Using modules with MPICH-G2 (and "loose ends")

Johnny Chang

johnny@nas.nasa.gov

July 9, 2001

Last Modified: October 19, 2001

Table of Contents

Abstract

A new approach to running complex, distributed MPI jobs using the MPICH-G2 library is described. This approach allows the user to switch between different versions of compilers, system libraries, MPI libraries, etc. via the "module" command. The key idea is a departure from the prescribed "(jobtype=mpi)" approach to running distributed MPI jobs. The new method requires the user to provide a script that will be run as the "executable" with the "(jobtype=single)" RSL attribute. The major advantage of the proposed method is to enable users to decide in their own script what modules, environment, etc. they would like to have in running their job.

Prerequisites

This document is intended for application developers and users who want to run complex, distributed MPI jobs across two or more machines. It assumes the reader is familiar with Unix and the basics of Globus as expressed in the Globus Quick Start Guide.

Background

In mid-April 2001, Nick Karonis (karonis@olympus.cs.niu.edu) discovered that a bug in SGI's

implementation of mpi causes some codes to hang. This particular bug was fixed in newer releases of the mpt module (mpt.1.4.0.2 and higher). The question then arose as to how one would go about using modules with Globus/MPICH-G2 (http://www.hpclab.niu.edu/mpi/). One solution, implemented by Judith Utley (utley@marcy.nas.nasa.gov), is to hard-wire the newer mpt module into all MPICH-G2 jobs. Better solutions that do not rely on a particular hard-wiring of modules are being contemplated.

This is an important problem because the ability for a user to switch between modules is crucial for a wide variety of reasons. For example, some codes run on older modules but not on newer ones or vice versa. As modules on a system are updated, users may need to switch between modules to assess the impact, if any, of the change. Users may need to switch between modules to determine why a code that used to run a year ago, now behaves differently. These are just a few of the many reasons why various versions of modules are available on the system at any given time. This paper describes a solution that any IPG user could use right away that would not rely on any future "fix" to the Globus middleware or tools or services derived thereof.

This step-by-step description is incremental in nature and touches upon a number of techniques that I've found useful while learning to use the IPG. They form the "loose ends" that I've chosen to include in this document. Readers can skip to the solution for using modules with MPICH-G2 by clicking here.

Introduction

When a user logs into a machine at NAS (NASA Advanced Supercomputing division), a rich environment (paths, aliases, environment variables, etc.) is already pre-defined to provide easy accessibility to Unix commands. This facility is replicated in jobs submitted to PBS. That is, batch jobs enjoy much of the same computing environment (and much more) as interactive jobs (run at the command prompt). Jobs submitted to Globus, on the other hand, have almost a non-existent environment. Even a simple 'ls' command to list files requires some knowledge of where (which directory) the command resides or some "trick" to provide an instantaneous environment to process the command. This was a design issue and some email discussion along this issue is attached in Appendix B.2 . Globus provides no mechanism for using modules currently. Users who want to use modules will have to do so in their own scripts.

Rich Environment

The rich environment that users have become accustomed to is completely due to four files that are executed (source'd) when a user logs in, or when a PBS batch job is run, and they are (for the C shell user):

/etc/cshrc
$HOME/.cshrc
/usr/local/lib/init/cshrc.global
$HOME/.login

The cshrc.global file is source'd from the user's $HOME/.cshrc file unless they have explicitly commented this out. Similar files exist for users of other shells.

The "module" command, which allows users to load/switch modules is (for me)

```
evelyn:/u/johnny> which module
module:          aliased to /usr/bsd/logger -i -p local4.notice "module !*" ;
eval `/opt/modules/modules/bin/modulecmd tcsh !*`
```

and this alias is defined in /opt/modules/modules/init/csh (or /opt/modules/modules/init/tcsh) which is source'd from both /etc/cshrc *and* /usr/local/lib/init/cshrc.global (again, similar files exist for users of other shells).

This is the **first piece** of the puzzle. Namely, if the 'module' command is not enabled, the user will need to explicitly have the line

```
source /opt/modules/modules/init/csh
```

in their script before using modules. It doesn't hurt to have this line in the script even if the module command is already enabled. This method of enabling the module command is universal across all the SGI Origins and Cray computers in the NASA IPG .

To see the current setup with regards to modules for Globus jobs, consider the following script called 'module.csh':

```
-------------------------------------------------
#! /bin/csh
set verbose
sleep 100      ! sleep for 100 seconds
module list
which mpirun
-------------------------------------------------
```

The first line starts up a C shell and sources the $HOME/.cshrc file. The second line (set verbose) causes all subsequent commands that are run to be echo'd to stderr. The third line (sleep 100) is one that I use a lot when I want to see what PBS script was created by the Globus-to-PBS interface. More on this later. The fourth line lists what modules, if any, are loaded for this job (and it will also serve as a test for whether the 'module' command works or not).

Experiments with module.csh

First, a Globus job submitted to evelyn.nas.nasa.gov's jobmanager-fork:

```
-------------------------------------------------
evelyn:/u/johnny> globusrun -s -r evelyn '&(executable=/u/johnny/module.csh)'
sleep 100      ! sleep for 100 seconds
mpirun not in /usr/nas/bin /usr/bin /usr/sbin /usr/bin/X11 /usr/local/pkg/pgi/sgi/bi
/usr/local/pkg/pgi/bin /usr/prg/bin /usr/bsd /usr/local/pbs/bin /usr/local/pbs/sbin
/usr/local/bin /usr/java/bin /usr/etc /usr/prg/pkg/globus/1.1.3/tools/mips-sgi-irix6
/etc .
module list
No Modulefiles Currently Loaded.
setenv _MODULESBEGINENV_ /u/johnny/.modulesbeginenv ;
which mpirun
-------------------------------------------------
```

Result: No modules loaded and the mpirun command is not found in my path. Interestingly, the

'module' command is enabled even though no modules are loaded. This is because /etc/cshrc was not run and so no modules are loaded, but /opt/modules/modules/init/csh was source'd from /usr/local/lib/init/cshrc.global. (Note: stdout and stderr output are mixed when returned to the screen.)

Second, a Globus job to hopper.nas.nasa.gov's jobmanager-pbs:

```
-------------------------------------------------
evelyn:/u/johnny> globusrun -s -r hopper '&(executable=/lc/johnny/module.csh)'
sleep 100      ! sleep for 100 seconds
module list
Currently Loaded Modulefiles:
          1) modules    2) MIPSpro    3) mpt        4) scsl

which mpirun

<*motd snipped*>

[1] 1091589       ! executable run in background, see PBS script below.
/opt/mpt/mpt/usr/bin/mpirun
[1]    Done                  /lc/johnny/module.csh < /dev/null
logout

<*PBS Job resource summary snipped*>
-------------------------------------------------
```

Result: The default modules are loaded (PBS jobs automatically execute /etc/cshrc at start-up) and the mpirun from the *default* mpt module is accessed from the script.

I've often found it useful to look at the PBS script that is generated by the Globus-to-PBS interface /globus/deploy/libexec/globus-script-pbs-submit. One can view the PBS script when the PBS job id is known. This is the reason I insert sleep commands in my script -- to give me enough time to execute the following two commands:

```
-------------------------------------------------
turing:/cluster/hopper/PBS/mom_priv/jobs> qstat -au johnny

fermi.nas.nasa.gov: NAS Origin 2000 Cluster Frontend

        Tue Jul 10 17:26:55 2001

Server reports 1 job total (R:1 Q:0 H:0 W:0 T:0 E:0)
  hopper: 1/30 nodes used, 58 CPU/14210mb free, load 56.18 (R:1 T:0 E:0)

                                                          Req'd      Req'd   Elap
Job ID       Username Queue    Jobname    SessID  TSK Memory  Nds wallt S wallt
------------ -------- -------- ---------- ------- --- ------- --- ----- - -----
26023.fermi  johnny   submit   STDIN       810567   2   490mb   1 00:05 R 00:00

-------------------------------------------------
```

The PBS job id is 26023 and the status of the job is Running (2nd to last column). Then, in the directory shown on the command prompt, one can view one's own PBS scripts once the PBS job has started running (note: 'ls' will not execute here since the directory's permission is 751)

```
-------------------------------------------------
turing:/cluster/hopper/PBS/mom_priv/jobs> cat 26023.fermi.SC
```

```
# PBS batch job script built by Globus job manager

#PBS -o /u/johnny/.globus/.gass_cache/globus_gass_cache_994811212
#PBS -e /u/johnny/.globus/.gass_cache/globus_gass_cache_994811213
#PBS -l ncpus=1
#PBS -v GLOBUS_GRAM_MYJOB_CONTACT=URLx-nexus://hopper.nas.nasa.gov:24803/,    \
X509_CERT_DIR=/usr/prg/pkg/globus/1.1.3/.deploy/share/certificates,           \
GLOBUS_GRAM_JOB_CONTACT=https://hopper.nas.nasa.gov:24802/810796/994811209/, \
GLOBUS_DEPLOY_PATH=/usr/prg/pkg/globus/1.1.3/.deploy,                         \
GLOBUS_INSTALL_PATH=/usr/prg/pkg/globus/1.1.3,                                \
X509_USER_PROXY=/u/johnny/.globus/.gass_cache/globus_gass_cache_994811211,

# Changing to directory as requested by user

cd /u/johnny

# Executing job as requested by user

/lc/johnny/module.csh  < /dev/null &
wait

------------------------------------------------
```

More interestingly, when the (jobtype=mpi) parameter is added to the RSL, one gets a PBS script that contains:

```
------------------------------------------------
turing:/cluster/hopper/PBS/mom_priv/jobs> cat 26282.fermi.SC
# PBS batch job script built by Globus job manager

#PBS -o /u/johnny/.globus/.gass_cache/globus_gass_cache_994885658
#PBS -e /u/johnny/.globus/.gass_cache/globus_gass_cache_994885659
#PBS -l ncpus=1
#PBS -v GLOBUS_GRAM_MYJOB_CONTACT=URLx-nexus://hopper.nas.nasa.gov:41340/,    \
X509_CERT_DIR=/usr/prg/pkg/globus/1.1.3/.deploy/share/certificates,           \
GLOBUS_GRAM_JOB_CONTACT=https://hopper.nas.nasa.gov:41339/1106709/994885655/,\
GLOBUS_DEPLOY_PATH=/usr/prg/pkg/globus/1.1.3/.deploy,                         \
GLOBUS_INSTALL_PATH=/usr/prg/pkg/globus/1.1.3,                                \
X509_USER_PROXY=/u/johnny/.globus/.gass_cache/globus_gass_cache_994885657,

# Changing to directory as requested by user

cd /u/johnny

# Executing job as requested by user

module load mpt.new
module swap mpt mpt.new
/opt/mpt/mpt.new/usr/bin/mpirun -np 1 /lc/johnny/module.csh  < /dev/null

------------------------------------------------
```

The last line of the PBS script is the *wrong* way to run a user script (module.csh), but this example shows three points:

1. the addition of (jobtype=mpi) into the RSL triggers a hard-wired load and swap to the mpt.new module as implemented by Judith Utley,

2. the 'executable' module.csh is run with a hard-wired mpirun command which is the wrong usage

for running user scripts, and

3. the actual version of mpirun that is being used in the module.csh script is /usr/bin/mpirun (which differs from the mpirun in both the mpt and the mpt.new modules). This last point can be seen by running the experiment and looking for the output to the 'which mpirun' command (left as an exercise to the reader). The reason for this is somewhat obscure, but it has to do with the fact that SGI's mpirun command calls the array services library, which, in turn, clobbers the user's PATH environment and replaces it with: "/usr/sbin:/usr/bsd:/sbin:/usr/bin:/usr/bin/X11:". The processes started by mpirun inherit this path, and within the module.csh script, the only mpirun that is found by the 'which mpirun' command is the one under /usr/bin.

These three experiments show considerable variability in which (and whether or not) modules are loaded, and which (if any) version of mpirun is accessed depending on how the Globus job is run. **The major advantage of the method proposed in this paper is to remove this variability by having the user decide in their own script what modules, environment, etc. they would like to have in running their job.** Once this shell script is written, and the intention is to have it run as the executable in a Globus job, the only appropriate jobtype to use in the RSL is 'single'. This is the **second piece** of the puzzle.

(jobtype=single)

Turning now towards running distributed MPI jobs on 2 (or more machines) with MPICH-G2, the main question is whether or not message passing between different machines using MPICH-G2 is possible with (jobtype=single). From a historical perspective, Globus jobs using MPICH-G did not specify a 'jobtype' parameter and thus defaulted to (jobtype=multiple). With MPICH-G2, the RSL created by the mpirun script (link) from the appropriate MPICH-G2 directory used a (jobtype=mpi) RSL parameter.

The different 'jobtypes' result in different ways that the MPI job is run. With (jobtype=multiple) the MPI job is run as (assuming no stdin):

```
path_to_executable/mpi_executable  < /dev/null &
path_to_executable/mpi_executable  < /dev/null &
path_to_executable/mpi_executable  < /dev/null &
    ...
wait
```

The number of occurrences of mpi_executable (in the batch script) is determined by the 'count' parameter in the RSL. When the MPI job is run across multiple hosts, a similar repeating pattern of the mpi_executable appears in the batch script for each subjob (the number of repetitions is determined by the 'count' parameter in each subjob).

With (jobtype=mpi) the MPI job is run as (again, assuming no stdin):

```
module load mpt.new
module swap mpt mpt.new
/opt/mpt/mpt.new/usr/bin/mpirun -np count_parameter path_to_executable/
mpi_executable  < /dev/null
```

That is, the NAS hard-wired path to the vendor's new mpirun is used to run mpi_executable.

With the proposed (jobtype=single) way of running MPICH-G2 jobs via a user script, the job is run as (assuming no stdin):

```
path_to_user_script/user_script < /dev/null
```

This is fine for an MPI job run out of a single user script on a single host. The question then is how one runs a distributed MPI job across several hosts. The answer can be found by looking at the RSL generated by the mpirun script in either the MPICH-G or MPICH-G2 directories. Reproduced here is an example taken from my Globus user tutorial.

```
evelyn% cat hello_duroc.rsl
+
( &(resourceManagerContact="evelyn.nas.nasa.gov")
   (count=4)
   (jobtype=mpi)
   (label="subjob 0")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0))
   (directory="/u/johnny/duroc")
   (executable="/u/johnny/duroc/hello_mpichg")
)
( &(resourceManagerContact="turing.nas.nasa.gov")
   (count=6)
   (jobtype=mpi)
   (label="subjob 4")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1))
   (directory="/u/johnny/duroc")
   (executable="/u/johnny/duroc/hello_mpichg")
)
```

This RSL is used to run a simple "Hello World" MPI program across evelyn and turing, using 4 processes on evelyn and 6 processes on turing. The only differences between the MPICH-G and MPICH-G2 RSLs are the greatly shortened resourceManagerContact string in the latter version and the previously alluded to (jobtype=mpi) RSL parameter. The key ingredient (link) for the coordination and communication across different hosts is the GLOBUS_DUROC_SUBJOB_INDEX environment. The label parameter is superfluous (but may be useful in interpreting error messages which refer to subjobs by their label). This key ingredient is the **third and final piece** of the puzzle.

The above RSL will run an MPI job with a total number of 10 processes (not counting the shepherd processes). The four processes associated with GLOBUS_DUROC_SUBJOB_INDEX 0 will run with ranks 0 through 3, and the six processes associated with GLOBUS_DUROC_SUBJOB_INDEX 1 will run with ranks 4 through 9. The order of the subjobs described by each GRAM type RSL (&(.....)) is unimportant, but the indices associated with the GLOBUS_DUROC_SUBJOB_INDEX environment must run from 0 through the number of subjobs minus one.

In the section below, we look at several examples of running MPI jobs under MPICH-G2 with (jobtype=single). It must be stressed that this is not the prescribed method for running MPICH-G2 jobs, therefore, several issues regarding correctness, performance, and limitations will also be addressed.

EXAMPLES

Example 1: Hello World MPI program.

For pedagogical reasons, I've included here all the necessary parts for running my MPI version of the "Hello World" program using scripts.

```
------------------------------------------------------------------
evelyn:/u/johnny/duroc/mpich-g2> cat hello_mpi.f
      program hello_mpi
! A basic "Hello World" MPI program intended to demonstrate how to
! execute an MPI program under Globus on the NAS IPG
      include "mpif.h"
      integer date_time(8)
      character(len=10) big_ben(3), hostname
      call MPI_INIT(ierr)
      call date_and_time(big_ben(1), big_ben(2), big_ben(3), date_time)
      call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
      call gethostname(hostname)
      print *,'Process #', myid, 'of', numprocs, 'at time: ',
     & big_ben(1), big_ben(2),' on host: ',trim(adjustl(hostname))
      call MPI_FINALIZE(ierr)
      end
```

The gethostname routine is a Fortran-to-C interface that uses the well-known C function call by the same name:

```
evelyn:/u/johnny/duroc/mpich-g2> cat ftoc.c
void gethostname_(char *hostname)
{
      #include
      gethostname(hostname, 10);
      return;
}
------------------------------------------------------------------
```

The compilation and linking using the MPICH-G2-provided compiler scripts (which I've aliased via environment variable settings):

```
evelyn:/u/johnny/duroc/mpich-g2> echo $MPICC
/globus/mpich-n32/bin/mpicc
evelyn:/u/johnny/duroc/mpich-g2> echo $MPIF90
/globus/mpich-n32/bin/mpif90
```

is done via:

```
evelyn:/u/johnny/duroc/mpich-g2> $MPICC -c ftoc.c
evelyn:/u/johnny/duroc/mpich-g2> $MPIF90 -o hello_mpichg2 hello_mpi.f ftoc.o
```

The user script that enables the module command and loads the mpt module is:

```
evelyn:/u/johnny/duroc/mpich-g2> cat hello_mpichg2.scr
#! /bin/csh
source /opt/modules/modules/init/csh
module load mpt
mpirun -np $NP ./hello_mpichg2
```

It takes an environment variable (which I've called NP) that must be set in the RSL:

```
evelyn:/u/johnny/duroc/mpich-g2> cat hello_script.rsl
```

```
+
( &(resourceManagerContact="evelyn.nas.nasa.gov")
   (rsl_substitution = (nprocs "4") )
   (count=$(nprocs))
   (jobtype=single)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0) (NP $(nprocs)) )
   (directory="/u/johnny/duroc/mpich-g2")
   (executable="/u/johnny/duroc/mpich-g2/hello_mpichg2.scr")
)
( &(resourceManagerContact="turing.nas.nasa.gov")
   (rsl_substitution = (nprocs "6") )
   (count=$(nprocs))
   (jobtype=single)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1) (NP $(nprocs)) )
   (directory="/u/johnny/duroc/mpich-g2")
   (executable="/u/johnny/duroc/mpich-g2/hello_mpichg2.scr")
)
```

This RSL assumes that I have already setup the appropriate directory structure on evelyn and turing, and that I have the appropriate scripts (made executable) and MPI executables in the correct locations on the two machines.

The Globus job is launched via:

```
evelyn:/u/johnny/duroc/mpich-g2> globusrun -s -f hello_script.rsl
```

with the result:

```
Job Limits not enabled: Job not found or not part of job
Job Limits not enabled: Job not found or not part of job
 Process # 4 of 10 at time: 20010709  145346.692 on host: turing
 Process # 5 of 10 at time: 20010709  145346.699 on host: turing
 Process # 6 of 10 at time: 20010709  145346.692 on host: turing
 Process # 8 of 10 at time: 20010709  145346.692 on host: turing
 Process # 7 of 10 at time: 20010709  145346.692 on host: turing
 Process # 9 of 10 at time: 20010709  145346.692 on host: turing
 Process # 0 of 10 at time: 20010709  145346.660 on host: evelyn
 Process # 1 of 10 at time: 20010709  145346.660 on host: evelyn
 Process # 2 of 10 at time: 20010709  145346.660 on host: evelyn
 Process # 3 of 10 at time: 20010709  145346.660 on host: evelyn
```

(Since the upgrade of the OS to IRIX 6.5.10f, there have been some error messages that presage the output, but they are innocuous.)

The output shows the correct number of processes and rank running on evelyn and turing. The time stamp is in the form YYYYMMDD HHMMSS.fractional_seconds. This example shows that the two subjobs are synchronized to start at the same time (modulo a time zone change) on the MPI_INIT call. I have run this example many times, and occasionally, have seen an approximately 5 minute delay between the time stamps on the two hosts. This is *not* due to the clocks on the two hosts going out of sync, but appears to arise from some underlying communication layer which I do not yet understand. It is unrelated to the (jobtype=single) RSL parameter since the same problem arises with (jobtype=mpi).

Example 2: ring example

This example uses the ring.c code from the MPICH-G2 website (http://www.niu.edu/mpi/)(link) The

new wrinkle is that the executable and scripts all reside on one machine, evelyn, and the goal is to run this MPI job across 3 machines: evelyn, turing, and rogallo. The user script (ring.scr), executable (ring), and RSL (ring_script.rsl) all reside under evelyn:/u/johnny/duroc/mpich-g2. The script (ring.scr) can be staged (or transferred) to turing and rogallo via the $(GLOBUS_GASS_URL)# prefix. The staged script will reside in the .globus/.gass_cache directories on turing and rogallo for the duration of the job and will be be deleted automatically at the end of the job.

The RSL is:

```
evelyn:/u/johnny/duroc/mpich-g2> cat ring_script.rsl
+
( &(resourceManagerContact="evelyn.nas.nasa.gov")
   (rsl_substitution=(nprocs "5"))
   (count=$(nprocs))
   (jobtype=single)
   (directory=$(HOME)/duroc/mpich-g2)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0) (NP $(nprocs)) )
   (executable=$(HOME)/duroc/mpich-g2/ring.scr)
)
( &(resourceManagerContact="turing.nas.nasa.gov")
   (rsl_substitution=(nprocs "4"))
   (count=$(nprocs))
   (jobtype=single)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1) (NP $(nprocs)) )
   (executable=$(GLOBUSRUN_GASS_URL)#$(HOME)/duroc/mpich-g2/ring.scr)
)
( &(resourceManagerContact="rogallo.larc.nasa.gov")
   (rsl_substitution=(nprocs "3"))
   (count=$(nprocs))
   (jobtype=single)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 2) (NP $(nprocs)) )
   (executable=$(GLOBUSRUN_GASS_URL)#$(HOME)/duroc/mpich-g2/ring.scr)
)
```

Notice that for the subjob to be run on evelyn there is a directory change to $HOME/duroc/mpich-g2, where my ring executable resides. For the subjobs to be run on turing and rogallo, the ring.scr script will do a remote file transfer of the ring executable and run from the defaulted $HOME directories.

The ring.scr script is:

```
evelyn:/u/johnny/duroc/mpich-g2> cat ring.scr
#! /bin/csh
source /opt/modules/modules/init/csh
module load mpt
if (`hostname` != "evelyn") then
   scp evelyn.nas.nasa.gov:duroc/mpich-g2/ring .
endif
mpirun -np $NP ./ring
```

In this example, the line "source /opt/modules/modules/init/csh" is crucial to enable the module command. It is not automatically enabled for users accessing rogallo at the current time, and so sourcing $HOME/.cshrc (via #! /bin/csh) is not sufficient for the module command, but is sufficient for all the other commands in the script. Another important, albeit subtle, detail is the use of 'scp' in the file transfer. On rogallo, the 'scp' command is the GSI enabled version of 'scp', which means that it does not require a password when a valid full proxy exists. When the subjob on rogallo starts, it receives a

full proxy from evelyn which remains valid for the duration of the job. On turing, the 'scp' command is not GSI enabled, and uses the rhosts or /etc/hosts.equiv with RSA host authentication method to authenticate. Since turing-ec.nas.nasa.gov is in evelyn's /etc/hosts.equiv file, the scp will also not require a password for file transfer. Lastly, one could have added (to ring.scr) the deletion of the 'ring' executable at the end of the job when 'hostname' != "evelyn". I have chosen to leave the 'ring' executable behind to give one a warm and fuzzy feeling that everything is working as expected.

Execution of this job appears as:

```
evelyn:/u/johnny/duroc/mpich-g2> globusrun -s -f ring_script.rsl
Job Limits not enabled: Job not found or not part of job
Job Limits not enabled: Job not found or not part of job
Master: end of trip 1 of 1: after receiving passed_num=12 (should be =trip*numpr
ocs=12) from source=11
```

The passed_num=12 corresponds to the sum of 5, 4, and 3 processes run on evelyn, turing, and rogallo, respectively.

Example 3: Nick Karonis' root_of_problem.c and bad.c

As noted in the beginning of this document, Nick Karonis discovered a bug in SGI's implementation of MPI that caused some codes to hang. The link to that email also leads to the two codes that he provided. The code "bad.c" reproduces the hang when run using MPICH-G2 with settings procA = 1, procB = 2, but works with SGI's MPI independent of procA and procB settings. This assertion cannot be verified now by running a Globus job with (jobtype=mpi) on the NAS machines because the mpt.new module, which fixes the hang, has been hard-wired into (jobtype=mpi) jobs. However, with (jobtype=single) and running a script as the executable, one can freely switch between modules and verify the assertion. One such script is given below.

The "root_of_problem.c" code differs from "bad.c" only in an MPI_Comm_dup function call (and another printf statement), and mimics how MPICH-G2 implements the native MPI_Intercomm_create function call in the "bad.c" code. MPICH-G2 implements some MPI functions by calling one or more vendor-supplied MPI functions. This code will hang when run with settings procA = 1, procB = 2 and using SGI's mpt.1.4.0.1 or *some* earlier modules (code fails/hangs with versions 1.4.0.1, 1.4.0.0, and 1.3.0.4, passes/runs with versions 1.3.0.0, 1.2.1.2, and 1.4.0.3).

As an aside, it is worth mentioning that the mpt module is used in three different phases. (1) During compilation/linking, the MPI library is used to resolve all the MPI function calls, (2) When launching the MPI job with SGI's mpirun, the version of the mpirun command invoked depends on which mpt module is loaded and, therefore, what is in the user path, and (3) During runtime, the version of the MPI library accessed depends on which mpt module is loaded. SGI uses dynamic libraries which are accessed during runtime as opposed to being statically compiled into the executable. It is only in the third phase that the version of the loaded mpt module matters for creating the hang. But for practical purposes it is best to be consistent in using the same modules for all three phases.

In this example, we will run both codes across evelyn and turing, with three processes on evelyn and two on turing. It turns out that the processes with ranks 0, 1, and 2 all need to be on the same host to reproduce the hang. The user script (hang.scr) is:

```
-----------------------------------------------------------------
#! /bin/csh
source /opt/modules/modules/init/csh
module load mpt.new

if (`hostname` != "evelyn") then
   scp evelyn.nas.nasa.gov:duroc/mpich-g2/bad .
   scp evelyn.nas.nasa.gov:duroc/mpich-g2/root_of_problem .
endif

mpirun -prefix "%@ " -np $NP ./bad
#mpirun -prefix "%@ " -np $NP ./root_of_problem

if (`hostname` != "evelyn") then
  rm -f bad root_of_problem
endif
-----------------------------------------------------------------
```

To obtain the cases that hang, one loads mpt instead of mpt.new in the script.

WARNING: If you try running the cases that hang under MPICH-G2, remember to clean up the stray processes after experimentation. The stray processes will continue to consume resources and rack up CPU time.

Note that I have commented out one of the mpirun commands. The current setup in MPICH-G2 does *not* allow running two MPICH-G2-compiled executables out of the same script (in a serial fashion as in the script above). Attempts to run a second MPI program out of the same script will encounter error messages of the type:

```
globus_duroc_barrier: aborting job!
globus_duroc_barrier: reason: our checkin was invalid!
```

It must be emphasized that this is a limitation inherent in MPICH-G2 and not in using (jobtype=single). The prescribed (jobtype=mpi) method of running MPI jobs under MPICH-G2 allows running only one MPICH-G2 job in a single Globus job submission. This limitation presents a problem for a class of problems that require running distributed, complex MPI jobs involving pre- and/or post-processing, all of which might entail running two or more MPI jobs out of the same script. The key to the solution is to realize that the limitation is present only in MPICH-G2 compiled executables, but not with native MPI. To run the afore-mentioned "complex" problem, one would build the pre- and/or post-processing MPI executables with the native MPI library, and the running of these parts of the script will be done on only one host (or more than one host as long as it is not distributed across hosts in the MPICH-G2 sense). Presumably, the computation in the pre- and/or post-processing parts of the job are less time-consuming and do not need to be run as a single code distributed across two or more hosts. More discussion on this point will be presented in Example 4.

The last point to note about the user script above is the option given to mpirun which enables the output from the different processes to be prefixed with a hostname. With a user script, one is free to make this choice, which would otherwise be lacking with (jobtype=mpi).

The RSL used for this example is:

```
-----------------------------------------------------------------
+
```

```
( &(resourceManagerContact="evelyn.nas.nasa.gov")
   (rsl_substitution=(nprocs "3"))
   (count=$(nprocs))
   (jobtype=single)
   (directory=$(HOME)/duroc/mpich-g2)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0) (NP $(nprocs)) )
   (executable=$(HOME)/duroc/mpich-g2/hang.scr)
)
( &(resourceManagerContact="turing.nas.nasa.gov")
   (rsl_substitution=(nprocs "2"))
   (count=$(nprocs))
   (jobtype=single)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1) (NP $(nprocs)) )
   (executable=$(GLOBUSRUN_GASS_URL)#$(HOME)/duroc/mpich-g2/hang.scr)
)
-----------------------------------------------------------------
```

Example 4: "Real" code example

All three of the previous examples could have been run using the conventional MPICH-G2 approach with (jobtype=mpi). In this example, we consider some issues that "real" MPI applications face which cannot be run satisfactorily in the conventional (jobtype=mpi) approach.

Issues:

1. Need to allocate an extra processor for the shepherd process to avoid performance problems. Whenever an MPI job with NP processes are run via "mpirun -np NP ...", there are actually NP+1 instances of the executable running. The extra "shepherd" process consumes very little CPU time, but is capable of destroying any load balancing built into the code. At a minimum, this implies that setting NP to the value in the (count=##) RSL parameter may be inadequate. Additionally, on some time-sharing machines such as the Crays, the batch job daemons that monitor job resource usage might kill the job if they catch NP+1 processes running when only NP were requested/allowed.

2. Even if the NP parameter in mpirun is set to count - 1, this would still be inadequate for hybrid MPI+OpenMP codes. These codes take advantage of any multi-level parallelism present in the algorithm by using MPI for the (outer) course-grained parallelism and OpenMP for the (inner) fine-grained parallelism. In this case, the value of NP will need to be much less than the "count" RSL parameter which specifies the processor count resource.

3. "Real" MPI applications require input/data files. There are many ways that the filenames of these input/data files become "associated" with those required for the application. For example,

   ```
   ln -s input_case47.dat fort.10
   ```

   creates a symlink between a particular input/data file with an expected target. This could also be accomplished with the 'assign' command which might be adorned with other 'assign' attributes for controlling data conversion during I/O. The input/data filenames could be renamed just prior to launching mpirun. Certainly, one could "prepare" all the filename associations (via ln, assign, or mv commands) before launching the MPICH-G2 job, but that would be very restrictive, especially for projects where many cases need to be run.

4. "Real" MPI applications involve substantial pre- and/or post-processing around the mpirun command. These go beyond making sure that the requisite input/data files are in the right place (Issue 3). Again, these steps could be segregated away from launching the MPICH-G2 job, but this would be a major design flaw. The directory where the execution takes place could be volatile and exist only for the duration of the MPICH-G2 job.

5. The default set of modules (MIPSpro, mpt, and scsl) may be inadequate for certain applications at any given time. In Example 3 above, the problem attributed to the mpt module only manifested itself in some versions of the mpt module.

These are just a few of the issues that could be resolved by using the method proposed in this document.

One of the main advantages of using MPICH-G2, which has heretofore not been mentioned except for a hint in the previous example, is the ability to "tie" two or more applications together by passing data between the applications via MPI. The applications in the two subjobs do not have to be the same. In fact, in the more general case using scripts and (jobtype=single), the work done in the different subjob scripts could be completely different albeit related through the requisite passing of information. In fact, even this last coupling is unnecessary except for a reason to couple and use MPICH-G2. The key to the coupling is that codes compiled with the MPICH-G2 library will be synchronized at the MPI_INIT function call, and the processor ranks are determined by the processor counts passed to mpirun in conjunction with the GLOBUS_DUROC_SUBJOB_INDEX environment. If the script in one subjob begins to run before the other, it will run up to the code compiled with the MPICH-G2 library and then stall and spin cycle at the MPI_INIT funtion call waiting for the other subjob to reach its corresponding MPI_INIT function call. If the two subjobs are run on separate batch systems, some care in choosing the maxWallTime resource must be exercised to allow for asynchronous job start times.

Example 5: NAS Parallel Benchmarks (NPB2.3)

In the previous examples, we looked at the issues of correctness and limitations accompanying the use of (jobtype=single) to run MPICH-G2 jobs. In this example, we look at the performance issue. That is, whether jobs run slower when the jobtype parameter is switched from mpi to single. For this, we examine the performance of some well-known NAS Parallel Benchmarks (NPB) run under different scenarios.

The three NPB's chosen for this study are:

- bt: linear equations for implicit scheme in Navier-Stokes equation
- lu: LU decomposition for Navier-Stokes equation
- sp: linear equations for Navier-Stokes equation

They can each be compiled and run with different 'Classes'. Class A is the smallest case and class B mimics a medium size problem. The number of processes required to run each benchmark is also built in at compilation time. Thus, lu.A.4 corresponds to the class A version of the lu benchmark run with 4 processes. Table 1 shows the timings for the three benchmarks run as a single MPICH-G2 job with 4 CPUs on one host (jobtype=mpi) or split as two subjobs with 2 CPUs in each subjob (2+2). The latter case was run with (jobtype=mpi) and with (jobtype=single), with both 2-CPU subjobs on the same host (hopper or steger) or split between two hosts (2 CPUs on hopper and 2 CPUs on steger).

All experiments were run numerous times over a period of three weeks. The reported timings are averages of the 5 lowest elapsed walltimes as reported by each benchmark, and, therefore, represent the best case scenarios that one could expect on production machines. Both hopper and steger are SGI Origin 2000 machines containing 250 MHZ IP27 processors and are located in the same machine room at NASA Ames Research Center.

The versions of the compiler and MPT (message passing toolkit) modules used in all the calculations for Table 1 correspond to MIPSpro.7.3.1.1m and mpt.1.4.0.3, respectively. The compilation of all the executables used the MPICH-G2 provided script /globus/mpich-64/bin/mpif77 and compiler options "-O2 -64".

Table 1: Timings on one host versus split between hopper and steger (elapsed times in seconds)

code	one host (4) (mpi)	one host (2+2) (mpi)	one host (2+2) (single)	hopper+steger (2+2) (mpi)	hopper+steger (2+2) (single)
lu.A.4	270	282	282	281	281
sp.A.4	353	367	367	401	401
bt.A.4	651	655	655	655	655

The first, and most important, conclusion one infers from the data in Table 1 is that there is no performance penalty asscociated with using (jobtype=single) instead of (jobtype=mpi). The average of the 5 lowest elapsed walltimes are identical for the MPICH-G2 jobs run with either jobtype. Comparing the timings for the (2+2) split subjobs versus the unsplit case in column 1, one sees a 4% increase in time for the split case in lu.A.4, less than 1% increase in bt.A.4, and for sp.A.4, there is either a 4% increase when the subjobs are on the same host or 14% increase when the subjobs are on different hosts. From these numbers, one can infer that, of the 3 benchmarks, bt.A.4 contains the least amount of communication (in a relative sense) between processes of ranks 0 and 1 with those of ranks 2 and 3. Data for sp.A.4 shows the expected behaviour that the timings increase when subjobs are split between hosts rather than being on the same host. Although not indicated in Table 1, the best single host timings were obtained on hopper.

It is interesting to see how these timings change when the two subjobs are split between hosts that are geographically separated at great distances. One 2-CPU subjob was run on hopper or steger on the West Coast and the other 2-CPU subjob was run on rogallo or whitcomb on the East Coast.

The default compiler on rogallo/whitcomb is version MIPSpro.7.3.1.2m and the MPT module is the older mpt.1.2.1.0. The newer mpt.1.4.0.3 module was not available on either rogallo or whitcomb. To verify that the older mpt module did not contribute to any performance penalties on rogallo/whitcomb, the unsplit 4-CPU MPICH-G2 jobs were re-run on rogallo/whitcomb. The timings for these runs are shown in column 1 of Table 2, and they show no performance degradations in using the older mpt module. Rogallo/whitcomb are also Origin 2000 machines containing 250 MHZ IP27 processors, but they are smaller machines. Rogallo contains 4 processors, whitcomb contains 16, while hopper and steger contain 64 and 256 processors, respectively.

Not surprisingly, the elapsed walltimes increase dramatically when the communication needs to go

across large distances. The increase in times for the best case scenarios are between 54% and 180%. In fact, there is quite a bit more scatter in the raw data due in part to the unpredicability of the network traffic. Jobs on rogallo/whitcomb are not run on dedicated nodes and care must be exercised to avoid interference from other jobs. Most of the data were collected when there were no other jobs running on rogallo/whitcomb. There is the possibility to improve the quality of service provided by the network, but that investigation is outside the scope of this work.

The important point to note again is that there is no performance penalty associated with using (jobtype=single) instead of (jobtype=mpi).

Table 2: Timings on one host versus split between steger (or hopper) and whitcomb (or rogallo) (elapsed times in seconds)

code	whitcomb (4) (mpi)	steger+whitcomb (2+2) (mpi)	steger+whitcomb (2+2) (single)
lu.A.4	268	412	412
sp.A.4	352	986	983
bt.A.4	652	1023	1016

Larger CPU experiments were run with Class B benchmarks. The results are shown in Table 3. Again, no performance penalty is seen with using (jobtype=single). The most surprising, and as yet unexplained, result is the significantly larger timings for bt.B.16 when the two subjobs are run on the same host as opposed to separate hosts. Similar to the Class A results, the sp.B.16 benchmark incurs the greatest performance penalty when split between two separate hosts.

Table 3: Timings on one host versus split between hopper and steger (elapsed times in seconds)

code	one host (16) (mpi)	one host (8+8) (mpi)	one host (8+8) (single)	hopper+steger (8+8) (mpi)	hopper+steger (8+8) (single)
lu.B.16	313	335	335	334	334
sp.B.16	355	411	411	520	520
bt.B.16	676	750	753	691	690

Finally, in Table 4, the effect of requesting extra CPUs for the shepherd processes on the runtimes is explored. Each subjob contains its own shepherd. Comparing the timings in Table 4 with the corresponding ones in Table 3, we see that the performance improvement could be as little as 1% to as much as 9% for the bt.B.16 benchmark run split on one host. The odd man out is the lu.B.16 benchmark which actually shows a 1% performance degradation when run with extra CPUs and split on one host. Generally, allocating extra CPUs for the shepherd processes helps the much larger CPU count jobs more than the smaller CPU count jobs. On hopper/steger, allocating an extra node (2 CPUs) for each shepherd process when each subjob requires only 8 CPUs provides a greater opportunity for processes to be scheduled on nodes that are further away from their communicating partners and their private data. Instead of the 8 processes for each subjob being confined to a physically "tight" 8 CPU cluster, the extra

node for the shepherd process provides opportunity for some communication and memory access to be further apart.

Table 4: Timings for runs with scripts that account for shepherds (elapsed times in seconds)

code	one host (8+8+2shepherds) (single)	hopper+steger (8+8+2shepherds) (single)
lu.B.16	338	329
sp.B.16	391	499
bt.B.16	693	689

The RSL used to run the lu.B.16 benchmark in the last column of Table 4 is:

```
+
(& (resourceManagerContact="steger.nas.nasa.gov")
   (rsl_substitution = (nproc "9") )
   (count = $(nproc) )
   (maxWallTime=15)
   (jobtype=single)
   (environment = (GLOBUS_DUROC_SUBJOB_INDEX 0) (NP $(nproc)) )
   (executable = /path_to_user_script_on_steger/lu.B.16.shepherd.scr)
   (stdout = /path_to_stdout_file_on_steger)
   (stderr = /path_to_stderr_file_on_steger)
)
(& (resourceManagerContact="hopper.nas.nasa.gov")
   (rsl_substitution = (rproc "9") )
   (count = $(nproc) )
   (maxWallTime=15)
   (jobtype=single)
   (environment = (GLOBUS_DUROC_SUBJOB_INDEX 1) (NP $(nproc)) )
   (executable = /path_to_user_script_on_hopper/lu.B.16.shepherd.scr)
   (stdout = /path_to_stdout_file_on_hopper)
   (stderr = /path_to_stderr_file_on_hopper)
)
```

and the user script, lu.B.16.shepherd.scr, is:

```
#! /bin/csh
source /opt/modules/modules/init/csh
module load mpt.new
set numproc =  `expr $NP - 1`
mpirun -np $numproc /path_to_executable/lu.B.16
```

Note that the 'count' RSL parameter is chosen to be 9 to account for the extra shepherd process in each subjob, and the user script substracts that extra process out (numproc = 9 - 1 = 8) to start the mpirun with the correct number of processes.

Summary

The three key ingredients to using modules with MPICH-G2 are:

1. The 'executable' parameter in the RSL is a user-provided script which enables the module command by sourcing the file:
   /opt/modules/modules/init/csh -- for C shell
   (equivalent files are available for tcsh, bash, ksh, or sh)

2. The 'jobtype' parameter must be 'single' to run a user script.

3. The 'environment' parameters GLOBUS_DUROC_SUBJOB_INDEX must be set for each subjob starting from 0 to the number of subjobs minus 1.

Several issues regarding correctness, limitations, and performance associated with using (jobtype=single) instead of the prescribed (jobtype=mpi) were addressed in the examples. The approach presented here is robust since it involves running a user script as the executable.

Acknowledgment

I thank Ray Turney for helping me get started with running the NAS Parallel Benchmarks (NPB2.3) and both Samson Cheung and Scott Emery for useful discussions on the NPB2.3 timing results.

Appendices

A.1

```
========================================================================
Subject: Re: request for OVERFLOW that demonstrates bug?
   Date: Wed, 18 Apr 2001 10:21:58 -0500
   From: "Nicholas T. Karonis" <karonis@olympus.cs.niu.edu>
     To: <recipient_list_omitted>


we have figured why testg2.F (and therefore overflow) hangs when you use
mpich-g2 but why it doesn't hang when using sgi's mpi.  unfortunately, the
root of the problem is an error in sgi's implementation of mpi.

the code below (bad.c) is a distillation of the problem that reliably
reproduces the hang when using mpich-g2 and setting procA = 1; procB = 2;.
note that the code below always works (independent of procA,procB settings)
when using sgi's mpi.

-------- bad.c
#include <mpi.h>
#include <stdio.h>

/*
 * intended to be run with at least 3 procs
 */
int main(int argc, char ** argv)
{
    MPI_Comm new_intercomm;
    int my_rank;
```

```
    int rrank;
    int procA, procB;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    printf("%d: Entering main()\n", my_rank); fflush(stdout);

    /* pick one of the following two settings for procA,procB */

        /* uncomment these and program will work */
        procA = 0; procB = 2;

        /* uncomment these and program will hang */
        /* procA = 1; procB = 2; */

    if (my_rank == procA || my_rank == procB)
    {
        if (my_rank == procA)
        {
            rrank = procB;
        }
        else
        {
            rrank = procA;
        }

        printf("%d: Calling MPI_Intercomm_create()\n", my_rank);
fflush(stdout);
        MPI_Intercomm_create(MPI_COMM_SELF, 0,
                             MPI_COMM_WORLD, rrank,
                             0, &new_intercomm);

    }

    printf("%d: Calling MPI_Finalize()\n", my_rank); fflush(stdout);
    MPI_Finalize();

} /* end main() */
-------- bad.c
```

this raises the question (the one you posed) why does the code above work with sgi's mpi but not with mpich-g2? the answer is in mpich-g2's implementation of MPI_Intercomm_create. mpich-g2 implements some mpi functions by calling one or more vendor-supplied mpi functions. for example, MPI_Intercomm_create is implemented by calling sgi's MPI_Intercomm_create followed by a call to sgi's MPI_Comm_dup (the details of _why_ mpich-g2 does this are too complicated to describe over email).

consider the code below (root_of_problem.c, a slight modification of the example program above) which approximately models the calls mpich-g2 makes to sgi's mpi in implementing MPI_Intercomm_create. if you compile and run the program below using sgi's mpi with procA = 1; procB = 2; i think you will find that it will hang.

```
-------- root_of_problem.c
#include <mpi.h>
#include <stdio.h>

/*
 * intended to be run with at least 3 procs
 */
```

```
int main(int argc, char ** argv)
{
    MPI_Comm new_intercomm;
    MPI_Comm new_comm;
    int my_rank;
    int rrank;
    int procA, procB;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    printf("%d: Entering main()\n", my_rank); fflush(stdout);

    /* pick one of the following two settings for procA,procB */

        /* uncomment these and program will work */
        procA = 0; procB = 2;

        /* uncomment these and program will hang */
        /* procA = 1; procB = 2; */

    if (my_rank == procA || my_rank == procB)
    {
        if (my_rank == procA)
        {
            rrank = procB;
        }
        else
        {
            rrank = procA;
        }

        printf("%d: Calling MPI_Intercomm_create()\n", my_rank);
fflush(stdout);
        MPI_Intercomm_create(MPI_COMM_SELF, 0,
                             MPI_COMM_WORLD, rrank,
                             0, &new_intercomm);

        printf("%d: Calling MPI_Comm_dup()\n", my_rank); fflush(stdout);
        MPI_Comm_dup(new_intercomm, &new_comm);
    }

    printf("%d: Calling MPI_Finalize()\n", my_rank); fflush(stdout);
    MPI_Finalize();

} /* end main() */
-------- root_of_problem.c
```

unfortunately, there's not much that we can do in mpich-g2 to resolve this problem ... certainly not in the short term. we would have to re-design that portion of both the mpich and the globus2 device layers to "code around" this error in sgi's implementation of mpi.

other possible alternatives are (a) modify overflow to avoid triggering the problematic sgi mpi code and/or (b) petition sgi to correct their implementation of mpi. i don't know how much 'influence' nasa and/or the ipg has with sgi, but the later may be a reasonable alternative to pursue.

i'm sorry that the news could not have been more hopeful. i know that it would have been better to hear that you uncovered a bug in mpich-g2 that we have/would fix.

nick

==========

A.2

==========
Subject: solution may be as simple as an upgrade
Date: Fri, 20 Apr 2001 16:55:10 -0500
From: "Nicholas T. Karonis" <karonis@olympus.cs.niu.edu>
To: <recipient_list_omitted>

there have been a couple of people at sgi that have looked at the root_of_problem.c file i sent. it looks as though that this is a bug that has been fixed ... you may need only upgrade to a later version of sgi's mpi. here is what i've been told throughout the course of the day.

1. howard pritchard tells me that the bug was fixed as of MPT 1.4.0.2 (he was able to reproduce the hang in MPT 1.4.0.1, but it passes with MPT 1.4.0.2). they are currently up to MPT 1.5.

2. bonita mcpherson contacted bron nelson and he ran root_of_problem on turing and found that it ran to completion when he used "module swap mpt mpt.1.4.0.3".

could you please try your testg2.F with mpich-g2, but making sure that you are using sgi's MPT 1.4.0.2 or later? if that runs to completion, could you then try overflow with MPT 1.4.0.2 or later?

please let me know how things go.

nick

==========

B.1

==========
The standard "trick" to provide an instantaneous PATH environment is to cause $HOME/.cshrc (or $HOME/.profile) to run just prior to executing the command(s) by using the syntax:

/bin/csh -c <command> (or /bin/sh -c <comand>)

One then only needs to remember that csh (or sh) is in /bin, and the <command> can reside in any directory that is searched from the instantaneous PATH environment.

Thus, while both the commands:

globus-job-run evelyn ls

and

globusrun -s -r evelyn '&(executable=ls)'

will return the error:

GRAM Job submission failed because the executable does not exist (error code 5)

either,

globus-job-run evelyn /bin/csh -c ls

or

globusrun -s -r evelyn '&(executable=/bin/csh)(arguments="-c ls")'

will produce the expected result (a listing of $HOME). Perhaps the only unexpected aspect is that running /bin/env or /bin/printenv under Globus on the NAS IPG machines shows that the PATH environment is already defined. This is due to a modification of the Globus source at NAS that invokes $HOME/.cshrc *after* the Globus job starts at the target location at NAS. Prior to launching the Globus job, the executable is searched only in $HOME if no path (relative or absolute) to the executable is provided. See the Globus Quick Start Guide for other examples of using this method.

==========

B.2

==========
Subject: Re: [Globus-discuss] Interesting Problem
Date: Thu, 06 Sep 2001 11:13:11 -0500
From: Steve Tuecke <tuecke@mcs.anl.gov>
To: Allen Holtz <Allen.Holtz@grc.nasa.gov>
CC: discuss@globus.org

The GRAM services does not source your local dot files. This was a very conscious design choice, for the following reasons:

* Starting a shell, sourcing user environments, etc add significant overhead to job startup path. Many applications do not require this. If you want things run under a normal shell environment you can build this yourself as appropriate. But you don't want to impose this on all jobs.

* Much of the point of GRAM is to not require users to have to customize each and every machine to which they submit jobs. So the base assumption is that you have no assumed local environment, and its up to the submitting job to build up the environment it needs. Various hooks are supplied with GRAM to help you bootstrap up, such as RSL variable that you can use in your submission to find the local globus install path (GLOBUS_INSTALL_PATH), a script (globus-sh-tools) that you can source to get full paths for a bunch of common programs, etc. I'm sure there is much more that could be done to improve this, though we have no specific plans at the moment.

* The whole model of assuming a shell with user dot files breaks down with some scheduling systems, and some resource setups.

-Steve

At 10:21 AM 9/6/2001, Allen Holtz wrote:

>Hi,
>
>   We've got a user here who is trying to run gsincftp from
>a Globus submission. Every time he tries to run the job it
>fails because gsincftp cannot be found. It appears that
>Globus is not obtaining the PATH environment variable. Our
>job manager is LSF.
>
>   When I submit a job, say "printenv," directly to LSF I get
>back several environment variables including the PATH variable.
>When I submit the job using Globus there are several environment
>variables that are not set, including PATH. So this leads me
>to believe that somehow our login scripts are not run when we
>submit the job through Globus. Has anyone else run into this
>problem?
>
>Thanks,
>
>Allen
>--
>Allen Holtz
>Phone: (216)433-6005
>NASA Glenn Research Center
>21000 Brookpark Road
>Cleveland, OH   44135

Subject: Re: [Globus-discuss] Interesting Problem
   Date: Thu, 06 Sep 2001 12:29:20 -0400
   From: Gabriel Mateescu <gabriel.mateescu@nrc.ca>
Organization: NRC
     To: Allen Holtz <Allen.Holtz@grc.nasa.gov>
     CC: discuss@globus.org

<initiating email in the thread snipped>

It appears that globus-jobmanager only sets up $HOME
and $LOGNAME, without creating a login shell for
the user, which is probably a design decision.

One can set the user environment, though. For example,
one can issue

% globus-job-run <host_name> /bin/csh -c "source .cshrc; printenv"

Gabriel

Subject: Re: [Globus-discuss] Interesting Problem
   Date: Thu, 06 Sep 2001 14:18:35 -0500
   From: Doru Marcusiu <marcusiu@ncsa.uiuc.edu>
     To: Allen Holtz <Allen.Holtz@grc.nasa.gov>
     CC: <recipient_list_omitted>

Allen,

The environment for an LSF batch job is passed on from the environment from which the job was submitted. In your case, once you logged on and your .cshrc or .profile files were executed then your PATH variable was properly set. Then when you submitted a job directly to LSF from within that same shell then the batch job will inherit the PATH variable as it was set for you in your submitting shell.

Globus doesn't execute your local startup files for your batch jobs. AT NCSA we suggest to our users that they always submit shell scripts as there batch jobs. Then, within the shell script one can execute any appropriate startup files such as .cshrc or .profile to obtain the desirable environment for their batch job.

<initiating email in the thread snipped>

==========