JOBID is an 8 digit number. These results can be gathered as follows.

```
   set out2 ""
   foreach errfile [glob -nocomplain *.stderr] {
      if [file size $errfile] {
         if {[catch {open $errfile r} fid] == 0} {
            set info [read $fid]
            close $fid
            append out2 $info
         }
      }
      file delete -force $errfile
   }
   foreach outfile [glob -nocomplain *.stdout] {
      if [file size $outfile] {
         if {[catch {open $outfile r} fid] == 0} {
            set info [read $fid]
            close $fid
            append out2 $info
         }
      }
```

```
      file delete -force $outfile
   }
```

The script file should be removed.

```
file delete -force $submitScriptPath
```

The output is presented as the job output log.

```
$driver put output.log $out2
```

All other result processing can proceed as normal.

Python Wrapper Script

submit can be called from a python Rappture wrapper script for remote batch job submission. An example of what code to insert in the wrapper script is detailed here.

An initial code segment is required to import some predefined functions that manage typical aspects of remote submission. An important aspect is the handling of user interruption via the Abort button.

```
import os
import stat
import Rappture
from Rappture.tools import getCommandOutput as RapptureExec
```

A second code segment is used to build an executable script that can be executed using RapptureExec. The trap statement will catch the interrupt thrown when the wrapper script execution is Aborted. Putting the submit command in the background allows for the possibility of issuing multiple submit commands from the script. The wait statement forces the shell script to wait for all submit commands to terminate before exiting and returning control to the application wrapper script.

```
   submitScriptName = 'submit_app.sh'
   submitScript     = """#!/bin/sh

trap cleanup HUP INT QUIT ABRT TERM
```

```
cleanup()
{
   echo "Abnormal termination by signal"
   kill -s TERM `jobs -p`
}

"""
   submitScript += "cd %s\\\n" % (os.getcwd())
   submitScript += "submit -v %s -n %s -w %s \\\\\\n" % (venue,nodes,w
alltime)
   submitScript += "       %s %s &\\\n" % (COMMAND,ARGUMENTS)
   submitScript += "wait\\\n"

   submitScriptPath = os.path.join(os.getcwd(),submitScriptName)
   fp = open(submitScriptPath,'w')
   if fp:
      fp.write(submitScript)
      fp.close()

   os.chmod(submitScriptPath,
            stat.S_IRWXU|stat.S_IRGRP|stat.S_IXGRP|stat.S_IROTH|stat.S
_IXOTH)
```

The standard method for wrapper script execution of commands can now be used. This will stream the output from all submit commands contained in submit_script.sh to the GUI display. The same output will be retained in the variable stdOutput.

```
   exitStatus,stdOutput,stdError = RapptureExec([submitScriptPath])
```

Each submit command creates files to hold COMMAND standard output and standard error. The file names are of the form JOBID.stdout and JOBID.stderr, where JOBID is an 8 digit number. These results can be gathered as follows.

```
   reStdout = re.compile(".*\.stdout$")
   reStderr = re.compile(".*\.stderr$")

   out2 = ""
   errFiles = filter(reStderr.search,os.listdir(os.getpwd()))
   if errFiles != []:
      for errFile in errFiles:
         errFilePath = os.path.join(os.getpwd(),errFile)
         if os.path.getsize(errFilePath) > 0:
            f = open(errFilePath,'r')
            outFileLines = f.readlines()
```

```
            f.close()
            stderror = ''.join(outFileLines)
            out2 += '\n' + stderror
         os.remove(errFilePath)

   outFiles = filter(reStdout.search,os.listdir(os.getpwd()))
   if outFiles != []:
      for outFile in outFiles:
         outFilePath = os.path.join(os.getpwd(),outFile)
         if os.path.getsize(outFilePath) > 0:
            f = open(outFilePath,'r')
            outFileLines = f.readlines()
            f.close()
            stdoutput = ''.join(outFileLines)
            out2 += '\n' + stdoutput
         os.remove(outFilePath)
```

The script file should be removed.

```
   os.remove(submitScriptPath)
```

The output is presented as the job output log.

```
   lib.put("output.log", out2, append=1)
```

All other result processing can proceed as normal.

Perl Wrapper

submit can be called from a perl Rappture wrapper script for remote batch job submission. An example of what code to insert in the wrapper script is detailed here.

An initial code segment is required to catch the Abort button interrupt.

```
use Rappture

my $ChildPID = 0;

sub trapSig {
    print "Signal @_ trapped\\n";
    if($ChildPID != 0) {
        kill 'TERM', $ChildPID;
```

```
        exit 1;
    }
}
$SIG{TERM} = \&trapSig;
$SIG{HUP}  = \&trapSig;
$SIG{INT}  = \&trapSig;
```

A second code segment is used to build an executable script that can executed using Rappture.tools.getCommandOutput. The trap statement will catch the interrupt thrown when the wrapper script execution is Aborted. The wait statement forces the shell script to wait for the submit command to terminate before exiting.

```
$SCRPT = "submit_app.sh";
open(FID,">$SCRPT");
print FID "#!/bin/sh\\n";
print FID "\\n";
print FID "trap cleanup HUP INT QUIT ABRT TERM\\n\\n";
print FID "cleanup()\\n";
print FID "{\\n";
print FID "   kill -s TERM `jobs -p`\\n";
print FID "   exit 1\\n";
print FID "}\\n\\n";

print FID "submit -v cluster -n $nPROCS -w $wallTime COMMAND ARGUMENTS
 &\\n";
print FID "wait %1\\n";
print FID "exitStatus=\\$?\\n";
print FID "exit \\$exitStatus\\n";
close(FID);
chmod 0775, $SCRPT;
```

The standard fork and exec method for wrapper script execution of commands can now be used. Using this approach does not allow streaming of the command outputs.

```
if      (!defined($ChildPID = fork())) {
    die "cannot fork: $!";
} elsif ($ChildPID == 0) {
    exec("./$SCRPT") or die "cannot exec $SCRPT: $!";
    exit(0);
} else {
    waitpid($ChildPID,0);
}
```

Each submit command creates files to hold COMMAND standard output and standard error. The file names are of the form JOBID.stdout and JOBID.stderr, where JOBID is an 8 digit number. These results can be gathered with standard perl commands for file matching, reading, etc. All other result processing can proceed as normal.