Preparing the Grid for Scientific Applications

Clay Lindsay
Department of Physics and Astronomy
University of Victoria

Physics Co-op Work Term Report

in partial fulfillment
of the requirements of the Physics Co-op Program

September 17, 2004

Abstract

Today's high energy physics experiments are exceeding the limits of our computational capabilities. Single computers or even clusters of processors are insufficient for the projects physicists are undertaking. One facility can not provide the space necessary to store the massive collections of data produced. Worldwide collaborative efforts such as the BaBar and ATLAS projects require a worldwide collaboration of computational resources. One solution is the grid. Analogous to the world wide web and its proficiency in communication, the grid paradigm describes a transparent interface to computational resources. Grid Canada, a coordinated effort to implement grid technology, has produced GridX1, a computational grid that allows Canada's resources to be pooled together and made available to physicists worldwide. To date GridX1 is only able to handle applications tailored to its hardware. Lacking key components such as data management and a transparent implementation, GridX1 is still in its infancy. To make GridX1 into a practical platform for everyday academic use, a set of tools must be developed. This report details the architecture of GridX1, requirements for grid-enabling scientific applications, and the use of tools developed throughout the term in executing applications on the grid.

Contents

List of Figures

1 Introduction

The purpose of a computational grids is to provide storage and processing power to consumers. An ideal grid is able to allocate and distribute computational resources independent of the use. Similar to how an electrical power grid can provide electricity generated in Northern Ontario to run an air conditioner in Saskatoon, a computational grid can provide CPU time in California to a High Energy Physics (HEP) application submitted in Switzerland. Grid computing is undoubtedly the next step in scientific computation. New projects conceptualized in fields such as High Energy Physics and Bio-Informatics exceed the practical limits of on-site computation. One notable example of the use of grid architecture is the Large Hadron Collider Computing Grid (LCG).

The Large Hadron Collider(LHC) currently under construction in Geneva, is undoubtedly one of the largest and most expensive scientific endeavors in history. At a colossal 12-14 petabytes per year, the projected data output from experiments using the LHC dwarf even some of the largest data storage facilities in the world. Projects such as ATLAS require unprecedented computing power to the tune of 70,000 modern processors [1]. To meet this immense challenge, the LCG was created. LCG's main components include databases, storage and processors which all must be dedicated to LCG's infrastructure. Components are linked together via a complex software package and high speed network. Facilities that wish to join the LCG are required to provide resources for the soul use of the LCG. Now with an equivalent computing power of over 4,000 PC's [2], LCG is beginning to emerge as the answer to LHC's data challenges. LCG is likely the largest grid project in the world, but many other groups are making contributions to grid software and architecture.

One of Canada's major grid efforts is Grid Canada, a partnership between the National Research Council, CANARIE and C3.ca. Grid Canada's main

focus is implementing grid technology and making it available to various fields that require scientific computation. UVic's grid research group, lead by Dr. Randall Sobie, is a large contributor to grid software development and design. GridX1 is Grid Canada's current experimental computational grid. Comprised of 3 sites nationwide, GridX1 has access to roughly 300 modern processors and 2TB of disk space. GridX1 employs a shared resources philosophy in order to link together Canadian computational resources in a way that allows these resources to still be used by entities outside of the grid. GridX1 is an effort to add an extra layer of communication on top of existing network architecture while leaving as little footprint as possible.

The goal of GridX1 is to run real applications. While existing grid and scheduling packages are able to provide a backbone for grid operations, end users can not be expected to know the fine details of the grid. A consumer that wishes to use the grid needs a simple interface through which he can execute his applications. In order to make GridX1 viable to consumers, it needs to be as simple as submitting a job to a local cluster of computers. A grid should appear as a cluster; the user doesn't need to know it spans worldwide. To accomplish this a set of tools needs to be developed for data management, submission, installation and scheduling.

2 Discussion

2.1 GridX1

2.1.1 Architecture

GridX1 is made up of 4 main components. Figure 1 shows how the submit machine, resource broker, high speed network and resources connect to form a grid.

1. High Speed Network

   This connects the resources of a grid together. A high speed network is necessary to allow for the fast transfer of potentially large amounts of data. GridX1 makes use of a light path based gigabit network.

2. Resource Broker

   This portion of the grid matches jobs (instances of application execution) to resources. Acting as a gateway to the grid, the resource broker receives and processes all job submissions. GridX1 employs a central resource broker design where only one broker is used for the entire grid. The central resource broker is aware of all jobs executing on the grid, as well as the status of each resource providing services. The broker passively receives status reports from each resource and intelligently distributes jobs based on factors such as wait time, free CPU's and data location.

3. Computational Resources

   These are any entity providing a service to the grid such as data storage and processing power. GridX1's computational resources consist of the University of Victoria's Mercury cluster, the University of Alberta's Thuner cluster and NRC's Mercury cluster. Each is responsible for keeping the resource broker, located in Vancouver, informed of its status. The status of a resource includes information such as estimated wait time (for clusters) and disk storage remaining (for storage facilities).

4. Submit Machine

This is the user's window onto the grid. A submit machine is where a consumer submits the parameters of his job to the resource broker. All submit machines are connected to the resource broker via a network or Internet connection. Normally, the information passed in submission is small relative to the data used in application execution; a high speed network is not necessary to connect the submit machine and resource broker.

Figure 1: Basic grid structure

2.1.2 Software

GridX1's existing software architecture is, more or less, two pieces of software: The Globus Toolkit and Condorg. These two form a stable, well tested base for a grid.

As de facto standard in grid computing software, the Globus Toolkit accommodates most basic grid operations such as file transfer, authentication, job submission and monitoring. The toolkit provides a set of binaries and an API to the C programming language through which, with an expert knowledge of the grid, a user can execute any grid operation.

Condor is a well known scheduling program that organizes the distribution of jobs on clusters of computers. CondorG is an extension to Condor to the Globus Toolkit so that it can be used as a scheduler for a grid. CondorG makes up the base for resource brokering on GridX1. Each resources submits a "classad" to CondorG periodically to keep the resource broker up to date in the status of the grid.

While these pieces of software make for a solid backbone to GridX1's structure, they lack key components that are required to run applications on the grid in a practical manner.

2.2 Requirements

Before developing a set of tools for grid enabling scientific applications, the requirements of the applications and the grid need to be examined.

2.2.1 Grid Requirements

A grid architecture needs to meet specific requirements in order for it to be useful to consumers.

1. Transparency

   Execution of applications on the grid should ideally be no more difficult than submitting a job to a cluster of processors. Consumers should not require an expert knowledge of the grid to be able to use it. Transparency describes a grid's ability to abstract consumers from the complicated architecture beneath their submit machine. To make the grid a practical tool in scientific computing, a transparent method of job submission and application installation is required. Without transparency, a grid expert would be required for every consumer that wished to make use of the technology. This would be akin to requiring an electrician to plug in a

blender. This is wasted resources when the architecture could be built with transparency in mind. GridX1 has effective methods for transparent job submission but application installation and data management still require an intimate knowledge of the grid.

2. Scalability

   The ability to expand is of key importance to the grid paradigm. Scalability is the ability for a grid to add resources and applications. There is no doubt demand for computational resources is growing. To meet this demand, a grid architecture needs to be able to add new resources seamlessly. GridX1 uses a software layer that sits on top of existing network architecture. This non-invasive software approach allows groups to add their resources to GridX1's pool without interruptions to the resource's other tasks.

3. Efficiency

   One of the main goals of a grid architecture is to allow the efficient execution of applications. Efficiency is important in many grid operations like submission and data transfer; but most notably, an efficient method of scheduling can severely impact the performance of a grid. GridX1 implements a greedy method of scheduling which assigns jobs to the resource with the lowest expected waiting time. This method does not take into account location of data or transfer times. While the greedy method may be acceptable, it is not the most efficient solution. A scheduler that could employ a more sophisticated scheduling algorithm that takes into account data location, user access and network transfer speeds is desirable.

4. Security

   Security is essential in any public system. GridX1 makes use of the Globus

Toolkit's GSI authentication system to restrict resource and disc access. Each consumer first requests a certificate, from a trusted authority, which allows them access to grid resources. Each certificate is mapped to a user account on each resource. GridX1's disc access restriction relies on the UNIX permission system which is secure and reliable. Under this system, a consumer is only required to have his certificate in a secure location on his submit machine. Through this, the grid is both accessible and secure.

2.2.2 Application Requirements

While it is impossible to predict how to accommodate every application that a consumer may wish to execute on the grid, a few basic requirements, that appear commonly in scientific applications, can be defined.

1. Installation

   Installations can range from copying a single execution file to configuring a complex tree of inter-dependant software. One of the most difficult aspects of grid computing, installation can provide any number of problems to the grid consumer. Often application installation and configuration requires an administrative level of access to the system, meaning that the grid consumer would require administration privileges to each grid resource. This is clearly not acceptable because security would be severely compromised. In addition, installation is difficult to make transparent because of the heterogeneous nature of the grid. It's very difficult to build a system through which to install applications (even simple ones) that works on all resources. Requiring system administrators to preform installations may allow for transparency, but it violates scalability and efficiency. There's no guarantee that a resource administrator has the time to grid-enable all consumer's applications. Despite violations in scalability and efficiency,

GridX1 has administrators and grid developers preform the installations, until a better system is developed.

2. Execution

   Execution is far more simple than installation. Once correctly installed on a resource, execution is left up to the local scheduler (if the resource is a cluster of processors). GridX1's three resources make use of PBS scheduling software to distribute jobs throughout their clusters. This method of execution is transparent to the user, scalable (its safe to assume any new resources would have a similar capacity for job execution), secure and efficient.

3. Data Management

   Data is an extremely important aspect of the majority of scientific applications. Practically every application requires some form of input, and all applications provide output. Data is most commonly stored in files or data bases. Input data is necessary to execute the application and can either be staged in by the consumer, or stored somewhere on the grid. Output data needs to be made available to the consumer on completion of the job. Data management involves the cataloging and movement of data through the grid. GridX1 has the capability to move data quickly, but there is no transparent way to access or store data on the resources. Without a data management system that adheres to the grid requirements, running applications on the grid would be impractical.

2.3 Data Management

GridX1's biggest fault is its lack of data management. To a consumer with no knowledge of the grid, there is no way to register, stage or access data on the

grid. The core of the problem is the lack of a catalogue for keeping track of data. The Globus Toolkit contains a tool for data cataloging in a grid environment: The Replica Location Service.

2.3.1 Replica Location Service

Globus' replica location service (RLS) is simply a database mapping logical file names, or file aliases, to physical locations. The RLS database by default accepts two parameters for each entry: an alias for a file and a physical location. While this seems simple, it is a basic system that fills the data management requirements of GridX1. Each storage resource has a local replica catalogue (LRC) which is a catalogue of data stored there. Resources periodically update a central replica location index (RLI), which maps file aliases to LRCs.

mercury.uvic.ca LRC table

Logical File Name	File Location
Foo.txt	/home/griduser/foo.txt
test.empty	/home/griduser/tests/test.empty

thuner-gw.phys.ualberta.ca LRC table

Logical File Name	File Location
Foo.txt	/homes/gridX1/gc1/Foo.txt

Central RLI table

Logical File Name	LRC Host
Foo.txt	mercury.uvic.ca thuner-gw.phys.ualberta.ca
test.emtpy	mercury.uvic.ca

Figure 2: Sample RLS tables

Figure 2 shows a simple example of a file registered on the grid. "Foo.txt" is the file alias and it exists on the "mercury.uvic.ca" resource at the location "/home/griduser/foo.txt" in a UNIX file system. Mercury's LRC database entry for "Foo.txt" contains the mapping "Foo.txt = /home/griduser/foo.txt". The central RLI database contains the mapping "Foo.txt = mercury.uvic.ca" as well as "Foo.txt = thuner-gw.phys.ualberta.ca" because Foo.txt also exists on the

thuner-gw resource.

Using this cataloging system, GridX1 is able to add new resources and maintain a unified cataloging system with few changes. While the system is simple and scalable, it is far from transparent and secure. Inaccurate entries can be easily placed into an LRC database which could result in errors when a consumer tries to make use of the data.

2.3.2 Consumer RLS Interface

The development of a consumer-RLS interface stemmed from the need for users to be able to register data with the grid easily. Globus's RLS tools are simple to use, but require that the user enter each data file individually. If a user wishes to register 10,000 data files, this is clearly a daunting task. Rather than requiring consumers to create ways to register data, a script was developed to ease the process.

The Grid Canada File Manager[A] script contains a user friendly interface to the RLS database through which users can register their data. It can be executed from the command line and takes simple commands such as "Reg *.txt" (which registers all files ending in '.txt', into the local LRC database). Other features of the file manager can be found in the documentation. Future development for the file manager includes a web portal, and more layers of security.

While registration is now simple, security issues arise. Despite the file manager's ability to only register existing files, users can still directly access LRC databases and make erroneous, potentially malicious, entries. To prevent this, direct access to the LRC database was limited to administrators, forcing users to use the file manager program to update the database.

2.3.3 Input Data Access

Cataloging is accomplished through Globus's RLS database and copying across the grid is done by Globus's copy routines; but, to make the system transparent, a link between the two is necessary. The basic steps in copying data on the grid are query an RLI database with filename, get LRC name(s) containing file, query that LRC for file location, and run a Globus copy. This is far from transparent. A user running an application needs knowledge of the location of the RLI database, as well as knowledge of Globus commands. To overcome this, a script was written to preform transparent data copying: gccp[A].

Use of this routine involves simple commands such as: "gccp dataset1" which copies the file "dataset1" to the current location. The script first queries an RLI database, queries an LRC based on RLI output, and either Globus copies the file, or creates a symbolic link to the file if it is located locally. A user application can make use of this copy routine to transparently copy any files registered on the grid to the resource the application is running on.

2.3.4 Data Staging and Output

The last big obstacle in data management in a grid architecture is dealing with temporary data. While transparent access is still required, staged input files (provided by the consumer on the submit machine) and application output need to be removed from the grid once their purpose is served. This can be overcome with a caching system.

Caching involves keeping commonly used data close to where it is needed. In the grid paradigm, a "grid cache" holds temporary files that require registration with the grid, but not permanently like data files. A grid caching system was developed for this purpose. Each resource has a specified directory in which consumers or submit programs can place data temporarily. This di-

rectory is tracked by a UNIX daemon which periodically updates the resource's LRC database with the new/removed contents of the directory.

Through this caching system, users are able to stage temporary input files and return output data back to the submit machine, without committing permanent files to grid registration.

2.4 Applications on the Grid

Using the data management tools developed, a generic application can now be executed on the grid with only minor adjustments. A common application execution would consist of:

1. Consumer submits job and staged input files(s) to resource broker
2. Resource broker chooses a resource to execute the job
3. Resource broker caches input file(s) on resource
4. Resource broker submits job to resource
5. Grid Resource executes job script
6. Job script copies input from cache
7. Job script runs gccp to access data
8. gccp grid copies or links local data
9. Job script executes physics application
10. Job script copies output files to resource cache
11. Grid Resource returns status and location of output to RB
12. Resource broker returns output to submit machine and consumer

Grid enablement of an application merely requires the user provide the necessary input files to stage, the input files to copy, and a script to execute the application. This transparent implementation can now be used to grid-enable scientific applications.

2.5 Test Applications

With some basic data management software in place, application testing can begin on GridX1. The main goal of the testing is to work out any bugs in the network architecture. Two applications were used in testing: ATLAS Data Challenge 1 and BaBar BetaMiniApp.

2.5.1 ATLAS Data Challenge 1

The ATLAS data challenges are an effort to prepare the Large Hadron Collider Computing Grid (LCG) for the immense amount of data it will need to process when experiments at LHC begin taking data. Data challenge 1 (DC1) was the first phase of the project. The test application requires an input file (1.5Gb), a database and a local installation of the software. The DC1 was already installed on each resource from tests of a previous grid implementation. The database was set up on each resource to avoid congestion and the input files were evenly distributed between the three grid resources.

Tests consisted of 100 job submissions, over a period of 24 hours, to the resource broker from a standard submit machine. The execution script submitted[C] set up the ATLAS environment, created work space, copied the input, ran the application, and placed the output in the resource cache. Brokering was based solely on expected wait time. Not all jobs required a grid copy, some jobs were submitted to a machine that already had a local copy of the data file.

Figure 3 shows the success rates for each resource taking part in the tests.

Test #	UVic (success/total)	UAlberta (success/total)	NRC (success/total)	All
1	4 / 5	10 / 73	17 / 22	31 / 100
2	0 / 0	14 / 80	20 / 20	34 / 100
3	0 / 0	24 / 44	50 / 56	74 / 100
4	27 / 30	2 / 55	15 / 15	44 / 100
5	50 / 53	20 / 23	23 / 24	93 / 100
6	3 / 3	27 / 30	65 / 67	95 / 100
Total	84 / 91 (92.3%)	97 / 305 (29.8%)	190 / 204 (93.1%)	371 / 600 (61.8%)

Figure 3: Success rate for each resource

Failure Breakdown	UVic	UAlberta	NRC	All
Submission	85.7%	10.1%	0.0%	11.7%
Data Transfer	14.3%	87.0%	14.3%	79.9%
Execution	0.0%	2.8%	85.7%	8.3%

Figure 4: Breakdown of failures

The low number of jobs submitted to the UVic site is due to the cluster maintenance. Test 6 submits only 3 jobs to the UVic site because the cluster is quite busy running production grid jobs. UAlberta receives a large number of jobs in early tests because many jobs failed right away on submission, resulting in that site remaining free for most of the test. The resource broker distributed the jobs as expected given the conditions. No errors were found in the resource broker during tests.

Figure 4 shows the breakdown for errors in the jobs. Most errors occurring at the UAlberta site were due to configuration problems with the cluster that were solved by the end of the tests. UVic's failure rate, far less significant than UAlberta's, is due to submission problems to the resource. NRC, the most successful site, only failed jobs in which it received corrupted data from a transfer.

Through application testing, many bugs in the grid architecture were worked out resulting in an acceptable final success rate of over 90%.

2.5.2 BaBar BetaMiniApp

While the ATLAS DC1 software was a good test platform for all the resources, a practical application still needed to be grid-enabled. The BaBar BetaMiniApp is an application used commonly to analyze data from the BaBar experiment. This application is quite well suited for grid-enablement since it takes most of its data in the form of data files. Since the BaBar environment is complex, there was no time to install it on all three resources. The Mercury cluster at UVic had an existing installation that would be suitable for grid-submission.

The MiniApp requires a database, input files, and configuration files. The database was already set up on Mercury. 10 input files were distributed about the 3 grid resources. The configuration files are constructed by the user, so they needed to be staged onto the grid where they could be accessed.

A script was written to execute simple MiniApp jobs via Globus. Submission involved first staging the configuration files (test.tcl) onto a grid resource's cache then submitting the job script via Globus to the resource broker. Once submitted, the resource broker recognized it required a BaBar installation and routed the job to UVic. Once executed, the job script set up the BaBar environment based on local parameters, copied the staged configuration files from the resource cache, ran gccp to receive its data files and finally executed the BetaMiniApp application. Figure 5 displays the process.

Final tests of the BaBar BetaMiniApp software were successful, proving that practical applications could be grid enabled using the tools developed.

Figure 5: BaBar BetaMiniApp flowchart

3 Conclusion

Basic data management tools have now been developed and tested for GridX1. Application tests have proved helpful in determining bugs in GridX1's resources and have provided promising final results. A commonly used high energy physics application (BaBar BetaMiniApp) was successfully grid enabled indicating GridX1's prospects in practical scientific computation.

4 Recommendations

All of the scripts and routines developed during this work term should be considered prototypes. Initial testing of GridX1 as a platform for scientific computation proved promising, but data management tools should either be further developed or alternative tools should be considered. While grid-enablement of applications is now possible, the process is far from scalable. A uniform method of application installation needs to be developed to ease the grid-enablement

process.

5 Acknowledgements

I would like to thank Randy Sobie for offerring me the opportunity to contribute to the GridX1 project. In addition I would like to acknowledge the contributions of Ashok Agarwal, Dan Vanderster and Lila Klektau in the development and implementation of GridX1's hardware and software architecture. Without their contributions this report would not have been possible.

6 Appendices

A "gccp" Program

```
#!/usr/bin/perl -w

# Set perl @INC variable to include Globus module
BEGIN {
    chomp($libloc = `echo ~gcprod01/bin`);
    unshift(@INC, $libloc);
};
use Globus;
use Getopt::Std;

# Set default RLI server
$RLIHOST = "grid.phys.uvic.ca" ;

# Check for proper environment or stop
my $dest=(), $desttype=(), @filelist=(), $cache="NULL", $localhost = `echo
\$GLOBUS_HOSTNAME`;
die "\$GLOBUS_HOSTNAME environment variable not defined.\n" if ($localhost eq "\n");

sub help();
sub printv($);
sub linkfile($);

%options=();

# Get options and assign destination server, type, directory
getopts("hvs:d:c", \%options) or help();
help() if defined $options{h};
if ( defined $options{s} ) {
    $destserv = $options{s} ;
    chomp ($destdir = (defined $options{d}) ? $options{d} : globus("echo \$HOME",
     $destserv) );
    $desttype = "server";
    eval{$globusout = Globus::run("source ~gcprod01/gcsource ; echo \$RLSHOST ;
     echo \$GC_CACHE", $destserv)};
    die "Error contacting destination server: $@\n" if $@;
    ($destlrc, $cache) = split(/\n/, $globusout);
}
else {
    chomp ($destserv = `echo \$GLOBUS_HOSTNAME`);
    chomp ($destdir = (defined $options{d}) ? $options{d} : `pwd`);
```

```
    $desttype = "local";
    chomp ($destlrc = `echo \$RLSHOST` ) ;
    chomp ($cache = `echo \$GC_CACHE`) if (defined $options{c});
}

# Die if no files were input to copy
die "Error:  You need to enter a file to copy!\n" if (scalar @ARGV == 0);
die "Error:  Environment variables no properly set on $destserv\n" if ($destlrc
eq "\n" || $cache eq "\n");

# Retrieve lfn's of files to be copied
unshift(@filelist, pop(@ARGV) ) while (scalar @ARGV > 0);

# Copy/link each lfn requested

foreach my $lfn (@filelist) {
    my $transfercomplete = 0;

    # Query RLI server for entries matching lfn
    $transfercomplete = 1 if ( Globus::rli_check($lfn, $destlrc, $RLIHOST)
     && linkfile($lfn) );

    # Parse RLI output to find LRC's with required LFN
    if (!$transfercomplete) {
printv("\nScanning RLI $RLIHOST for LFN matching $lfn...\n");
# Extract LRCs from RLI output
eval{%lrcmap = Globus::rls_get_mappings("rli" , $RLIHOST, $lfn)};
if($@) {
    print "Error contacting $RLIHOST: $@\n";
    $RLIHOST = "ontario.iit.nrc.ca";
    eval{%lrcmap = Globus::rls_get_mappings("rli" , $RLIHOST, $lfn)};
    if ($@) {
die "Could not contact backup rli: $RLIHOST:$@\n";
    }
}
@lrclist = values (%lrcmap);

# Correct for multiple LRC returns
if (scalar @lrclist == 1 && "@lrclist" =~ /\s/) {
my $temp = "@lrclist";
$temp =~ s/ $lfn//g;
@lrclist = split(/ /, $temp);
}

if (scalar @lrclist == 0) {
    print "Error: LFN \"$lfn\" was not found in RLI database.\n";
```

```
    next;
}
printv ( (scalar @lrclist)." LRC(s) found with $lfn\n" );
    }

    # Pick a host at random to copy from, attempt to copy, and repeat with
     another host if failure
    while ( !$transfercomplete  && (scalar @lrclist > 0) ) {

        $srcserv = "";
@pfnlist = ();
# Pick a random host
my $random = int(rand(scalar @lrclist));
$lrc = $lrclist[$random];
# Determine host LRC's resource URL from RLS entry
eval{$srcserv = Globus::rls("query lrc lfn", "GC_RESOURCE", "$lrc")};
if ($@) {
    print "Error determining resource associated with LRC $lrc:
     $@\nTrying another LRC...\n";
    splice (@lrclist, $random, 1);
            next;
}
$srcserv =~ s/  GC_RESOURCE: //;

# Query LRC to find PFNs
printv("\nScanning LRC $lrc for LFN matching $lfn...\n");
eval{%pfnmap = Globus::rls_get_mappings("lrc", $lrc, $lfn)};
if ($@) {
    printv "Error contacting LRC: $@\n";
    splice (@lrclist, $random, 1);
            next;
}
(@pfnlist) = values (%pfnmap);

# Correct for multiple LRC returns
        if (scalar @pfnlist == 1 && "@pfnlist" =~ /\s/) {
                my $temp = "@pfnlist";
                $temp =~ s/ $lfn//g;
                @pfnlist = split(/ /, $temp);
        }

printv( (scalar @pfnlist)." copies of $lfn found in $lrc\n");

# Pick a PFN at random (if more than one) and attempt to copy
while ( !$transfercomplete && (scalar @pfnlist >0) ) {
```

```
    # Pick a random PFN
    my $random2 = int(rand(scalar @pfnlist));
    $pfn = $pfnlist[$random2];

    # If cache option is chosen, cache the file and link
    if ( defined $options{c}) {
printv "\nCaching $lfn\nFrom: $srcserv:$pfn\nTo:
$destserv:$cache\n";

# Cache the file
$type = ($desttype eq "local") ? "file://" :
"gsiftp://$destserv";
chomp( $copyout = `globus-url-copy gsiftp://$srcserv$pfn
$type$cache/$lfn 2>&1` ) ;
printv "Creating link to cached file $destdir/$lfn to
$cache/$lfn\n";
# Create link
my $linkout = ($desttype eq "local") ? `ln -s $cache/$lfn
/$destdir/$lfn` : Globus::run("ln -s $cache/$lfn
$destdir/$lfn", $destserv);

    }
    else {
printv "\nCopying $lfn\nFrom: $srcserv$pfn\nTo:
$destserv$destdir/$lfn\n";

# Copy the file
$type = ($desttype eq "local") ? "file://" :
"gsiftp://$destserv";
chomp( $copyout = `globus-url-copy gsiftp://$srcserv$pfn
$type$destdir/$lfn 2>&1` );
    }

    # If copy was successful, set success flag
    if ( $copyout eq "" ) {
printv( "Transfer successful.\n\n");
$transfercomplete = 1;
    }

    # Copy unsuccessful, remove PFN from list and try to copy from
     other file
    else  {
print "An error was encountered copying $lfn from $srcserv$pfn.
Attempting to copy from another location...\n";
print "$copyout\n";
splice (@pfnlist, $random2, 1);
```

```
    }

}

# If transfer from LRC was unsuccessful, remove from list and attmempt
copy from another LRC
if (!$transfercomplete ) {
    print("Failed to copy from $srcserv, Trying another location...\n")
     if (@lrclist > 1);
    splice (@lrclist, $random, 1);
}

    }

    # Failed to copy from any PFN on any LRC
    if (scalar @lrclist == 0 && !$transfercomplete ) {
print "Failed to copy $lfn from GC resources\n";
    }
}


sub help() {
    print <<eot;
    Usage: gccp [-h] [-v] [-s hostname] [-d destdir] [-c] file1 file2 file3...

where

-h displays this page

-v executes in verbose mode (recommended)

-s hostname sets the destination server (default current machine) to
hostname

-d destdir sets the destination directory (default current directory)
to destdir

-c caches the file on the destination machine and creates a link in
the destination directory

-fileX name of file to be copied

eot
    exit(0);
}
```

```
sub printv($){
    print $_[0] if (defined $options{v});
}


# Linkfile checks an LRC for a PFN to input LFN and
# creates a link.  This function is tightly coupled with the main script.
sub linkfile($) {
    my $lfn = shift;

:    my %mappings = Globus::rls_get_mappings("lrc", $destlrc, $lfn);
    my (@pfnlist) = values (%mappings);

    print "linkfile\n";

    while (scalar @pfnlist > 0) {
$pfn = pop(@pfnlist);
printv "Requested file exists on GC node;\nCreating link from
$pfn to $destserv/$destdir/$lfn\n";
$linkout = ($desttype eq "local") ? `ln -s $pfn /$destdir/$lfn` :
globus("ln -s $pfn $destdir/$lfn", $destserv);
return 1 if ($linkout eq "" || $linkout =~ /File exists/);
    }


    return -1;


}
```

B "gcfm" Program

```
#!/usr/bin/perl -w

# Set perl @INC to inlcude Globus module
BEGIN {
    chomp($libloc = `echo ~gcprod01/bin`);
    chomp($RLSHOST = `echo \$RLSHOST`);
    unshift(@INC, $libloc);
};
use Globus;
```

```
use Getopt::Std;

# Get "preserve" or "recursive" arguments
%options = ();
getopts("ph", \%options);

# Subroutine Getmappings:  Confirms the existance and returns locations of
files and wildcard expansions received as arguments
# Input: file names or wildcard expansions
# Output: A hash containing logical filename keys and physical filename values
of existing files corresponding to input
sub getmappings {
    my $args = shift;
    $mapopts{r} = 1 if ($args =~ s/-r //gi);
    $mapopts{p} = 1 if ($args =~ s/-p //gi);

    my @wilds = split(/[\s]+/ , $args);
    my $pwd = `pwd`;
    my $recurse = (defined $mapopts{r}) ? "" : "-maxdepth 1";
    chomp($pwd);

    foreach $wild (@wilds) {
$dir = "";
($dir,$wild) = $wild =~ /(.*\/)(.*)$/ if ($wild =~ /\//);
$findout = `find $dir -name '$wild' -type f $recurse`;
$findout =~ s/\.\///g;
@matches = split(/\n/, $findout);
$error = (-d $wild) ? "ommitting directory $wild\n" : (scalar @matches == 0)
? "could not match \"$wild\"\n" : "";
print $error;
push(@files, pop(@matches)) while (scalar @matches > 0);
    }

    for ($i = 0 ; $i < scalar @files ; $i++) {
$lfn = $files[$i];
($lfn) = $lfn =~ /^.*\/(.*)/ unless (defined $mapopts{p} || $lfn !~ /\//);
($mappings{$lfn}) .= ( $files[$i] =~ /^\// ) ? " $lfn ".$files[$i] : " $lfn
$pwd/$files[$i]" if (defined $mappings{$lfn});
($mappings{$lfn}) = ( $files[$i] =~ /^\// ) ? $files[$i] : "$pwd/$files[$i]"
unless (defined $mappings{$lfn});
    }
    return %mappings;
}

# Exit if environment not properly set
die "\$RLSHOST environment variable not defined\n" if ($RLSHOST eq "");
```

```
# Retrieve STDIN input
$_ = <STDIN>;
chomp;

# Take action according to input

# Register files with RLS database defined in environment variable $RLSHOST (files must
if ( m/^reg (.*)/ ) {
    my %mappings = getmappings($1);
    Globus::rls_add(\%mappings, "$RLSHOST", 50) if (keys %mappings >0);
}

# Deregister files from RLS database (files must exist)
elsif ( m/^dereg (.*)/ ) {
    my %mappings = getmappings($1);
    Globus::rls_rm(\%mappings, "$RLSHOST", 50) if ( keys %mappings > 0);
}

# Register mapping with RLS database
elsif ( m/^rls-add (.*)/ ) {
    my @args = split( /[\s]+/ , $1);
    die "malformed input: incorrect number of strings to map\n" if  (  ( (scalar @args %
     2) == 1) || (scalar @args == 0) );
    my %mappings = map { $args[$_*2], $args[$_*2 + 1]} 0 .. ( (scalar @args) / 2 - 1);
    Globus::rls_add(\%mappings, "$RLSHOST", 50);
}

# Remove mapping from RLS database
elsif ( m/^rls-rm (.*)/ ) {
    my @args = split( /[\s]+/ , $1);
    die "malformed input: incorrect number of strings to map\n" if  ( ( (scalar @args % 
     == 1) || (scalar @args == 0) );
    my %mappings = map { $args[$_*2], $args[$_*2 + 1]} 0 .. ( (scalar @args) / 2 - 1);
    Globus::rls_rm(\%mappings, "$RLSHOST", 50);
}

# Execute a command on a remote host (job submission)
elsif ( m/^cmd (.*?) (.*)/ ) {
    my ($host, $cmd) = ($1, $2);
    my $globusout = Globus::run("$cmd", $host);
    print "$globusout\n" unless ($globusout eq "");
}
else {
    die "unrecognized command\n";
}
```

C ATLAS Execution Script

```
#!/bin/bash
echo Job Started at: `/bin/date`
echo Job Running on: `/bin/uname -a`

# Set up GC/Atlas environment
source ~gcprod01/gcsource

# GET VARIABLES FROM ARGUMENTS
partition=$1
jobid=$2

# Fix condor attribute bug
if [[ $jobid = "" ]] ; then
    jobid=`echo $partition | sed -e 's/.* //'`
    partition=`echo $partition | sed -e 's/ .*//'`
fi
partition=00$partition

# Record execution in job log
echo Job ${jobid} - started at: `/bin/date` >> $GC_CACHE/joblog

# SET OTHER VARIABLES
VERSION=6.0.2-1
DEBUG=true
DATASET=002000
LUMI=lumi02
DESCR=hlt.pythia_jet_17
INPUTFILE=dc1.${DATASET}.${LUMI}.${partition}.${DESCR}.zebra
EXECFILE=recon.gen.v5.with602rpmkit
JOBOPTFILE=eg7.602.job
MAXEVENT=1000
export JOBNAME=dc1.${DATASET}.${LUMI}.recon.${partition}.${DESCR}
export WORKDIR=$HOME/atlas/${VERSION}/${JOBNAME}_${jobid}
export OUTDIR=$HOME/atlas/${VERSION}/project/dc1/recon/data/${DATASET}

####################
### Start Script ###
####################
```

```
# SETUP
echo Job Setup Started at: `/bin/date`

# Create workspace
mkdir -p ${WORKDIR}
cd ${WORKDIR}

# Copy executables
cp $ATLASDIR/${VERSION}/${EXECFILE} .
cp $ATLASDIR/${VERSION}/${JOBOPTFILE} .

# Grid-copy data file
echo "Running gccp -v ${INPUTFILE}"
gccp -v ${INPUTFILE}

# Execute applications
# ATHENA EXECUTION
echo Job Execution Started at: `/bin/date`
time ./${EXECFILE} ${INPUTFILE} ${JOBNAME}.ntuple ./${JOBOPTFILE} ${MAXEVENT}
>& ${JOBNAME}.log 2>&1

# SAVE IMPORTANT FILES
echo Job Cleanup Started at: `/bin/date`
mv atlas.his         ${JOBNAME}.atlas
mv histo.hbook       ${JOBNAME}.histo

OUTPUTFILE=${jobid}_${JOBNAME}_output.tar

# Tar output and move to cache
tar cf ${OUTPUTFILE} ${JOBNAME}.*
mv ${OUTPUTFILE} $GC_CACHE

# REMOVE WORKING DIRECTORY
cd ${WORKDIR}
cd ..
rm -rf ${WORKDIR}

# Report job finished
echo Job ${jobid} - finished at: `/bin/date` >> $GC_CACHE/joblog
echo "Job output has been cached on `echo $GLOBUS_$HOSTNAME`/`echo $GC_CACHE
`/$OUTPUTFILE."
echo Job Finished at: `/bin/date`
```

D BaBar BetaMiniApp Script

```
#!/usr/bin/perl -w

# Set release and architecture parameters
$REL= "14.5.2";
$ARCH= "Linux24RH72_i386_gcc2953";

use Getopt::Std;

sub printv($);

my %options = ();

# Get arguments
getopts("j:vr:h:f:c:s:d:a:",\%options);
die "One of -r [filename.root] or -h [filename.hbook] is required\n" unless
(defined $options{r} ^ defined $options{h});
die "A ConfigPatch field (-c) is required and must be one of: Run1, Run2, MC\n"
unless (defined $options{c} && $options{c} =~ /MC|Run1|Run2/);
die "A valid source file (-f [sourcefile] ) is required\n" unless (defined
$options{f} && -f $options{f});
die "Invalid levelofdetail type\n" if (defined $options{d} && $options{d} !~
/micro|cache|extend|refit|rebuild/);
die "A job identifier is required\n" unless (defined $options{j});
die "Input collections required\n" unless (scalar @ARGV > 0);
my ($source, $cpatch,  $afiles, $job, @input) = ($options{f} , $options{c},
$options{a},$options{j},@ARGV);
my ($type, $outfile) = (defined $options{h} ) ? ("hbook", $options{h}) :
("root", $options{r});
my $detail = (defined $options{d}) ? $options{d} : "cache";
my ($sourcename) = $source =~ /.*\/(.*)$/;
$sourcename = $source unless (defined $sourcename);

# Print verbose information
printv "JobID: \t\t\t$job\n";
printv "sourceFile: \t\t$source\n";
printv "ConfigPatch: \t\t$cpatch\n";
printv "$type Output: \t\t$outfile\n";
printv "level of detail: \t$detail\n";
printv "Input Collections: \t@input\n";

# Create main TCL file
open TCL, ">$job.tcl";
print TCL "lappend inputList";
```

```
foreach $kanga (@input) {
    print TCL " $kanga";
}
print TCL "\n";
print TCL <<eot;
set levelOfDetail "$detail"
set ConfigPatch "$cpatch"
set BetaMiniTuple "$type"
set histFileName "$outfile"
sourceFoundFile $source
eot
close TCL;

# Tar input files and upload to mercury cache
printv "\nTarring input files...\n";
printv `tar cvf $job.tar $source $job.tcl`;
printv "\nUploading analysis files to mercury.uvic.ca cache...\n";
print `globus-job-run mercury.uvic.ca -l /bin/sh '-c' 'source ~gcprod01/gcsource ;
cd \$GC_CACHE ; mkdir $job'`;
chomp(my $pwd = `pwd`);
print `globus-url-copy file://$pwd/$job.tar gsiftp://mercury.uvic.ca/home1x/gcprod
/gcprod01/GC_CACHE_mercury.uvic.ca/$job/${job}in.tar`;

# Remove temporary files
printv `rm $job.tar $job.tcl`;

# Create execution script
open SCRIPT, ">scriptfile.csh";
print SCRIPT "#!/bin/tcsh -f\n";
print SCRIPT <<eot;

setenv PATH /bin:/usr/bin:.
source ~gcprod01/gcsource.csh
source /home1x/gcprod/gcprod06/babar/.babrc
setenv LD_LIBRARY_PATH `echo \$LD_LIBRARY_PATH`:/home1x/OtherMounts/hep04/gcprod/gcprod0
cd babar
newrel -t $REL $job
cp -r /home1x/gcprod/gcprod06/babar/packages/$REL/workdir $job/workdir
cp -r /home1x/gcprod/gcprod06/packages/$REL/BetaMiniUser $job/BetaMiniUser
cp -r /home1x/gcprod/gcprod06/packages/$REL/BetaPid $job/BetaPid
cd $job
srtpath $REL $ARCH
gmake installdirs
gmake workdir.setup
cd workdir
globus-url-copy gsiftp://mercury.uvic.ca/home1x/gcprod/gcprod01/GC_CACHE_mercury.uvic.ca/
```

```
$job/${job}in.tar file:///\$PWD/${job}in.tar
tar -xvf ${job}in.tar

eot
print SCRIPT "gccp -v";
foreach $kanga (@input) {
    print SCRIPT " $kanga.01.root";
}
print SCRIPT "\n";
print SCRIPT "which BetaMiniApp\n";
print SCRIPT "BetaMiniApp $job.tcl \n";
close SCRIPT;

# Submit script to mercury cluster
print `globus-job-submit mercury.uvic.ca/jobmanager-pbs -s scriptfile.csh`;

sub printv($){
    print $_[0] if (defined $options{v});
}
```

References

[1] M. Barnett, "The Atlas Experiment", [Online Document], Available at HTTP:http://atlasexperiment.org/index.html

[2] W. Tomlin, "LHC Computing Grid Project (LCG) Home Page", [Online Document], Available at HTTP:http://lcg.web.cern.ch/LCG