Uniting the Condor Batch System with the NorduGrid Middleware to Form a Powerful Tool for ATLAS and High-Throughput Computing

Haakon Riiser
University of Oslo
June 2005

Thesis presented for the Cand. Scient. degree
in Experimental Particle Physics

Abstract

Scientific progress has traditionally been driven forward by the efforts of individual experimentalists and theorists, but the highly complex challenges in today's science and increasingly powerful technology calls for new methods and tools: Computation, data analysis and collaboration has become essential in most modern sciences. But despite that computer power, data storage, and communication continue to improve exponentially, computational resources are failing to keep up with scientists' demands. The purpose of this thesis is to study and develop new systems for computation that will make better use of the resources we have. This thesis will attempt to accomplish this goal by bridging the gap between a local resource management system called *Condor* and a Grid middleware called *ARC* that was developed under the NorduGrid project. The culmination of this work is the successful application of the unified Condor/ARC system in the highly demanding ATLAS Data Challenges.

Acknowledgments

This thesis owes much to a number of people, and hopefully I will be able to mention the most significant contributors here. First and foremost, I would like to thank my supervisor, Alex Read, for having the initial ideas of what this thesis should be about, for providing invaluable testing of the software written as part of this project, for helping with maintaining the group's Condor and NorduGrid installation, and for proofreading and commenting this text. I would also like to thank the development teams behind Condor and NorduGrid for their continued support while I was writing the software to interface with their systems. Three people from the NorduGrid project deserve special mention: Aleksandr Konstantinov for providing patches that kept my software up-to-date with the latest versions of his software, for helping with debugging, and for answering a great number of questions regarding the Grid Manager interface; Balázs Kónya for going out of his way to rewrite the Information System to make it easy to integrate systems like Condor, and for creating template scripts that made writing the interface a pleasant experience; Jakob Langgaard Nielsen for tracking down a number of bugs during his valuable ATLAS Data Challenge testing. I am also very grateful to Farid Ould-Saada for providing me with books and other study materials, and a lot of his time in helping me prepare for my final exam. Finally, I would like to thank my parents and my colleagues for their patience and understanding during my protracted Cand. Scient. program.

Contents

Chapter 1

Introduction

1.1 Computing Demands of New Physics

The next generation of particle physics experiments, such as CERN's ATLAS [1] program, will accumulate an overwhelming amount of data for scientific analysis. The LHC experiments alone will generate a data stream of up to one terabit per second [2], which greatly exceeds our current capabilities to analyze, store, and transfer by network. Extensive real-time filtering will make it more manageable by reducing the data flow by several orders of magnitude, but one can still expect tens of petabytes per year to be written to permanent storage for further analysis.

This section will give a very brief overview of modern particle physics, and list some of the challenges issued by CERN's next big project, ATLAS.

1.1.1 Accelerators and Detectors

Particle physics is the study of the fundamental constituents of matter and their interactions. As experimental methods have improved, particles that were believed to be *elementary* (i.e., indivisible particles without internal structure) had to be reclassified and the theory revised.

The current theory is called the *Standard Model*, and it attempts to explain all phenomena of particle physics in terms of the properties and interactions of a small number of particles of three distinct classes: *leptons*, *quarks* and *gauge bosons*. The latter particle class is special in that it mediates the interaction of two elementary particles; the notion is that a force is felt between two particles when a gauge boson is exchanged between them.

The experimental methods used to verify the Standard Model depend

on a fundamental principle in special relativity, stating that matter can be converted to energy, and vice versa. This means that when particles moving at very high speeds collide, the loss in kinetic energy can be balanced by the creation of new particles.

Until the early 1950s, the only source of particles moving at sufficiently high energies were cosmic rays, but nowadays, beams of high energy particles are created by accelerating stable charge particles using electromagnetic fields. These devices are aptly named *particle accelerators*, and they have the obvious advantage of allowing the experimenter to control the energy and the types of particles used.

The constructed beam of accelerated particles is directed onto a target[1] so that interactions may be produced, and the particles produced in the interactions are in turn identified by observing their interactions with the material of the *detector*.

1.1.2 The Large Hadron Collider

Particle accelerators and detectors have been used to turn the Standard Model into a well-tested physics theory, that has predicted and explained a number of phenomena with amazingly high precision.

Nevertheless, there are still a number of open questions, such as why elementary particles have mass, why gravity and the strong force cannot be described in the same theoretical framework as the electromagnetic and the weak force, why antimatter is not the perfect opposite of matter, etc.

Physicists have come up with ideas for extending the Standard Model so that it can explain many of these open questions, but this requires the existence of particles for which there is no experimental evidence.

The reason why no one has yet been able to observe these hypothetical particles could be that they cannot be created at the energy levels available in current accelerators. In order to make progress, CERN (the European Organization for Nuclear Research) has retired its old *Large Electron Positron* (LEP) collider, and is building the *Large Hadron Collider* (LHC) in its place. LHC is projected to begin operating during 2007 [3], and will be able to accelerate two counter-rotating beams of protons to energies of 7 TeV, creating proton-proton collisions with a center of mass energy of 14 TeV. It will also do heavy ion (such as lead) collisions up to 1148 TeV, making it the most powerful accelerator ever built. Figure 1.1 gives a very high level illustration of the LHC layout.

[1] The target can either be stationary relative to the laboratory (a *fixed target* experiment), or another particle beam moving in the opposite direction (a *colliding beam* experiment).

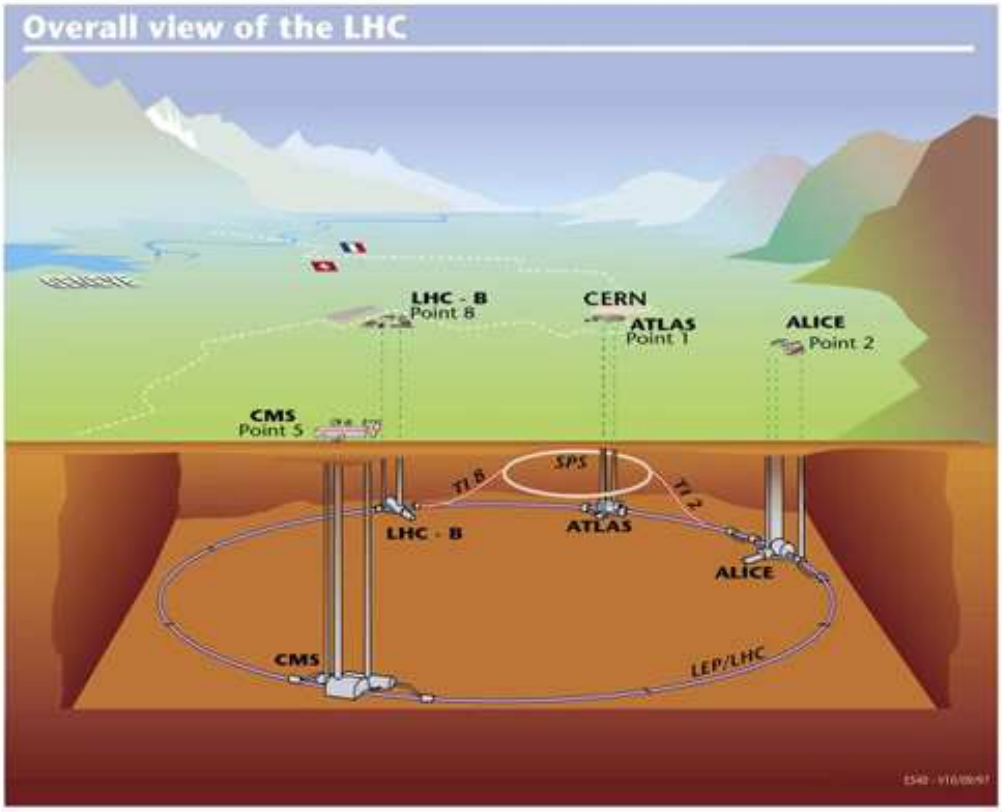

Figure 1.1: A view of the LHC

1.1.3 The ATLAS Project

The *ATLAS* (A Toroidal LHC ApparatuS) project is an international collaboration involving 34 countries, that will study high energy proton-proton collisions in the LHC.

The main goal of the experiment is to look for *Higgs* and *supersymmetric* particles, but ATLAS will also be used to determine if quarks and leptons really are elementary particles, to search for new heavy gauge bosons, and perform more precise measurements on particles and phenomena we have already seen.

The Higgs particle, if detected, will give invaluable support for the Standard Model's explanation for particle mass. The Standard Model postulates that the concept of mass is a result of the interaction between particles and a field almost indistinguishable from empty space that is called the *Higgs field*. Particles that interact strongly with this field are heavy, and particles that interact weakly are light. Associated with the Higgs field is

at least one new particle, called the Higgs particle (or Higgs boson), and it is hoped that the ATLAS detector installed in LHC will be able to detect it.

The concept of supersymmetry is particularly important in the context of *Grand Unified Theories* (which try to unify the strong and electroweak forces). The supersymmetric theories predict that every observable particle has a massive "shadow" partner. The lightest shadow particle is thought to be stable, electrically neutral, massive but weakly interacting and thus may also be the answer to the question of the nature of *dark matter*[2]. No shadow particle has yet been detected, but the new ATLAS detector will be capable of searching for a wide variety of supersymmetric particles.

The results for all of the above experiments will be hidden in the *enormous* amount of data produced by the ATLAS detector during collision events: Up to 20 simultaneous events may be "piled up" on top of each other at the same beam-crossing! Fortunately, the probability that two interesting events pile up is insignificant and most of the pile-up is easy to distinguish from the interesting signal. Computers will have to be used to filter out the signal from the noise, and figure out the most likely particle paths and decays, as well as anomalies from the expected behavior. LHC and the ATLAS experiment will put unprecedented demands on computer processing, so new tools were developed in the context of these experiments, and much relies on the existence of a functional Grid infrastructure in time for the opening of LHC.

1.1.4 The ATLAS Data Challenges

The ATLAS Data Challenges [4] were intended to be used to validate ATLAS' computing and data model, its software, and to ensure correctness of the technical choices to be made. It was originally planned for three parts (DC0 to DC2) of increasing complexity, followed by annual Data Challenges whose goals would depend on the results of the preceding ones. All but DC0 would require more processing power than CERN could provide locally, and for this reason, Data Challenges would run worldwide, making as much use of Grid tools as possible.

A description of the goals of each individual part of the Data Challenges follows.

[2]Dark matter is believed to make up the vast majority of the mass of the universe, and is called "dark" because it emits no radiation and is only inferred by its gravitational effects on visible matter.

Data Challenge 0

The primary goal of DC0 was to test the continuity of the software chain, and to verify that it was ready for Data Challenge 1. It was intended to simulate and reconstruct samples of the order of 10^5 Z + jet events, or similar. Issues to be checked included:

- GEANT3 [5] (simulation of detector response to the final-state particles).
- "Pile-up" handling.
- Reconstruction running.
- Database access (read/write).

A few samples of single particle data (electrons, muons, and photons) would also be generated to study the impact of the detector layout.

Data Challenge 1

DC1 increased the sizes of simulated event samples to 10^7, a number that was too big to be handled locally by CERN, thus creating the need to involve outside sites for extra processing power. DC1 had the following goals:

- Provide a sample in the order of 10^7 di-jet events each, for High Level Trigger [6] (HLT) studies.
- Do reconstruction and analysis on a large scale to find out where the bottlenecks are.
- Provide samples of events for physics studies.
- Utilize Grid tools to involve outside sites and to gain experience with this computing model in practice.
- Use some of the produced data to carry out evaluation of dBase technologies.

Data Challenge 2

The details of this final Data Challenge depended on the results from DC0 and DC1, but the initial goals included:

- Use the Grid test bed that would be built in the context of the LHC Computing Grid Project with the scale at a sample of the order of 10^8 events.
- GEANT4 simulation.
- Investigate generated samples for new physics.
- Extensive use of Grid middleware.
- Perform studies of the trigger system (a sequential series of hardware- and software-based event filters).

1.2 Computational Tools for Modern Science

It is clear from the above that ATLAS and future projects will require new tools for computing that encourage international collaboration.

While today's personal computers are both faster and offer more storage space than the supercomputers of the 1990s, they are not – by themselves – capable of meeting the coming demands for storage and computing power. The concept of Grid computing has emerged as a means to cope with this problem, by allowing computational tasks to be delegated via the Internet to a large number of Grid clusters scattered around the world, easily allowing off-site resources to contribute to the analysis. However, traditional Grid computing has relied on dedicated computer clusters, and this makes the threshold for participating higher than it needs to be.

With the advent of the personal computer, something almost as important as the increase in performance has happened. Traditionally, computing power was concentrated in a small number of supercomputers, and could readily be taken advantage of when needed.

Today, the situation is different: As computers became more powerful and affordable, the tremendous growth in the world's combined computing power came not only from more powerful computers, but also from an amazing growth in personal computer ownership. A natural consequence of this is that the world's computing power has become decentralized, scattered around in millions of personal computers.

The owner of each workstation has dedicated access to his own computer's resources at any time, but, unfortunately, most of the CPU cycles provided by the hordes of personal computers are wasted, either because the owner is away, or because most of the time the computer is running interactive programs where it cannot do much of anything besides waiting for user input.

The primary goal of the *Condor Project* was to develop software that would enable scientists and engineers to harness these wasted resources, while still giving the owner of each workstation its full capabilities when he is present [7].

Simply put, the Condor software manages computing resources in an effective manner – it provides job queuing mechanisms, scheduling policies, priority schemes, resource monitoring, and resource management; it is a general batch system optimized for heterogeneous clusters of non-dedicated computers. Users submit their jobs to Condor, which in turn places them into a queue, chooses when and where to run the jobs based upon a (configurable) policy, carefully monitors their progress, and ultimately informs the user upon completion.

It is clear that allowing Condor pools to act as Grid clusters is a good way to attract more computer power to the Grid. Wasted resources benefit no one, so Condor can in theory provide the means to add extra resources to the Grid at a very low cost and with relatively small effort. The Grid connection is also a clear advantage from a Condor user's point of view, since the Grid extends Condor in a natural way: Condor by itself allows for resource utilization across computer boundaries, and combined with the Grid, this is taken one step further by also spanning institutional boundaries.

The next two chapters will give a more detailed description of Condor and Grid, the latter with a focus on the NorduGrid middleware that was, for reasons that will be explained later, found to be the best candidate for bringing Condor pools to the Grid. The chapter on Condor does in particular discuss some of the potential pitfalls with respect to performance and security that a system administrator should be aware of. The last two chapters describe the creation of the software that connects Condor to NorduGrid, and experiences with the finished product under ATLAS' Data Challenge 2.

Chapter 2

The Condor Batch System

2.1 Exceptional Features

For users with access to computer parks where most machines are connected by network filesystems and shared user databases, a simple and unsophisticated batch system might not provide the incentive required to make them want to learn and use a new system. After all, users privileged with such an environment can simply log into any personal workstation remotely and start their jobs by hand. This scheme burdens the user with a lot of manual labor, but it works and gives him more direct control of his job, which is something many people are very reluctant to give up.

Fortunately for site administrators who would like to deploy a batch system in their computer park, Condor provides benefits beyond the traditional set of features, that might prove useful enough to sway the more conservative and skeptical users. The most attractive feature is probably the checkpointing mechanism, which is an effective safeguard against many types of problems.

For computing jobs that last weeks or even months, a sudden power outage or a system crash would be catastrophic. The owner of the job may have no choice but to start the job from scratch and hope that whatever caused it to terminate prematurely will not happen again.

Condor's checkpointing mechanism provides protection against this scenario; by backing up the job's memory image at regular intervals to a checkpoint server (either a machine dedicated for this purpose or the submit machine), the person who submitted the job is assured that the job *will* finish, even in an unstable environment.

If a machine goes down or becomes unavailable, the last checkpoint image will be sent to an available execute machine that matches the job's

requirements, and the job can continue running from the state of the last image.

Unfortunately, checkpointing comes at a cost: It is only available to a certain class of Condor jobs (so-called "standard universe" jobs, as will be discussed later in this section), and there are many restrictions on jobs which Condor has been asked to transparently checkpoint. Among the things not allowed are: multiprocess jobs, interprocess communication, alarms, timers, sleeping, memory mapped files, opening files for both reading and writing at the same time, and dynamically linked executables[1].

The same class of Condor jobs can also benefit from *remote system calls*, which means that system calls[2] accessing the file system are forwarded from where the job is running to where the job was submitted. Consequently, one does not need to worry about the data requirements of a submitted job: Data that is read or written by such a Condor job is continually transferred between the submit and the execute machine, so that from the job's point of view, it is running on the submit machine. As long as the data requirements are satisfied on the submit machine, they are guaranteed to be satisfied on *any* machine in the pool with compatible hardware.

Condor offers several execution environments, commonly referred to as its "universes": *vanilla*, *standard*, *pvm*, *scheduler*, *globus*, *mpi*, and *java*; of these, the most important are *vanilla* and *standard*.

The *vanilla* universe is for any regular executable and script that one could run from the command line. The vanilla universe places no restrictions on what the job is allowed to do, but neither does it allow for checkpointing and remote system calls.

The *standard* universe is for executables that have been re-linked to the Condor libraries using `condor_compile`, which is a special program included with Condor that wraps around and alters the behavior of the normal command used to link executables. The standard universe is the *only* universe that enables the aforementioned checkpointing mechanism and remote system calls.

The remaining universes are less commonly used: *scheduler* is for jobs acting as metaschedulers[3] and *java* provides a way to run Java programs without having to use the vanilla universe, which would have excluded

[1]The restriction on dynamically linked executables applies only on the Digital Unix (OSF/1), HP-UX, and Linux operating systems.

[2]A system call is the mechanism used by an application program to request a service from the operating system kernel.

[3]A metascheduler is a program that manages dependencies between jobs at a higher level than the Condor Scheduler.

available computers of otherwise incompatible architectures (Java programs run on a virtual machine, so the actual underlying hardware does not matter).

Finally, Condor also provides an outgoing interface to PVM, MPI, and Grid computing through the *pvm*, *mpi*, and *globus* universes, thus providing a consistent framework around these environments, making them more accessible to users already familiar with Condor's job submission interface. A short description of these universes is given in Sections 2.7.1 (PVM), 2.7.2 (MPI) and 3.2 (Condor-G/Globus).

However, as these features are optional, they will not be available in every Condor installation. In fact, the only universe that is guaranteed to be available is vanilla, and the standard universe is available on operating systems that support it.

2.2 The Topology of a Condor Pool

Perhaps the most important part of installing Condor is designing the topology of the pool. A node in Condor can assume any combination of the following four roles: *central manager*, *execute*, *submit*, and *checkpoint server*.

The role of highest importance is that of the central manager, which is responsible for collecting information, and for negotiating between resource providers and resource requesters. If the central manager is down, no new jobs can be submitted, and jobs cannot migrate to a different execute machine. It is therefore crucial that a reliable machine with a good network connection to all machines in the pool is chosen for this role.

An execute machine is one that is configured to accept and run Condor jobs. Any machine in the pool, including the central manager, can assume this role. The same goes for the submit machines, which simply are computers from which jobs can be queued.

Perhaps surprisingly, the resource requirements for a submit machine can be greater than the requirements for an execute machine. For every active standard universe job, there must be a so-called *shadow process* running on the machine from which it was queued (the shadow process is there to handle forwarded system calls). Furthermore, unless a dedicated checkpoint server is used, the memory checkpoints are stored on the submit machine, so some disk space is also needed for this (exactly how much will of course depend on the job's memory footprint). Since there is no limit on the number of jobs that can be submitted from a single machine, it is clear that a submit machine's resources can easily be depleted.

The fourth and final role of a Condor machine is the dedicated checkpoint server. This role is optional; if configured, the chosen machine will store checkpoint files for every job in the pool. It follows that the most important properties of the checkpoint server is a high bandwidth network connection and lots of disk space.

2.3 Installing Condor on Linux

Although Condor is a portable system that runs on many different operating systems, it is probably the Linux version that is most frequently used. This, and the fact that only the Linux version has been used in this thesis, justifies a short discussion on the specific problems encountered while installing and setting up Condor on this operating system.

The first thing to note when preparing a Linux machine for Condor is the specific Linux distribution installed. Condor is currently only supported on a select group of versions of the "Red Hat" and "SuSe Enterprise Server" Linux distributions. Distributions and versions not listed may still work, but are obviously less tested (if at all).

A potential problem that will be immediately apparent for all unsupported distributions is that it can be difficult to prepare programs for the standard universe using `condor_compile` with other compilers than the one included with the distribution on which your Condor package was designed to run.

On earlier versions of Condor, `condor_compile` required GCC (GNU Compiler Collection) version 2.96, which is an unofficial version that only existed on older Red Hat systems. If you did not have this particular version of GCC, there were two possible solutions:

1. Compile and link the programs on a machine running a supported version of Red Hat; needless to say, this would be terribly inconvenient.

2. Install Red Hat's compiler on your own system. This alternative is probably the best, but it's not a trivial operation: The infamous GCC version 2.96 was broken in many ways, and should *not* overwrite or interfere with the standard system compiler.

Another potential source of problems for local area networks where IP addresses are assigned dynamically[4] is the hostname $\mapsto$ IP address map-

[4]This was not a problem in our group's local Condor pool, as all machines used in our pool have static IP addresses and hostnames that are registered in DNS.

ping for any machine in the Condor pool. The Condor system will not work properly if the hostname of a machine in the pool is mapped directly to to the loopback[5] address in `/etc/hosts` (the local address database on Unix systems), which happens to be the default setting on Red Hat.

Unfortunately, this situation is not likely to change, since mapping the machine's name to the loopback address is a reasonable configuration when the real IP address is dynamically assigned by a DHCP[6] server. This configuration ensures that, no matter what IP address the machine gets from DHCP, the internal hostname to IP address lookup from `/etc/hosts` will always refer to the right machine.

There are two ways to deal with this: One way is to hard-code the current IP address in `/etc/hosts` and hope that it will not change in the near future. This is not good enough in general, so a different method is required. Depending on the DHCP client program used, it may be possible to have `/etc/hosts` updated automatically by the DHCP client program every time the IP address changes.

The following will describe how to accomplish this using a DHCP client that makes it possible to run a customized program every time the IP address changes. In a typical scenario, this feature would just be used to log the event, but in this case, the program must update the IP address for the local machine in `/etc/hosts`.

The program that updates `/etc/hosts` can be a simple shell script – an example of such is given in the Appendix, Section A.1. What it does is update the IP address of the line in `/etc/hosts` containing the machine's fully qualified hostname with the IP address of the network interface specified as its first argument (typically `eth0`, which is the first Ethernet interface on a Linux system).

Next, the DHCP client needs to be instructed to execute the program whenever the IP address changes. How to do this will, of course, depend on the DHCP client. For recent versions of `dhcpcd` [8], which is one of the more common DHCP clients on Linux systems, this is accomplished by putting the command to be executed in `/etc/dhcpc/dhcpcd.exe`.

[5]The loopback address is a special address (127.0.0.1) that is used when the computer needs to connect to itself.

[6]Dynamic Host Configuration Protocol: A protocol that provides a means to dynamically allocate IP addresses to computers on a local area network.

2.4 Preparing a Program for the Standard Universe

Condor was designed to be able to run most existing computational jobs without much effort on the programmer's part. For vanilla jobs, nothing needs to be done, except making the job's data files available on the execute node, either manually or by telling Condor which files need to be transferred during job submission. This section only applies to jobs that are built for the standard universe.

To prepare an executable for the standard universe, it will often be enough to re-link the program with `condor_compile`. Specifically, this means that one takes the full command line normally used to create the program executable, and then sticks the text "`condor_compile`" in front of it. This ensures that all the appropriate Condor libraries are linked in.

As mentioned in Section 2.1, there are restrictions on the behavior of a checkpoint enabled job, and some modifications in the code will be necessary if the application does not adhere to the limitations [9] mentioned in the Condor manual. There could also be good reasons for rewriting the program to improve efficiency under Condor. See Section 2.5 for more on that – this section will only discuss the changes that are *required* because of limitations in Condor.

The main issue when preparing applications for Condor will usually be I/O (Input/Output). Network I/O is special, and the specific limitations are rather vague: Unless network access is "brief", checkpointing and job migration will be delayed.

Regular disk I/O will usually work fine, but there can be surprises. For example, if your program uses compressed data, be sure that it does not use external applications to handle the decompression. Neither multiprocessing nor interprocess communications are allowed, so unless the compressed data is not decompressed inside the job itself, it cannot be used at all.

One should also be aware of the fact that the semantics of system calls inside Condor's standard universe is not necessarily the same as it is outside. It is possible for system calls to misbehave because of bugs in the remote system call mechanism, and this can, obviously, produce quite unexpected results.

For example, the system call `mkstemp` will normally create a unique temporary file, and return the newly allocated file descriptor along with the associated filename. In Condor version 6.1.10, no file is created; the filename is generated, but the file itself is not. Instead, Condor's `mkstemp`

returns the file descriptor for standard output and, consequently, all output destined for the temporary file is merged with the standard output stream. Moreover, when closing the file descriptor returned from `mkstemp`, it is actually standard output that is closed, and all further output is inexplicably lost. Since a valid file descriptor is returned, the program cannot know that `mkstemp` is misbehaving. Note that this particular bug has since been fixed; the problem no longer exists in version 6.6.3.

Obviously, the exact semantics of misbehaving system calls can readily change from one version of Condor to another, so before starting a long-term job under an untested version, one should preferably write and run a simple test suite for all "unusual" system calls used by the job. The source code used for verifying `mkstemp` can be seen in the Appendix, Section A.2. Running this program under Condor 6.1.10 makes the "*hello, world*" text go to standard output instead of the temporary file.

2.5 Optimizing for the Standard Universe

Although Condor is designed to provide an environment for high-*throughput* computing as opposed to high-*performance* computing, using the system would be far less desirable if it meant that each individual job would finish much later than the same job running as a normal process.

This section discusses Condor's impact on the efficiency of a single standard universe process, and how to work around some of its inherent pitfalls. This chapter does not apply to programs running in the vanilla universe, since they are completely ordinary processes, and get neither the safety benefits nor the performance and resource penalties that a standard universe job would get.

All benchmarks have been run on the same system; an Intel Celeron 800 MHz with 256 MB RAM. In the case where a job is submitted to Condor as a standard universe job, both the execute and the submit machine had the same CPU and memory size. The benchmarks were run on Condor version 6.1.10.

Knowing that system calls are forwarded to the submitting host, one would expect that programs involving lots of I/O would be degraded most in performance, while CPU/memory intensive applications should run with little (if any) slowdown.

The CPU and memory performance impact was measured using the Linux version [10] of BYTE Magazine's "BYTEmark" [11] benchmark suite (Nbench, release 2) which is specifically designed to expose the capabilities of the CPU, FPU and memory system.

The benchmark was first run nine times as a normal process (Table 2.1), followed by nine runs as a standard Condor job (Table 2.2). The performance figures in these tables are the performances relative to an AMD K6 233 MHz baseline system. Table 2.3 compares the benchmark's performance as a standard Condor process to a normal process by presenting the average over the nine runs in column one and two, and the ratio of columns one and two in column three.

Nbench test	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**avg.**
Numeric sort	3.12	3.06	3.06	3.07	3.06	3.12	3.13	3.13	3.11	3.10
String sort	2.42	2.24	2.23	2.24	2.25	2.24	2.24	2.24	2.24	2.26
Bitfield	4.19	4.18	4.19	4.19	4.19	4.18	4.19	4.18	4.17	4.18
FP emulation	1.98	1.97	1.97	1.97	1.97	1.97	1.97	1.97	1.97	1.97
Fourier	4.59	4.59	4.59	4.59	4.59	4.58	4.59	4.59	4.59	4.59
Assignment	5.12	5.41	5.42	4.91	5.10	5.40	5.40	5.41	5.34	5.28
IDEA	2.99	2.98	2.98	2.98	2.98	2.98	2.98	2.98	2.98	2.98
Huffman	3.03	3.02	3.02	3.02	3.02	3.02	3.02	3.02	3.02	3.02
Neural net	4.45	4.45	4.45	4.45	4.45	4.45	4.45	4.46	4.44	4.45
LU decomposition	6.87	7.87	8.26	7.03	7.91	7.55	7.69	7.98	7.07	7.58
Memory (Linux)	3.734	3.697	3.711	3.589	3.634	3.695	3.701	3.700	3.684	3.683
Integer (Linux)	2.732	2.716	2.716	2.716	2.716	2.727	2.730	2.730	2.726	2.723
FP (Linux)	5.198	5.438	5.524	5.234	5.448	5.359	5.396	5.463	5.240	5.387

Table 2.1: Nbench as a normal process on an Intel Celeron 800 MHz

Nbench test	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**avg.**
Numeric sort	3.10	3.06	3.10	3.12	3.12	3.16	3.11	3.02	3.11	3.10
String sort	2.25	2.24	2.25	2.24	2.24	2.24	2.24	2.25	2.24	2.24
Bitfield	4.20	4.20	4.20	4.19	4.21	4.20	4.16	4.19	4.20	4.19
FP emulation	1.95	1.96	1.96	1.95	1.96	1.96	1.95	1.95	1.95	1.95
Fourier	4.29	4.29	4.28	4.29	4.29	4.29	4.29	4.27	4.29	4.29
Assignment	4.94	5.41	5.42	5.41	5.42	5.10	5.40	5.30	5.41	5.31
IDEA	2.97	2.98	2.97	2.98	2.98	2.98	2.97	2.98	2.97	2.98
Huffman	3.00	3.00	3.00	3.00	3.00	3.00	3.00	2.99	3.00	3.00
Neural net	4.36	4.36	4.36	4.36	4.36	4.36	4.37	4.37	4.37	4.36
LU decomposition	7.81	8.00	7.37	7.35	7.62	8.10	8.11	5.57	7.83	7.53
Memory (Linux)	3.601	3.709	3.712	3.707	3.711	3.636	3.695	3.680	3.708	3.684
Integer (Linux)	2.711	2.703	2.712	2.716	2.716	2.725	2.713	2.693	2.714	2.711
FP (Linux)	5.267	5.310	5.165	5.163	5.225	5.334	5.334	4.701	5.276	5.197

Table 2.2: Nbench as a standard Condor process on an Intel Celeron 800 MHz

By calculating the arithmetic mean of the ratios in Table 2.3's final column, we have an average performance penalty of approximately one per-

Nbench test	normal avg.	standard Condor avg.	normal/standard Condor
Numeric sort	3.10	3.10	1.000
String sort	2.26	2.24	1.009
Bitfield	4.18	4.19	0.998
FP emulation	1.97	1.95	1.010
Fourier	4.59	4.29	1.070
Assignment	5.28	5.31	0.994
IDEA	2.98	2.98	1.000
Huffman	3.02	3.00	1.007
Neural net	4.45	4.36	1.021
LU decomposition	7.58	7.53	1.007
Memory (Linux)	3.683	3.684	1.000
Integer (Linux)	2.723	2.711	1.004
FP (Linux)	5.387	5.197	1.037

Table 2.3: A comparison of Nbench run as a standard Condor process and a normal process based on the averages over the nine runs in tables 2.1 and 2.2

cent. This confirms the expected results: One need not hesitate to use Condor's standard universe for CPU, FPU or memory intensive jobs, and there is no need to do any Condor specific optimizations.

I/O is an entirely different matter: Since every interaction with the file system must be forwarded to the submitting machine, one can expect a lower bound on the system call latency to be at least equal to the network's round-trip time (as measured with the standard `ping` program). Based on this fact, one would expect the I/O performance of a standard universe job to degrade faster than a regular process on decreasing block sizes.

The latency of forwarded system calls was first examined by running a benchmark program that `stats` the current directory one million times. `stat` is a system call that returns status information on an entry in a Unix file system, and must therefore be forwarded to the originating machine; `stat` is suitable for this test because the amount of data transferred is small enough to fit in a single network packet, and should therefore accurately measure the minimum latency of forwarded system calls. See Section A.4 in the Appendix for the source code.

The test revealed that one million `stats` on the current working directory as a normal process took 1.4×10^{-6} s per call. When forwarded by Condor's remote system call mechanism, the same job was timed at 3.7×10^{-4} s per call. Considering that the current round-trip time of the network was 2.0×10^{-4} s, one can see that there is some additional overhead in the Condor communication protocol.

Next, a synthetic benchmark was written to measure the performance impact on data transfers. The program basically writes a predetermined amount of random data (randomized so that possible network compression would not influence the results) directly to disk, and calculates the rate at which data is written – see Section A.3 in the Appendix for the source code. The results are presented in Table 2.4.

Block size (bytes)	normal (kB/s)	standard Condor (kB/s)	normal/ standard Condor
64	7666.34	141.332	54.2
128	11357.5	252.225	45.0
256	18657.8	444.575	42.0
512	22619.8	703.32	32.2
1024	26369.2	951.972	27.7
2048	28445.1	1493.56	19.0
4096	30926.5	47.651	649.0
8192	30600.4	2777.6	11.0
16384	30849.9	5542.3	5.6
32768	32062.5	6964.04	4.6
65536	27284.2	7241.96	3.8

Table 2.4: Results from I/O benchmark

The program revealed that Condor is *extremely* sensitive to the I/O block size. This is in itself is not surprising, since two times the round-trip time is added to each I/O operation; one would expect the performance to increase strictly (but asymptomatically approaching a maximum) when the block size is increased. This assumption turned out to be correct, except for an anomaly at a particular block size, which turned out to be *terrible* for performance.

As can be seen in Table 2.4, the relative difference in transfer rate between a normal process and a Condor process diminishes as the block size increases, except for one particular block size: 4096 bytes. For some reason, using a block size of 4096 bytes reduced the I/O transfer rate by a factor of over 12 compared to a block size of merely 64 bytes!

Considering that 4096 bytes is one of the most commonly used block sizes (it's a typical buffer size in standard ANSI C file stream library functions such as `fread`, `fwrite`, etc.), it would be wise to analyze every job that does non-trivial I/O and make sure that the block size is efficient.

2.6 Security Concerns in Condor

True Condor programs, in other words those linked for the standard universe, should not be any additional reason for concern. All allowed system calls that interact with the filesystem are intercepted and forwarded back to the submit machine, so this does not make it easier for the job owner to abuse machines he normally wouldn't have any access to. Illegal system calls should cause the process to terminate, which means that they also pose no additional threat (unless Condor failed to intercept the system call, which is supposedly not possible).

The primary security concern with Condor is that it is susceptible to denial of service attacks from vanilla programs – again, these are normal programs that have not been re-linked for Condor, and thus their system calls are not forwarded back to the originating machine. If the execute and the submit nodes are in the same user ID domain, vanilla jobs run as the user that owns the job; otherwise they run as the Unix user `nobody`.

Allowing vanilla jobs means that a malicious user could do things like filling up world writable storage areas (such as `/tmp`), or bringing the system to its knees by creating more processes than it can handle (an attack known on Unix systems as *fork death*, where one repeatedly calls the system call `fork` to create new processes until the system becomes so unresponsive that the offending process can not be terminated, forcing a hard restart).

Another problem is that, although Condor tries to clean up after a vanilla job finishes, a malicious user could create child processes that are left behind. If these lurker processes run as the user `nobody`, they could easily attack the next job that comes in under the anonymous `nobody` user. The only way Condor could be sure to terminate all child processes would be to kill *all* processes owned by `nobody`, but this is not a viable solution: `nobody` is used for many non-privileged Unix daemons, and Condor would have killed those as well. Fortunately, it is possible to instruct Condor to use a user ID dedicated to Condor for anonymous jobs, making it safe to kill all remaining processes owned by the dedicated user on job completion, thus closing the security hole. Note that this implies that multiprocessor systems need one dedicated user ID per CPU.

Finally, it is also worth mentioning that vanilla jobs pose yet another minor threat in that they can access all world readable files. For environments where security is a high concern, the Condor manual recommends using IP-based security mechanisms to restrict Condor access to machines where the `root` user can be trusted. Condor also provides support for

strong authentication, encryption, integrity assurance and authorization, allowing site administrators to tighten security even further. Most of these features can be enabled without affecting legitimate users.

See the proper section [12] in the Condor Manual for more detailed information on securing Condor.

2.7 Parallel Programming for Condor

Both NorduGrid and Condor can be made to take advantage of parallel programming, and the following two subsections details Condor's two supported parallel programming paradigms and the restrictions they impose on a Condor pool. As will be explained in Section 4.1, these restrictions make parallel programming impractical in combination with the NorduGrid middleware.

2.7.1 PVM

PVM [13] stands for *Parallel Virtual Machine*, and is a software system designed to run parallel jobs on heterogeneous computer clusters. Applications written for this system can use Condor's PVM universe.

PVM allows programs to run simultaneously on multiple machines, while communicating with each other via an interface provided by PVM. By assuming the role of the resource manager for the PVM daemon, Condor makes it possible to use this system on non-dedicated machines by dynamically constructing virtual machines from machines picked out of the Condor pool.

Not all parallel processing paradigms would fit in Condor's volatile machine environment where nodes may come and go, so the *master-worker* model was chosen. This model is based on one node – the submit node – acting as a controlling master for the parallel application, sending pieces of work to execute nodes as they become available. If a worker node vacates a running job, the master can simply request a replacement node. The number of workers is unimportant and changes in the virtual machine size can be dealt with easily. For these reasons, the master-worker model works very well with Condor.

Condor-PVM involves no changes to the PVM interface (neither the programming nor the binary interface), so some master-worker PVM applications can be used unchanged under Condor-PVM; not even re-linking is necessary!

Still, when comparing PVM to Condor-PVM, there are a few changes and new features. In particular, there is the concept of the "machine class": Machines of different architectures belong to different classes. The machine class concept is used in the job submission file, where it is specified how many nodes should be allocated in each class. This change is also the only thing thing in Condor-PVM that makes the job submission script different from standard and vanilla scripts. Again, refer to the appropriate section of the Condor manual [14] for the full details.

2.7.2 MPI

The *message passing paradigm* [15] is most commonly used on parallel machines with distributed memory. It provides an environment under which parallel programs may synchronize, by providing communication support.

Over the years, different vendors developed their own message passing systems, so to facilitate ease of use and portability, a standardization effort was made. The result of this standardization was the *Message Passing Interface* (MPI) standard [16].

Programs using MPI can be further simplified with Condor's MPI interface [17]. Condor dedicates machines for running the programs, and it does so using the same interface used when submitting non-MPI jobs. Except for the `machine_count` attribute that specifies the number of requested machines, the submission script looks exactly like that of a normal vanilla or standard job.

On the administration side, Condor-MPI requires special care. The MPI system requires dedicated execute machines, and a dedicated MPI scheduling server. These machines cannot be regular desktop systems, since programs running in the MPI universe cannot be moved to a different computer or suspended.

This means that once these MPI computers start running a program, they will continue running it until it completes. From the user's perspective, the biggest inconvenience related to running MPI jobs under Condor versus normal jobs is that it requires all jobs to be submitted from the dedicated MPI scheduling server.

But despite this, using Condor for MPI jobs can be useful since the procedures for submitting programs are almost identical to normal Condor jobs. A programmer who knows the programming interface for MPICH [18] version ch_p4 and the normal procedure for submitting Condor jobs needs no more knowledge to start using the system.

Chapter 3

Grid Computing

3.1 An Overview of Grid Computing

As mentioned in Chapter 1, the primary goal of Condor was to harness the lost computing power resulting from the tremendous growth in personal computer ownership. With the features discussed so far, Condor would be partly successful in solving this problem; by making it possible for a single institution to pool together its computing power, a local user will be able to tap into its resources and use as much as he currently needs, or as much as he is permitted to use.

Unfortunately, the original problem still remains, but in a larger scale: Although Condor provides the means to aggregate the resources of numerous individual computers into a single pool per institution, the pools themselves remain disconnected, just as individual computers were without Condor. The consequence is the same; some pools might at times not be able to satisfy the demands of its users, while others might waste a lot of computing cycles.

Grid computing emerged from the desire to close the gap between site-wide clusters such as Condor, making computing power available anywhere, independent of physical location. The name "Grid" comes from the analogy to the power grid, where one does not need to know where the electrical power comes from, only that it is readily available in the outlet.

3.2 Using Condor for Grid Computing

Condor has been extended to support Grid computing in two ways: *flocking* and *Condor-G*.

Flocking is the simplest extension: If enabled, and a queued job cannot be immediately started, Condor can move the job to a different Condor pool. This provides a Grid solution in the sense that jobs are no longer restricted to a single pool, but there are limitations to this technology. First of all, there is no support for interoperability with other batch systems: Flocking only works between Condor systems. Also, there is little support for authentication, so one would typically only connect to "trusted" Condor pools (i.e., pools where every local user is trusted by default). Both of these points make it difficult for a Grid built on Condor's flocking technology to grow beyond a certain size.

Condor-G is a more ambitious piece of software: Grid applications often involve large amounts of data and/or computing and often require secure resource sharing across organizational boundaries, making it difficult to handle with today's Internet and web infrastructures. Use of Grid computing is not yet commonplace, mostly because of the various problems related to managing jobs over a large global network.

Condor-G makes it possible to submit jobs from Condor to widely distributed Grid resources, using the *Globus* [19] toolkit. At first glance, it is not apparent why Condor-G is needed; obviously, it is possible to submit Grid jobs without it. But performing computations on resources that belong to different sites can be difficult in practice. Because of the heterogeneity that is possible in a Grid network, there will likely be different methods for authentication and authorization, different hardware and software, and the user will in general have no easy way to obtain the necessary information of his execution environment. On top of this, there are the obvious issues with security, failure handling and job status. What Condor-G brings is, among other things, a convenient interface to job monitoring, failure and completion notification, and automatic renewal of your Globus credentials, should they expire during operation.

Handling all of the above issues manually could make Grid computing impractical and more trouble than it was worth, so Condor-G was created in an effort to relieve the user from all of these problems by providing a complete framework that allows the user to harness multi-domain resources as if they all belong to one personal domain. Everything is done using the same Condor tools that are used for managing local Condor jobs. Also, Condor-G is a fault-tolerant system; no functionality is lost if the

submit machine crashes while a job is running.

The above mentioned practical problems with Grid computing can be grouped into three distinct problems: *remote resource access*, *computation management*, and *remote execution environment*.

Condor-G solves the issue of remote resource access by requiring that remote resources can be interfaced with standard protocols for resource discovery and management. The protocols used are defined by the Globus toolkit, a de facto standard for Grid computing [20]. The computation management and remote execution environment components can be taken from the Condor system (or any other resource management system supported by the Grid middleware).

3.3 Grid to Condor Job Submission

As described in the previous section, Condor-G makes it possible to submit Grid jobs with the standard Condor interface, but it does not provide the means for Condor pools to receive jobs from the Grid.

There are several different types of Grid middleware, and a Condor interface has to be written for each one that needs to be able to submit jobs to Condor.

As the main part of this thesis, such an interface has been written for the *NorduGrid* middleware, also known as the *Advanced Resource Connector* (ARC) [21]. This interface will from now on be referred to as the *ARC* $\rightarrow$ *Condor* interface, where the arrow indicates the direction in which jobs travel.

3.4 The Advanced Resource Connector

ARC is a GPL licensed open source Grid solution, developed under the NorduGrid project. Built atop a slightly modified version of the Globus toolkit, it provides all the basic Grid services, such as authentication, job submission, monitoring, etc. The software is mature, and is already in use in several Grid projects: NorGrid [22], Swegrid [23], DCGC [24], NDGF [25] and, of course, by the NorduGrid project [26] itself.

The NorduGrid middleware was selected as a suitable candidate for development of a Condor interface for several reasons:

- It has an active and open development process.
- Its design is modular and encourages third party extensions.

- It is used as the primary Grid tool for Northern-Europe in ATLAS DC2.

Although ARC is primarily used with variants[1] of *PBS* [27] (Portable Batch System), its developers had anticipated the need to support other batch systems.

It is obviously wise to build Grid middleware that is not too closely modeled around a specific batch system, since several systems exist that cater to different requirements and types of clusters. For example, Condor is especially designed to operate on a non-dedicated heterogeneous cluster where nodes can easily (and safely) be added or removed, while PBS works best in a traditional environment with dedicated nodes. Batch system independence makes it possible to build bridges between any local resource management system and the Grid middleware. This makes it much easier for the Grid to grow, and it also fits very well with the analogy to the electrical grid, where it is completely irrelevant what provides the power and where it comes from; the only thing that matters is that it is aggregated and available in the outlet.

[1] PBS Pro, OpenPBS and Torque (formerly ScalablePBS).

Chapter 4

The ARC → Condor Interface

4.1 Initial Goals

The design goal of highest importance in the ARC → Condor interface was that a Condor pool would not inconvenience the ARC user in any way. In particular, a user submitting an ARC job should never need to rewrite a job description that would have succeeded on PBS.

A typical cluster topology is illustrated in Figure 4.1. It shows a Condor pool consisting of a number of submit/execute nodes (drawn with dashed lines, to signify that they are non-essential[1] for basic functionality), a central manager for the Condor pool, and the Grid Manager which is also configured as a Condor submit node. The arrowheads indicate the direction in which jobs can flow, showing that external Grid jobs enter the Condor pool via the Grid Manager, which acts as a single entry point for all Grid jobs.

This cluster organization is ideal for vanilla jobs, which is the only required universe for Grid jobs. PVM, MPI, Condor's standard universe, and other special Condor features could naturally be supported in the future but it will probably never happen.

PVM would have been a good candidate for future extensions to the ARC → Condor interface since it does not require a dedicated computer pool, but it is not currently supported by ARC. MPI is supported by ARC but, unfortunately, it *does* require dedicated computers and thus it is unusable on most Condor installations.

[1] Both the Grid Manager and the Condor central manager can assume the roles of execute machines, eliminating the need for additional execute nodes, but this is obviously not recommended for a production system, since both machines are already under considerable load.

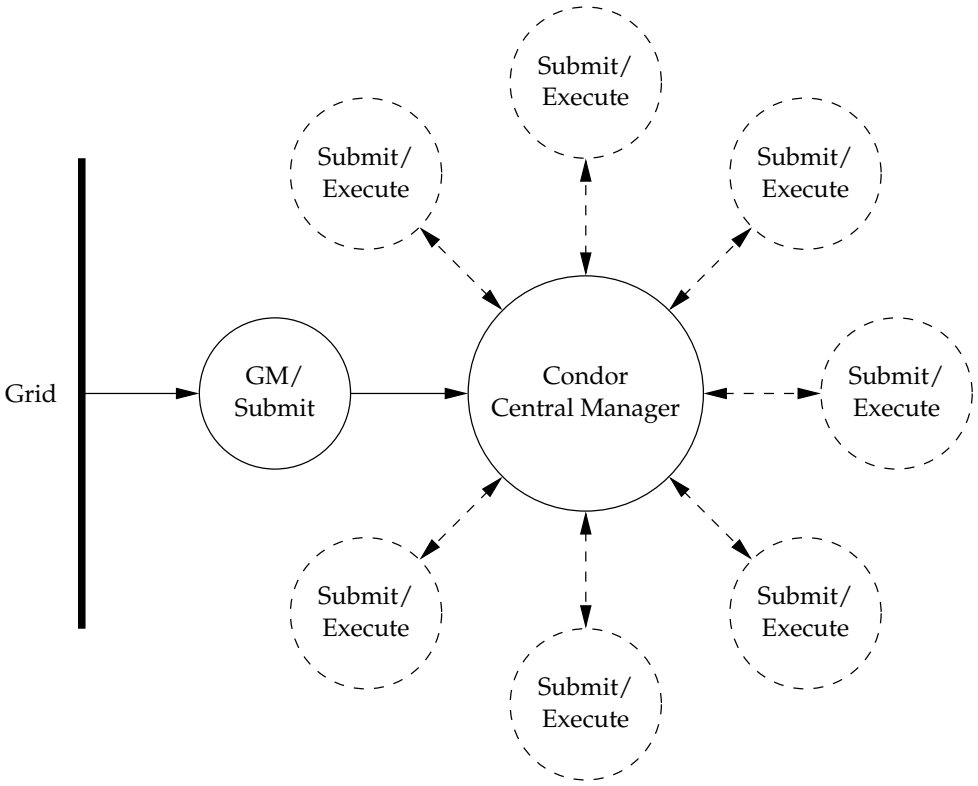

Figure 4.1: A typical ARC/Condor cluster configuration

Finally, the most important of the unsupported universes, the standard universe, could *never* be efficiently supported, as a result of the architectural design illustrated in Figure 4.1. The fact that all Grid jobs have a single entry point is what precludes Grid jobs from Condor's standard universe; as previously mentioned, standard universe jobs forward their system calls back to the submit machine; this is usually a good thing for normal interactive use of Condor, since different people will usually submit their jobs from different machines, effectively distributing the load among them. In the ARC → Condor case, *all* jobs are submitted from the Grid Manager, creating a huge bottleneck for I/O intensive jobs, and to top it off, there isn't even the normal benefit of system call forwarding (since all the data used by the job had to be uploaded to the Grid Manager in the first place, it is easy to simply pass this along to the execute node, using Condor's file transfer mechanism for vanilla jobs).

Support for the standard universe would also require the user to re-link his program using Condor's tools, and to ensure compatibility, this should be done using tools of the same version as that running on the remote

cluster. Alternatively, the system could be written so that the user submits an archive of his unlinked object code, allowing the link operation to be performed on the cluster that will run the job. Either way, it is complicated and clumsy and it makes support for the standard universe uninteresting, even when disregarding the performance penalties mentioned above.

In summary, the normal benefits of the standard universe do not apply in this special application, and would in fact only do harm. For all of these reasons, vanilla will likely remain the only supported universe.

4.2 An Implementation Overview

The ARC → Condor software consists of two distinct interfaces: A Grid Manager interface that deals with the actual job control (submission, cancellation, and finalization), and an interface to ARC's Information System, which provides detailed information on the cluster, its queues, and its jobs. The user will typically access this information via the *Grid Monitor* web interface.

Much of the work required for writing such an interface has already been taken care of by the NorduGrid developers: All code that would be common to the various batch system interfaces is already in place, and can easily be used when developing the batch system side of the interface.

For example, the first task undertaken on the ARC → Condor project was writing a parser[2] for xRSL, the language used to submit jobs in ARC. This proved difficult, because the xRSL grammar is not context free, and the most common parser generators are designed for grammars where the meaning of a symbol does not depend on its context. A solution to this problem was found by altering the grammar to make whitespace an active symbol, but unfortunately, this turned out to be highly complicated, very prone to errors, and hard to extend.

Luckily, it was soon discovered that an xRSL parser did not need to be written at all. Since this is something every batch system interface would require, the NorduGrid team had wisely made its own parser do all the work: The ARC software converts the xRSL job description to a simple file (called the *GRAMI* file) containing a flat list of variable name/value pairs, which is trivial to parse. By discarding the full xRSL parser in favor of simple expressions for extracting the name/value pairs, the code was greatly simplified.

[2] A program that determines the meaning of a sentence or string of symbols in a language.

4.3 The Grid Manager Interface

The programs that had to be written for Grid Manager communication were: `submit-condor-job`, `finish-condor-job`, and `cancel-condor-job`. Most of the information required to write this part was found in the Grid Manager Administrator's Manual [28].

4.3.1 `submit-condor-job`

When a job is first submitted, and is ready to be passed on to the local batch system, the program `submit-condor-job` is started. The Grid Manager will run this program with exactly one argument: the file containing the list of name/value pairs that was generated from the job description written in the xRSL language.

`submit-condor-job` starts by loading the name/value pairs into memory. Next, a shell script wrapper is created for starting the actual job executable. In earlier versions of the ARC → Condor interface, there was no wrapper script, and Condor was instructed to start the executable directly. There were several problems with this approach:

- Condor's job description language doesn't provide the means to embed whitespace or zero length strings in the job's command line arguments, both of which are required to be compatible with ARC and PBS.
- ARC states that, if the job specifies a runtime environment (such as ATLAS), it must run scripts associated with the runtime environment(s) before and after running the job, on the execute machine. This is also impossible to do directly from the Condor job description.
- xRSL has a `join` attribute that instructs the batch system to merge its standard output and error channels. Condor's job description language has no support for this.
- Errors in the list of output files causes Condor to re-run the job forever. By omitting the list of output files, a mistyped output file in the xRSL will never cause the job to be stuck in an infinite loop, but this makes Condor transfer *all* files created by the job back to the Grid Manager, regardless of whether they are required or not, wasting both network bandwidth and storage space on the Grid Manager.

These four points show that a more flexible tool than Condor's job submission language is required, so `submit-condor-job` was extended to create a wrapper script capable of handling them.

For the sake of debugging, the entire wrapper script is included in the Grid Manager log (this also applies to most output produced by the ARC → Condor interface; it is always a good idea to ask for the Grid Manager log by setting the xRSL attribute *gmlog* when submitting ARC jobs, as this returns enough information to determine the cause of most errors).

Following the creation of the wrapper script, `submit-condor-job` will create a job description for Condor. The created job description should mirror the xRSL as closely as possible, except for the parts now handled by the wrapper script. And of course, the executable used in the Condor job description is now the wrapper script, *not* the actual executable specified in the xRSL.

Having created both a wrapper script and a job description for Condor, the only thing that remains in `submit-condor-job` is to queue the job for execution in the Condor pool (job submission is done with Condor's standard user interface, i.e., `condor_submit`). Once the job is submitted to Condor, `submit-condor-job` will append the job ID assigned by Condor to the GRAMI file, allowing other parts of the ARC middleware to find the Condor job associated with each GRAMI file.

4.3.2 `finish-condor-job`

The purpose of the next program, `finish-condor-job`, is to inform ARC that a Condor job has finished. The normal way to handle job finalization in ARC is to repeatedly poll each job with the program `scan-condor-job`. This inefficient approach is thankfully not necessary with Condor, which has asynchronous feedback: The typical Condor configuration is to inform the job owner by email that his job has finished, but the mail program is configurable. In this environment, it is the Grid Manager that needs to be notified, not the owner (at least not at this time, and in this way). Therefore, Condor must be configured to substitute the standard Unix mail program, `/bin/mail`, with another program, `finish-condor-job`, written especially for communicating with the Grid manager.

`finish-condor-job` does basically two things: First, it determines the exit code of the finished job, and second, it writes this code to a designated file in the Grid Manager's control directory. Noticing the exit code, the Grid Manager will know that the job has finished and will clean up the job's session directory, leaving only the output files, ready to be downloaded by the job owner.

Note that, even though `finish-condor-job` renders `scan-condor-job` useless, a program by that name is still installed. This is because ARC expects all LRMSes to implement a `scan-`*`lrms`*`-job`, and will always try to run this program, even when it does not exist. This fills up the log with unnecessary error messages, and to fix it, a dummy version of `scan-condor-job`, whose only action is to sleep for one hour, is installed. Making the program sleep for this long might seem unnecessary, but `scan-condor-job` is repeatedly restarted, so making it sleep for a long time actually minimizes the system load.

4.3.3 `cancel-condor-job`

The last program in the job control interface is `cancel-condor-job`. The purpose of this script is obvious from the name: Cancel a Grid job running in the Condor pool on receipt of a ARC cancellation request.

Like the previously mentioned `submit-condor-job`, this program takes a GRAMI file as its only argument. The Condor job ID identifying the Condor job associated with the GRAMI file is retrieved, and `condor_rm` is used to terminate it immediately.

4.4 The Information System Interface

The second part of the ARC → Condor software, the Information System [29] interface, consists of two scripts: `cluster-condor.pl` and `queue+jobs+users-condor.pl`. These scripts are run periodically by ARC, providing a snapshot of the cluster's state at that time. Both scripts are based on the excellent templates accompanying the NorduGrid middleware.

4.4.1 `cluster-condor.pl`

`cluster-condor.pl` provides high-level information on the cluster, such as total number of CPUs, number of used CPUs, number of queued jobs, batch system type, version, and so on. This information is extracted using Condor's standard user interface (`condor_version`, `condor_status`, and `condor_q`).

4.4.2 `queue+jobs+users-condor.pl`

This program provides details on each individual job, both running and finished. Since the complete job history will always grow, efficiency in its

implementation was given high priority, and so communication with the Condor pool and disk accesses were kept at a minimum (if the script is too slow, the information system may assume that the cluster is down, causing it to drop out of the Grid Monitor).

For running jobs, information is extracted using the Condor programs `condor_status` and `condor_q`, while for finished jobs, the logs created by `submit-condor-job` and `finish-condor-job` contain all the necessary information.

Aside from the performance issues in `queue+jobs+users-condor.pl`, the primary challenge in writing the information system backend was the interpretation of the various attributes that have to be set by the information system. Explanations for each attribute can be found in the `nordugrid.schema` file distributed with ARC, but it was still not easy to decide how to represent all the values. Since the description was often very open for interpretation, careful thought had to be given to find values that closely match the semantics of the data returned by the PBS system, which in a way has set the standard by being the first supported batch system.

Chapter 5

ARC → Condor User Experiences

5.1 ATLAS DC2 via ARC → Condor

Having written the above programs, and successfully run some simple toy jobs, the ARC → Condor interface was ready to be challenged with a more realistic usage scenario. To really put it to the test, the system was used in the University of Oslo's participation in ATLAS Data Challenge 2 (DC2) which, for Northern Europe, would require an ARC cluster [30].

Two Condor pools were available for this purpose, and a Grid Manager was installed on both. One pool administered by the university's central IT department (USIT), the other one a smaller pool consisting of the desktop computers belonging to the group of Experimental Particle Physics.

A realistic test such as DC2 proved very valuable, unearthing several problems that were not detected in synthetic tests. Most of these problems could be fixed immediately without involving the developers of Condor or ARC, and are not worth a detailed discussion, but there was one issue with heterogeneous versus homogeneous clusters that deserves mention.

Our Condor pools are heterogeneous, whereas ARC's information system works best with homogeneous clusters, meaning that integration with ARC is easiest when every node in the cluster has approximately the same hardware.

In particular, it was a problem that ARC only specifies a single memory value for the entire cluster. In a heterogeneous pool, the difference in memory between the largest and smallest node will often be big. By advertising the lowest value, ARC would reject the cluster for all DC2 jobs, since all of them required more memory than our low-end machines. Conversely, increasing the advertised node memory to satisfy DC2's minimum requirements turned our cluster into a "black hole" that would at-

tract more jobs than it could handle.

The reason this happens is simple: ARC's information system is told the number of nodes in the Condor pool, and assumes they all have the advertised amount of memory, and will thus make its queuing decisions based on this. Condor, on the other hand, knows the exact amount of memory on each node, and once all nodes that satisfy the memory requirement are taken, the remaining jobs will wait in Condor's queue. How bad this really is will depend on the ratio of machines that make the memory cut to those that do not. In our pool, less than half the nodes made the cut, causing the queue to grow more than twice as big as it should.

In an effort to improve on this situation, the ARC → Condor interface was modified so that it would stop sucking in Grid jobs when the number of free nodes that satisfy the memory requirement drops to zero, and this change has the unfortunate consequence of also preventing smaller jobs from being accepted, even when there may well be free nodes in the Condor pool that could run them immediately, but it is the best solution until ARC's information system is extended to hold the entire memory distribution in the pool instead of a single number that is used for all nodes.

As is mentioned in Section 5.2 on future development, a better solution to the above problem is implementing virtual job queues that divide the heterogeneous pool into groups of machines of similar capabilities. This way, the queues with low-memory machines will still be available after the high-memory nodes are all occupied.

In the meantime, there are some factors that make the situation more tolerable: Small jobs that are submitted before all high-end machines are occupied will not block big jobs, unless Condor happens to run them on high-memory machines that exceed their minimum requirements.

The odds of that happening can be reduced by overriding the default ranking algorithm (read about the `condor_rank` attribute in the Condor setup section [31] in the ARC manual) to favor the lowest memory node that can still run the job. A downside to this trick is that low-memory nodes also tend to have slow CPUs, causing the fast nodes to be underutilized on pools with little traffic.

Finally, it should also be mentioned that, although the pool will not run jobs when there are no nodes that satisfy the memory limit advertised to ARC, it does not mean that you cannot *queue* jobs for execution when the nodes eventually become available. This should still be possible by explicitly specifying the name of the cluster.

5.2 Results and Future Development

At the time of writing, ATLAS DC2 has been running on Condor via the ARC → Condor interface since early summer 2004, and lately the success rate of DC2 jobs [32] has become the highest of all clusters in the NorduGrid project. The statistics for September 2004 is also included in Table 5.1. This period was chosen because the ARC → Condor interface had, at this time, matured enough to become useful in a production environment, and coincidentally, it was a month when our computers and network happened to be working smoothly. But still, considering that the top cluster (grid.uio.no) is running Condor on regular desktop computers that are frequently used by students and professors, it is nonetheless a surprisingly good result.

Cluster	**successful jobs**	**failed jobs**	**failure rate (%)**
grid.uio.no	1005	23	2
bluesmoke.nsc.liu.se	452	13	3
hypatia.uio.no	1497	75	5
farm.hep.lu.se	771	36	5
sg-access.pdc.kth.se	1019	68	7
benedict.aau.dk	2000	174	9
sigrid.lunarc.lu.se	772	72	9
lheppc10.unibe.ch	87	10	11
fe10.dcsc.sdu.dk	1544	242	16
ingrid.hpc2n.umu.se	572	94	16
hagrid.it.uu.se	2508	541	22
fire.ii.uib.no	580	154	27
lscf.nbi.dk	131	36	27
atlas.hpc.unimelb.edu.au	312	93	30
brenta.ijs.si	2078	674	32
atlas.fzk.de	1534	498	32
morpheus.dcgc.dk	1289	483	37
charm.hpc.unimelb.edu.au	66	27	41
genghis.hpc.unimelb.edu.au	47	23	49
hive.unicc.chalmers.se	18	10	56
lxsrv9.lrz-muenchen.de	83	49	59

Table 5.1: DC2 results for September 2004, ranked by failure rate (clusters using ARC → Condor are italicized)

The users of the machines allocated to the cluster used in DC2 were surprised by how little this experiment affected them in their daily work. Some extra fan noise was noted on hot summer days, since many of these machines are now running at 100 percent, 24 hours a day. One would also

occasionally notice a few seconds delay on login, when a running job was suspended and memory was swapped in from disk, but clearly, these are minor annoyances of no significance.

Our biggest problems had nothing to do with the ARC → Condor software, but rather limitations in hardware, stability issues in the local area network, and misconfigured machines. For example, there were initially some problems with the way our hard disks were partitioned, causing many jobs to fail because of disk space shortage. Another source of problems was the file server that hosted the multi-gigabyte ATLAS runtime environment. The load exceeded the server's capabilities, and this caused numerous stability problems. The failure rate dropped considerably once these issues had been taken care of. Finally, there was also a problem where the ATLAS application expected to find a substantial amount of free space on `/tmp` in addition to its working directory, and thus managed to trick Condor's ClassAd mechanism into guaranteeing sufficient disk space on machines that did not meet the job's actual requirements.

But overall, running ATLAS Data Challenge 2 on Condor via ARC was successful enough to prove that the time spent writing the interface to make this possible was well worth it. The Condor interface itself is currently very stable, and is now a standard part of the ARC distribution.

As for future development, support for multiple job queues is very high on the to-do list. Condor's tolerance for diversity can become a problem in environments that expect homogeneity[1] and multiple job queues can possibly be used to divide up a diverse cluster into multiple (more or less) homogeneous queues. One could, for example, make partitions based on memory, so that machines with less than 256 MB fall into one queue, machines with 256–512 MB in another, and so on.

Also on the list is support for flocking-enabled Condor pools and non-x86/x86-64 architectures. Flocking-enabled pools may already be working but has not yet been tested, and support for other CPU architectures and other flavors of Unix should be a trivial enhancement since all of the code in this project is shell scripts and Perl code. As long as recent[2] versions of Perl and the GNU core utilities are installed, the software should behave the same regardless of the flavor of Unix and the CPU architecture. It should be as simple as making the program that creates the Condor job description specify the appropriate (according the xRSL) architecture and operating system.

[1] As discussed in Section 5.1.

[2] Software requirements for the Condor interface are specified in the documentation accompanying ARC.

Chapter 6

Conclusion

The central theme in this thesis has been the continually growing demand for computing resources in modern science – advanced resource management systems such as Condor, and Grid tools for distributing jobs across institutional and national borders have emerged to fill this need; the basis for this thesis was the idea that these two technologies can be combined to form an even more powerful tool.

The successful application of the ARC → Condor interface in ATLAS Data Challenge 2 has proven this to be true and, hopefully, the creation of the ARC → Condor interface will make it easier for many institutions to participate in and contribute to Grid computing by enabling use of a batch system that does not require large investments in dedicated computer equipment, but instead makes use of hardware that is already used for different purposes, and resources that would otherwise have gone to waste.

Appendix A

Source Code

A.1 `update-hosts`

This file is available for download at ⟨`http://folk.uio.no/hakonrk/src/misc/update-hosts`⟩.

```
#!/bin/sh
#
# Update the IP address of the line in /etc/hosts that corresponds
# to the machine where this script is run.  The current IP address
# is determined using ifconfig on the interface given as the
# command line argument.  Typically called as 'update-hosts eth0'.
#

# Fail with usage description if no interface was specified.
if [ -z "$1" ]; then
    echo "Usage: $(basename $0) INTERFACE" >&2
    exit 1
fi

# Extract the IP address from the requested interface with "ifconfig".
IPADDR=$(/sbin/ifconfig $1 | sed -n 's/^ *inet addr:\([^ ]*\).*/\1/p')

# Verify that we got a valid IP address.
if ! ping -c 1 "$IPADDR" &>/dev/null; then
    mail -s "updates-hosts: invalid IP address: $IPADDR" root </dev/null
    exit 1
fi

# Remove junk files on exit.
trap "rm -f /etc/hosts.tmp /tmp/update-hosts.$$" EXIT
```

```
# Update the IP address for the line with the fully qualified hostname.
TAB=$(printf \\t)
sed "/\<$(hostname -f)\>/s/^[ $TAB]*[^ $TAB]*\(.*\)/$IPADDR\1/" \
  /etc/hosts >/etc/hosts.tmp

# If the update was a success, replace current /etc/hosts in a
# safe way.
if [ -s /etc/hosts.tmp ]; then
    mv -f /etc/hosts.tmp /etc/hosts &>/tmp/update-hosts.$$
    if [ -f /etc/hosts.tmp ]; then
        mail -s 'update-hosts failed' root </tmp/update-hosts.$$
    fi
elif [ ! -f /etc/hosts.tmp ]; then
    mail -s 'update-hosts: could not write /etc/hosts.tmp' root </dev/null
else
    mail -s 'update-hosts: no labeled line in /etc/hosts' root </dev/null
fi
```

A.2 `test_mkstemp.c`

This file is available for download at ⟨`http://folk.uio.no/hakonrk/src/misc/test_mkstemp.c`⟩.

```
/*
 * A simple example of what a test program for verifying system
 * calls could look like.  This program tests the mkstemp() call
 * (which was known to be broken in older versions of Condor).
 * To run this test, compile the program for Condor using
 * 'condor_compile gcc test_mkstemp.c' and submit the resulting
 * executable as a standard universe job.
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    int fd;
    char tempfile[] = "tmp.XXXXXX";

    if ((fd = mkstemp(tempfile)) == -1) {
        perror("mkstemp");
        return 1;
    }

    printf("mkstemp returned file descriptor %d and filename %s\n",
```

```
            fd, tempfile);
    fflush(stdout);

    if (write(fd, "hello, world\n", 13) != 13) {
        perror("write");
        return 1;
    }
    close(fd);

    return 0;
}
```

A.3 iobench.c

This file is available for download at ⟨http://folk.uio.no/hakonrk/src/iobench/iobench.c⟩.

```
/*
 * This program was used to test I/O performance under Condor's
 * standard universe by writing DATASIZE bytes to disk in various
 * block sizes (64, 128, ..., 65536).  The time it took to write
 * DATASIZE bytes for each block size is timed and displayed
 * on stdout.  The average system load is also displayed, to
 * make it easier to discard results that were perturbed by other
 * processes competing for CPU time.
 */
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define DATASIZE (32 * 1024 * 1024)

/* Wrapper function for mkstemp() that terminates the program on errors. */
int Mkstemp(char *template)
{
        int fd;

        if ((fd = mkstemp(template)) < 0) {
                perror(template);
```

```
                abort();
        }

        return fd;
}

/* Wrapper function for open() that terminates the program on errors. */
int Open(const char *pathname, int flags)
{
        int fd;

        if ((fd = open(pathname, flags)) < 0) {
                perror(pathname);
                abort();
        }

        return fd;
}

/* Create a temporary file using mktemp() and open(). */
int Tempfile(char *template)
{
        if (mktemp(template) == NULL) {
                abort();
        }
        unlink(template);
        return Open(template, O_CREAT | O_WRONLY);
}

/* Wrapper function for close() that terminates the program on errors. */
int Close(int fd)
{
        if (close(fd) != 0) {
                perror("close()");
                abort();
        }

        return 0;
}

/* Writes n bytes to file descriptor fd.  Handles interrupts and
 * incomplete writes. */
ssize_t writen(int fd, const void *buf, size_t n)
{
        const char     *ptr;
        size_t          nleft;
        ssize_t         nwritten;

        ptr = buf;
```

```
        nleft = n;
        while (nleft > 0) {
                if ((nwritten = write(fd, ptr, nleft)) < 0) {
                        if (errno == EINTR) {
                                continue;
                        } else {
                                return -1;
                        }
                }
                nleft -= nwritten;
                ptr += nwritten;
        }
        return (ssize_t) n;
}

/* Wrapper function for writen() that terminates the program on errors. */
ssize_t Write(int fd, const void *buf, size_t count)
{
        if (writen(fd, buf, count) != (ssize_t) count) {
                perror("write()");
                abort();
        }

        return count;
}

/* Wrapper function for fsync() that terminates the program on errors. */
int Fsync(int fd)
{
        if (fsync(fd) != 0) {
                perror("fsync()");
                abort();
        }

        return 0;
}

void print_loadavg(void)
{
        char buf[50000];
        FILE *loadavg;

        if ((loadavg = fopen("/proc/loadavg", "r")) == NULL) {
                perror("/proc/loadavg");
                abort();
        }
        fgets(buf, sizeof(buf), loadavg);
        fputs(buf, stdout);
        fflush(stdout);
```

```
        fclose(loadavg);
}

void randomize_buffer(char *buffer, size_t blocksize)
{
        unsigned i;
        unsigned rest = blocksize % sizeof(int);

        blocksize -= rest;

        /* Copy whole "int"s, as long as possible. */
        for (i = 0; i < blocksize; i += sizeof(int)) {
                *((int *) &buffer[i]) = rand();
        }

        /* If "blocksize" wasn't divisible by sizeof(int), fill
         * the rest here. */
        for (i = 0; i < rest; i++) {
                *buffer++ = 1 + (int) (255.0 * rand() / (RAND_MAX + 1.0));
        }
}

/*
 * Run benchmark using given blocksize.
 */
void do_iobench(int blocksize)
{
        struct timeval   tv1, tv2;
        char            *buffer;
        double           dt;
        long             n_written = 0;
        int              fd;
        char             template[] = "out.iobench.XXXXXX";

        /* Seed the random generator with a fixed number.  This ensures that
         * the file created is the same every time, but is still
         * incompressible. */
        srand(0);

        /* Allocate memory for I/O buffer. */
        if (!(buffer = malloc(blocksize))) {
                perror("malloc");
                abort();
        }
        memset(buffer, 0, blocksize);

        /* Get file descriptor to tempfile (name now stored in template). */
        fd = Tempfile(template);
```

```
        /* Write DATASIZE bytes using blocksize sized blocks, and get time
         * before and after writing. */
        gettimeofday(&tv1, NULL);
        do {
                randomize_buffer(buffer, blocksize);
                n_written += Write(fd, buffer, blocksize);
        } while (n_written < DATASIZE);
        Fsync(fd);
        gettimeofday(&tv2, NULL);

        /* Get time delta in seconds. */
        dt = tv2.tv_sec - tv1.tv_sec
           + (double) (tv2.tv_usec - tv1.tv_usec) / 1000000.0;

        /* Print results. */
        printf("---\n");
        printf("Blocksize %d: %g sec, %g kB/s\n", blocksize, dt, (DATASIZE/1024) / dt);
        print_loadavg();

        Close(fd);
        unlink(template);

        free(buffer);
}

int main(void)
{
        int blocksize;

        /* Do benchmark for block sizes 64, 128, 256, ..., 65536. */
        for (blocksize = 64; blocksize < (128*1024); blocksize *= 2) {
                do_iobench(blocksize);
        }

        return 0;
}
```

A.4 scbench.c

This file is available for download at ⟨http://folk.uio.no/hakonrk/src/scbench/scbench.c⟩.

```
/*
 * This program was used to test the system call overhead in
 * Condor's standard universe.  The method is simple:  do the
 * stat() system call on the current working directory N_ITER
```

```
 * times, and print the elapsed time on stdout once it's finished.
 * The average system load is also displayed on startup, to make
 * it easier to discard results that were perturbed by other
 * processes competing for CPU time.
 */
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define N_ITER 1000000

/* Initialize the timer. */
#define TIME_START \
        { \
                struct timeval t1, t2; \
                double dt; \
                gettimeofday(&t1, NULL);

/* Stop the timer and print out the elapsed time since the timer
 * was started. */
#define TIME_STOP \
                gettimeofday(&t2, NULL); \
                dt = (t2.tv_sec - t1.tv_sec) * 1000.0 + \
                     (t2.tv_usec - t1.tv_usec) / 1000.0; \
                printf("%s(): %g ms\n", __FUNCTION__, dt); \
                fflush(stdout); \
        }

void print_loadavg(void)
{
        char buf[4096];
        FILE *loadavg;

        if ((loadavg = fopen("/proc/loadavg", "r")) == NULL) {
                perror("/proc/loadavg");
                abort();
        }
        fgets(buf, sizeof(buf), loadavg);
        fputs(buf, stdout);
        fflush(stdout);
        fclose(loadavg);
}

int main(void)
{
        struct stat buf;
```

```
        int i;

        /* Print out the system load, so that we can be sure we're using an
         * idle system. */
        print_loadavg();

        /* Print out the inode number for verification. */
        stat(".", &buf);
        printf("inode: %d\n", (int) buf.st_ino);

        TIME_START;
        for (i = 0; i < N_ITER; i++) {
                stat(".", &buf);
        }
        TIME_STOP;

        return 0;
}
```

Appendix B

The ARC → Condor Interface Source Code

All files in this chapter is part of the NorduGrid middleware, which is available for download at ⟨http://www.nordugrid.org/⟩.

B.1 LRMS_Condor.pm

This Perl module contains shared code that is used internally by both the Grid Manager and the Information System interface.

```
package LRMS_Condor;

use strict;
use warnings;

BEGIN {
    use base 'Exporter';

    # Set the version for version checking.
    our $VERSION = '1.000';

    # This export list can be generated with the following Vim command:
    # :r !awk '/\<sub (lrms|nord|condor_)/ { print "       ", $2 }' % | sort
    our @EXPORT = qw(
        condor_config
        condor_get_job_field
        condor_get_queued
        condor_get_running
        condor_job_suspended
        condor_location
```

```
        condor_rank
        condor_run
        condor_status_total
        lrms_get_job_executionnodes
        lrms_get_job_executionnodes_completed
        lrms_get_job_status
        lrms_get_jobinfo_logged
        lrms_get_localids_running
        lrms_get_queued
        lrms_get_total
        nordugrid_authuser_freecpus
        nordugrid_authuser_queuelength
        nordugrid_cluster_cpudistribution
        nordugrid_cluster_lrms_type
        nordugrid_cluster_lrms_version
        nordugrid_cluster_totalcpus
        nordugrid_cluster_usedcpus
        nordugrid_job_lrmscomment
        nordugrid_job_queuerank
        nordugrid_job_reqcput
        nordugrid_job_usedcputime
        nordugrid_job_usedmem
        nordugrid_job_usedwalltime
        nordugrid_queue_defaultcputime
        nordugrid_queue_gridqueued
        nordugrid_queue_gridrunning
        nordugrid_queue_maxcputime
        nordugrid_queue_maxqueuable
        nordugrid_queue_maxrunning
        nordugrid_queue_maxuserrun
        nordugrid_queue_mincputime
        nordugrid_queue_queued
        nordugrid_queue_running
        nordugrid_queue_status
    );
}

#
# Total number of CPUs available for running jobs.
#
sub nordugrid_cluster_totalcpus {
    return (condor_status_total())[1];
}

#
# Number of CPUs that are busy, either in a job or interactive use.
#
sub nordugrid_cluster_usedcpus {
    my @total = condor_status_total();
```

```
    return $total[2] + $total[3];
}

#
# CPU distribution string (e.g., '1cpu:5 2cpu:1').
#
sub nordugrid_cluster_cpudistribution {
    # List all machines in the pool.  Machines with multiple CPUs are listed
    # one time for each CPU, with a prefix such as 'vm1@', 'vm2@', etc.
    my ($out,$err,$ret) = condor_run('bin/condor_status -format "%s\n" Name');

    # Count the number of CPUs for all machines in the pool.
    my %machines;
    for (split /\n/, $out) {
        next if /^\s*$/;
        my ($hostname) = $_ =~ /([^@]+)$/;
        $machines{$hostname}++;
    }

    # Count number of machines with one CPU, number with two, etc.
    my %dist;
    for (keys %machines) {
        $dist{$machines{$_}}++;
    }

    # Generate CPU distribution string.
    my $diststr = '';
    for (sort { $a <=> $b } keys %dist) {
        $diststr .= ' ' unless $diststr eq '';
        $diststr .= "${_}cpu:$dist{$_}";
    }

    return $diststr;
}

#
# Text string containing the LRMS type.
#
sub nordugrid_cluster_lrms_type {
    return 'Condor';
}

#
# String containing LRMS version.  ('UNKNOWN' in case of errors.)
#
sub nordugrid_cluster_lrms_version {
    my ($out, $err, $ret) = condor_run('bin/condor_version');
    return 'UNKNOWN' if $ret != 0;
    $out =~ /\$CondorVersion:\s+(\S+)/;
```

```
    return $1 || 'UNKNOWN';
}

#
# (Used to compute 'nordugrid-cluster-queuedjobs'.)
# Returns the number of queued jobs (idle and held) in the LRMS pool.  Counts
# both Grid jobs and jobs submitted directly to the LRMS by local users.
#
sub lrms_get_queued {
    my ($out, $err, $ret) = condor_run('bin/condor_q -global');
    return 0 if $ret != 0;
    return condor_get_queued($out);
}

#
# Returns the number of queued jobs (idle and held) in the LRMS pool.  Counts
# both Grid jobs and jobs submitted directly to the LRMS by local users.
# TODO: handle multiple queues!
#
sub nordugrid_queue_queued {
    return lrms_get_queued();
}

#
# Returns the number of queued jobs (idle and held) in the LRMS pool.  Counts
# only Grid jobs.
# TODO: handle multiple queues!
#
sub nordugrid_queue_gridqueued {
    my ($out, $err, $ret) = condor_run('bin/condor_q');
    return 0 if $ret != 0;
    return condor_get_queued($out);
}

#
# Helper function to lrms_get_queued() and lrms_get_gridqueued().
# Takes one argument, the output of a condor_q command.
# Returns the number of queued jobs (idle + held).
#
sub condor_get_queued {
    my $out = shift;
    my $sum = 0;
    $sum += $_ for map { /(\d+) idle.*?(\d+) held/ && $1 + $2 }
                  grep { /^\d+ jobs;/ } split /\n/, $out;
    return $sum;
}

#
# (Used to compute 'nordugrid-cluster-totaljobs'.)
```

```
# Returns the number of active jobs (running) in the LRMS pool.  Counts both
# Grid jobs and jobs submitted directly to the LRMS by local users.
#
sub lrms_get_total {
    my ($out, $err, $ret) = condor_run('bin/condor_q -global');
    return 0 if $ret != 0;
    return condor_get_running($out);
}

#
# Takes one argument:  the output from condor_q.
# Counts all running jobs listed in this output.
#
sub condor_get_running {
    my $sum = 0;
    $sum += $_ for $_[0] =~ /^\d+ jobs;.*?(\d+) running,/gm;
    return $sum;
}

#
# Takes one optional argument:  A string that, if defined, is placed on
# condor_status' command line, just after -total.
#
# Returns a list of the values in the 'Total' line in condor_status -total.
# (0:'Total' 1:Machines 2:Owner 3:Claimed 4:Unclaimed 5:Matched 6:Preempting).
# On errors: (Total 0 0 0 0 0 0).
#
sub condor_status_total {
    my $extra_args = (defined $_[0] ? " $_[0]" : '');
    my @failval = qw(Error 0 0 0 0 0 0);
    my ($out, $err, $ret) = condor_run("bin/condor_status -total$extra_args");
    return @failval if $ret != 0;
    $out =~ /^\s*(Total.*)$/m or return @failval;
    return split ' ', $1;
}

#
# Returns 'inactive' if condor_status fails.
#
sub nordugrid_queue_status {
    my @total = condor_status_total();
    return $total[0] eq 'Error' ? 'inactive' : 'active';
}

#
# Returns total number of CPUs claimed by jobs.  (This is equal to the number
# of running jobs.)
# TODO: handle multiple queues!
#
```

```
sub nordugrid_queue_running {
    return lrms_get_total();
}

#
# Returns number of running jobs on Condor that came from the Grid.  Since
# condor_q by default lists only the jobs submitted from the machine where
# condo_q is running, and only the Grid Manager is allowed to submit from
# there, we can easily tell how many jobs belong to the Grid.
#
sub nordugrid_queue_gridrunning {
    my ($out, $err, $ret) = condor_run('bin/condor_q');
    return 0 if $ret != 0;
    return condor_get_running($out);
}

#
# Returns the number of CPUs in the Condor pool, which is always equal to the
# maximum number of running jobs.
#
sub nordugrid_queue_maxrunning {
    return nordugrid_cluster_totalcpus();
}

#
# Returns 2 * maxrunning, which is an arbitrary number.  There is (as far as I
# know) no limit on the number of queued jobs.
#
sub nordugrid_queue_maxqueuable {
    return 2 * nordugrid_queue_maxrunning();
}

#
# Returns the maximum number of jobs that a single user can run at once.
# TODO: I don't know the details as to how Condor handles this.
# TODO: handle multiple queues!
#
sub nordugrid_queue_maxuserrun {
    return nordugrid_queue_maxrunning();
}

#
# There's no limit on the CPU time in Condor, so leave this blank.
#
sub nordugrid_queue_maxcputime {
    return '';
}

#
```

```
# There's no limit on the CPU time in Condor, so leave this blank.
#
sub nordugrid_queue_mincputime {
    return '';
}

#
# Always returns maxcputime, since there's no limit on the CPU time in Condor.
#
sub nordugrid_queue_defaultcputime {
    return nordugrid_queue_maxcputime();
}

#
# Takes two arguments:
#
#  1. A Condor ClassAd attribute name.
#  2. A Condor job ID (<ClusterId>.condor).
#
# Returns the value of the attribute named in the first argument for the job
# specified in the second argument.
#
{
    my %jobdata;

    sub condor_get_job_field {
        my $field = $_[0];
        my ($cluster) = $_[1] =~ /(.*)\.condor/;

        if (%jobdata) {
            return $jobdata{$cluster}{$field};
        }

        %jobdata = ();
        my ($out, $err, $ret) = condor_run("bin/condor_q -long");
        for my $jobdata (split /\n\n/, $out) {
            my ($clusterid) = $jobdata =~ /^ClusterId = (.*)$/m;
            for my $line (split /\n/, $jobdata) {
                my ($field, $value) = $line =~ /^(.*?) = (.*)$/m or next;
                $jobdata{$clusterid}{$field} = $value;
            }
        }
        return $jobdata{$cluster}{$field};
    }
}

#
# (Used in 'nordugrid-job-status'.)
# Takes two arguments:
```

```
# 1. The LRMS job id as represented in the GM.  (In Condor terms,
#    it's <cluster>.condor.  <proc> is not included, since only
#    one job is submitted at a time, so <proc> is always zero.)
# 2. The 'controldir' attribute from nordugrid.conf.
#
# Returns the current status of the job by mapping Condor's JobStatus
# integer into corresponding one-letter codes used by ARC:
#
#   1 (Idle)       --> Q (job is queuing, waiting for a node, etc.)
#   2 (Running)    --> R (running on a host controlled by the LRMS)
#   2 (Suspended)  --> S (an already running job in a suspended state)
#   3 (Removed)    --> E (finishing in the LRMS)
#   4 (Completed)  --> E (finishing in the LRMS)
#   5 (Held)       --> O (other)
#
# If the job couldn't be found, E is returned since it is probably finished.
#
sub lrms_get_job_status {
    my %num2letter = qw(1 Q 2 R 3 E 4 E 5 O);
    my $s = condor_get_job_field('JobStatus', $_[0]);
    return 'E' if !defined $s;
    $s = $num2letter{$s};
    if ($s eq 'R') {
        $s = 'S' if condor_job_suspended(@_);
    }
    return $s;
}

#
# There's no easy way to define the job's queue "position" in Condor.
#
sub nordugrid_job_queuerank {
    return '';
}

#
# Takes one argument, the LRMS job id as represented in the GM.  (In Condor
# terms, it's <cluster>.condor.  <proc> is not included, since only one job is
# submitted at a time, so <proc> is always zero.)
#
# Returns number of minutes of CPU time the job has consumed, rounded to the
# nearest minute.
#
sub nordugrid_job_usedcputime {
    my $time = condor_get_job_field('RemoteUserCpu', $_[0]);
    return 0 if !defined $time;
    return sprintf "%.0f", $time / 60;
}
```

```
#
# Takes one argument, the LRMS job id as represented in the GM.  (In Condor
# terms, it's <cluster>.condor.  <proc> is not included, since only one job is
# submitted at a time, so <proc> is always zero.)
#
# Returns number of minutes the job has been allocated to a machine, rounded to
# the nearest minute.
#
sub nordugrid_job_usedwalltime {
    my $time = condor_get_job_field('RemoteWallClockTime', $_[0]);
    return 0 if !defined $time;
    return sprintf "%.0f", $time / 60;
}

#
# Takes one argument, the LRMS job id as represented in the GM.  (In Condor
# terms, it's <cluster>.condor.  <proc> is not included, since only one job is
# submitted at a time, so <proc> is always zero.)
#
# Returns virtual image size of executable in KB.
#
sub nordugrid_job_usedmem {
    my $size = condor_get_job_field('ImageSize', $_[0]);
    return defined $size ? $size : 0;
}

#
# Condor has no "requested CPU time" attribute.
#
sub nordugrid_job_reqcput {
    return '';
}

#
# (Used in 'nordugrid-job-executionnodes'.)
# Takes one argument, the LRMS job id as represented in the GM.  (In Condor
# terms, it's <cluster>.condor.  <proc> is not included, since only one job is
# submitted at a time, so <proc> is always zero.)
#
# Returns the node the job runs on, or last ran on in case the job is not
# currently running.  Only returns one node, since we don't support MPI
# jobs.
#
sub lrms_get_job_executionnodes {
    my $tmp = condor_get_job_field('RemoteHost', $_[0]);
    if ($tmp) {
        my ($host) = $tmp =~ /^"(.+)"$/;
        return $host if $host;
    }
```

```
    $tmp = condor_get_job_field('LastRemoteHost', $_[0]);
    if ($tmp) {
        my ($host) = $tmp =~ /^"(.+)"$/;
        return $host if $host;
    }
    return 'UNKNOWN';
}

#
# Like lrms_get_job_executionnodes(), but this version looks at the
# job.ID.errors file to find out where completed jobs ran.
#
sub lrms_get_job_executionnodes_completed {
    my $gmlog = shift;
    local *GMLOG;
    open GMLOG, "<$gmlog" or return '';
    local $/;
    my $logdata = <GMLOG>;
    close GMLOG;
    my ($exechost) = $logdata =~ /.+Job executing on host: <([^:]+)/s;
    return $exechost || '';
}

sub lrms_get_jobinfo_logged {
    my ($jobinfo, $ctldir) = @_;

    my $e1 = '----- starting finish-condor-job -----';
    my $e2 = '.* Job executing on host: <.*>';
    my $e3 = 'Allocation/Run time:.*';
    my $e4 = 'Total Remote CPU Time:.*';
    my $tmp = `egrep -H '($e1|finish-condor-job: ($e2|$e3|$e4))\$' \\
                $ctldir/job.*.errors`;

    for my $chunk (split /^.*:\Q$e1\E\n/m, $tmp) {
        # Currently only fetching the last host the job executed on.
        my ($exechost) =
          $chunk =~ /.*finish-condor-job:[^\n]+executing on host: <([^:]+)/s;
        next if !$exechost;

        # The GM job id.
        my ($id) = $chunk =~ /job\.([^.]+)\.errors:/;

        # Strings in the form <days> <hours>:<minutes>:<seconds>.
        my ($walltstr) = $chunk =~ m{.*Allocation/Run time:\s+([^\n]+)}s;
        my ($cputstr) = $chunk =~ m{Total Remote CPU Time:\s+([^\n]+)};

        # Convert wallclock time string to minutes.
        my ($d, $h, $m, $s) = $walltstr =~ /(\d+) (\d\d):(\d\d):(\d\d)/;
        {
```

```
            no warnings 'uninitialized';
            $m += $d * 24 * 60 + $h * 60 + $s / 60;
            $m = sprintf '%.0f', $m;
        }
        $jobinfo->{$id}{WallTime} = $m;

        # Convert CPU time string to minutes.
        ($d, $h, $m, $s) = $cputstr =~ /(\d+) (\d\d):(\d\d):(\d\d)/;
        {
            no warnings 'uninitialized';
            $m += $d * 24 * 60 + $h * 60 + $s / 60;
            $m = sprintf '%.0f', $m;
        }
        $jobinfo->{$id}{CpuTime} = $m;

        # Execution host.
        $jobinfo->{$id}{exec_host} = $exechost;

        # Required CPU time.
        $jobinfo->{$id}{reqcputime} = '';
    }
}

#
# Takes one argument, the LRMS job id as represented in the GM.  (In Condor
# terms, it's <cluster>.condor.  <proc> is not included, since only one job is
# submitted at a time, so <proc> is always zero.)
#
# Returns no useful comment yet, but I'll improve this in the future.
#
sub nordugrid_job_lrmscomment {
    return '';
}

#
# Returns a list of job IDs corresponding to 'localid' in the GM.
# Only lists currently running jobs.
#
sub lrms_get_localids_running {
    my ($out, $err, $ret) = condor_run("bin/condor_q");
    $out =~ s/.*?CMD[ \t]*\n//s;
    $out =~ s/\n\n.*//s;
    my @localids;
    for (split /\n/, $out) {
        /^\s*(\d+)/;
        push @localids, "$1.condor";
    }
    return @localids;
}
```

```
#
# Currently set to the number of free nodes.
#
sub nordugrid_authuser_freecpus {
    return nordugrid_cluster_totalcpus() - nordugrid_cluster_usedcpus();
}

#
# Returns number of jobs queued by Grid.
#
sub nordugrid_authuser_queuelength {
    return nordugrid_queue_gridqueued();
}

my %condor_runcache;

#
# Takes one argument, which is a path to an executable (relative to
# CONDOR_LOCATION) that is appended to CONDOR_LOCATION, plus optional
# arguments.  The next time this function is called with exactly the
# same argument, the return value is fetched from %condor_runcache.
#
# Returns a list of three values:
#
# [0] String containing stdout.
# [1] String containing stderr.
# [2] Program exit code ($?) that was returned to the shell.
#
sub condor_run {
    my $condorloc = condor_location();
    if (not -e $ENV{CONDOR_CONFIG}) {
        $ENV{CONDOR_CONFIG} = "$condorloc/etc/condor_config";
    }
    my $program = "$condorloc/$_[0]";
    return @{$condor_runcache{$program}} if $condor_runcache{$program};
    my $stderr_file = "/tmp/condor_run.$$";
    my $stdout = `$program 2>$stderr_file`;
    my $ret = $? >> 8;
    local *ERROR;
    open ERROR, "<$stderr_file"
      or return @{$condor_runcache{$program} = [$stdout, '', $ret]};
    local $/;
    my $stderr = <ERROR>;
    close ERROR;
    unlink $stderr_file;
    return @{$condor_runcache{$program} = [$stdout, $stderr, $ret]};
}
```

```
#
# Takes two arguments, the Condor job id and the 'controldir' attribute from
# nordugrid.conf.  This function searches controldir for the grami file that
# belongs to the given Condor job, and extracts the location of the Condor job
# from it.  This log is parsed to see if the job has been suspended.  (condor_q
# reports 'R' for running even when the job is suspended, so we need to parse
# the log to be sure that 'R' actually means running.)
#
# Returns true if the job is suspended, and false if it's running.
#
{
    my $initialized_condor_log_db = 0;
    my %condor_log_db;

    sub condor_job_suspended {
        my ($localid, $controldir) = @_;

        # The first time condor_job_suspended() is called, the log database
        # must be initialized.
        if (!$initialized_condor_log_db) {
            $initialized_condor_log_db = 1;
            my @out = `egrep -H '^(joboption_jobid|condor_log)=' \\
                        $controldir/job.*.grami`;
            my $i = 0;
            while ($i + 1 < @out) {
                my $joboptline = $out[$i];  # joboption_jobid=...
                my $logline = $out[$i + 1]; # condor_log=...
                my ($grami, $id) =
                  $joboptline =~ /^(.+\.grami):joboption_jobid=(.*)/;
                if (!$grami || $out[$i + 1] !~ /^\Q$grami\E:/) {
                    # Grami didn't have both joboption_jobid and condor_log;
                    # Should not happen, but you never know!
                    $i++;
                    next;
                }
                my ($log) = $logline =~ /^\Q$grami\E:condor_log=(.*)/;
                $condor_log_db{$id} = $log;
                $i += 2;
            }
        }

        my $logfile = $condor_log_db{$localid};
        return 0 if !$logfile;
        local *LOGFILE;
        open LOGFILE, "<$logfile" or return 0;
        my $suspended = 0;
        while (my $line = <LOGFILE>) {
            $suspended = 1 if $line =~ /Job was suspended\.$/;
            $suspended = 0 if $line =~ /Job was unsuspended\.$/;
```

```
        }
        close LOGFILE;
        return $suspended;
    }
}

{
    my ($progdir) = $0 =~ m#(.*)/#;

    # Cached location so that subsequent calls are free.
    my $location;

    sub condor_location {
        return $location if defined $location;

        my $exe;

        # Extract condor_location from nordugrid.conf.
        my $nordugrid_conf = $ENV{NORDUGRID_CONFIG} || '/etc/nordugrid.conf';
        if (-r $nordugrid_conf) {
            $location = `eval "\$(egrep '^[[:blank:]]*condor_location=' \\
                                $nordugrid_conf)"; echo "\$condor_location"`;
            chomp $location;
            return $location if -x "$location/bin/condor_submit";
        }

        # Search for condor_submit in PATH.
        -x ($exe = "$_/condor_submit") and last for split /:/, $ENV{PATH};
        ($location) = $exe =~ m{(.*)/bin/condor_submit$} if -x $exe;
        return $location if $location;

        # Search for CONDOR_LOCATION in /etc/sysconfig/condor.
        if (-f '/etc/sysconfig/condor') {
            $location = `. /etc/sysconfig/condor; echo -n \$CONDOR_LOCATION`;
            return $location if -x "$location/bin/condor_submit";
        }

        # Use condor_master_location, if installed.
        if (-x "$progdir/condor_master_location") {
            ($location) = `$progdir/condor_master_location` =~ m{(.*)/sbin$};
            return $location if -x "$location/bin/condor_submit";
        }

        return $location = '';
    }

    my $config;

    sub condor_config {
```

```
        return $config if defined $config;

        my $nordugrid_conf = $ENV{NORDUGRID_CONFIG} || '/etc/nordugrid.conf';
        if (-r $nordugrid_conf) {
            $config = `eval "\$(egrep '^[[:blank:]]*condor_config=' \\
                                $nordugrid_conf)"; echo "\$condor_config"`;
            chomp $config;
            return $config if -r $config;
        }

        $config = condor_location() . "/etc/condor_config";
        return $config if -r $config;

        $config = $ENV{CONDOR_LOCATION} || '';
        return $config if -r $config;

        return $config = '';
    }

    sub condor_rank {
        my $nordugrid_conf = $ENV{NORDUGRID_CONFIG} || '/etc/nordugrid.conf';
        return undef if !-r $nordugrid_conf;
        my $rank = `eval "\$(egrep '^[[:blank:]]*condor_rank=' \\
                            $nordugrid_conf)"; echo "\$condor_rank"`;
        chomp $rank;
        return $rank;
    }
}

1;
```

B.2 submit-condor-job

```
#!/usr/bin/env perl

use File::Temp 'tempfile';
use lib ($ENV{NORDUGRID_LOCATION} ? "$ENV{NORDUGRID_LOCATION}/libexec" : '.');
use LRMS_Condor;

use strict;
use warnings;

# True if the program is run in debugging mode.
my $debug;

# The name of the Condor job description file.  This file is generated from the
```

```
# GRAMI file.
my $cmd_filename;

# Pathname of the Condor log.
my $condor_log;

# Pathname of the real executable.
my $real_exe;

# Pathname of the wrapper script.
my $exewrapper;

my %grami;

$0 =~ s#.*/##;
warn "----- starting $0 -----\n";
die "Usage: $0 [-d] GRAMI_FILE\n" unless @ARGV;

if (@ARGV == 2 && $ARGV[0] eq '-d') {
    $debug = 1;
    shift;
}

parse_grami(my $gramifile = $ARGV[0]);
create_shell_wrapper();
create_condor_job_description();
submit_condor_job();
warn "$0: job submitted successfully\n",
     "----- exiting $0 -----\n";
exit;

##############################################################################
## Function Definitons
##############################################################################

sub parse_grami {
    local @ARGV = $_[0];
    warn "$0: ----- begin grami file ($_[0]) -----\n";
    while (my $line = <>) {
        chomp $line;

        # Dump every line of the grami file into the log.
        warn "$0: $line\n";

        my ($name, $value) = split /=/, $line, 2;
        next if !$name;

        # Remove outer layer of single quotes.  Backslash escaped single quotes
        # are stripped of their backslashes, and strings protected by single
```

```
        # quotes are stripped of the single quotes.  This is supposed to work
        # exactly like Bourne shell quote removal:
        #
        #   foo'bar'     --> foobar
        #   foo\''bar'\' --> foo'bar'
        #
        {
            no warnings 'uninitialized';
            $value =~ s/(?:\\('))?'([^']*)'(?:\\('))?/$1$2$3/g;
        }

        # The variable names are case insensitive, so lowercase them and
        # remember to always refer to them by their lowercase names!
        $grami{lc $name} = $value;
    }
    warn "$0: ----- end grami file ($_[0]) -----\n";
}

#
# Returns the full path to the condor_submit executable, or an empty string on
# errors.  If called in debug mode, a bogus path is returned.
#
sub find_condor_submit_exe {
    return '/DEBUG/condor_submit' if $debug;
    my $condor_location = condor_location();
    my $exe = "$condor_location/bin/condor_submit";
    return (-x $exe ? $exe : '');
}

#
# Creates a shell script that:
#
#  (1) Sources the runtime scripts with argument "0" before evaluating the job
#      executable.  (This is in case the job refers to variables set by the
#      runtime scripts.)  TODO: should variables be expanded in
#      joboption_runtime_0?
#
#  (2) Sources the runtime scripts with argument "1" before running the job.
#
#  (3) Runs the job, redirecting output as requested in the xRSL.
#
#  (4) Sources the runtime scripts with argument "2" after running the job.
#
#  (5) Exits with the value returned by the job executable in step (3).
#
sub create_shell_wrapper {
    # Create the shell commands to run runtime environment files (stages 0-2).
    my ($setrte0, $setrte1, $setrte2) = ('true', '', '');
    if (notnull($grami{joboption_runtime_0})) {
```

```
    $ENV{RUNTIME_CONFIG_DIR} ||= '.';  # avoid undefined warning
    for (my $i = 0; notnull(my $r = $grami{"joboption_runtime_$i"}); $i++) {
        $setrte0 .= "; . \Q$ENV{RUNTIME_CONFIG_DIR}/$r\E 0";
        $setrte1 .= qq{. "\$RUNTIME_CONFIG_DIR/$r" 1\n};
        $setrte2 .= qq{. "\$RUNTIME_CONFIG_DIR/$r" 2\n};
    }
}

# Set $real_exe to the path to the job executable (environment variables
# expanded).  $exewrapper is the basename of $exe, plus some random
# characters for uniqueness.  Also, the file $exportfile is created,
# containing shell statements in the form
#
#   export NAME="value"
#
# for all environment variables in existence after sourcing the runtime
# scripts with argument 0.
my $exportfile = File::Temp::tempnam('/tmp', 'export.');
$real_exe = `{ $setrte0; } >/dev/null
             export >$exportfile
             echo -n $grami{joboption_arg_0}`;
$real_exe = "./$real_exe" if $real_exe !~ m{/};
$real_exe =~ m{([^/]+)$};
$exewrapper = File::Temp::tempnam($grami{joboption_directory}, "$1.");

# Get the name of the stdout file.
my $stdout = notnull($grami{joboption_stdout}) ?
             $grami{joboption_stdout} : '/dev/null';
$stdout =~ s{^\Q$grami{joboption_directory}\E/*}{};

# Get the name of the stderr file.
my $stderr = notnull($grami{joboption_stderr}) ?
             $grami{joboption_stderr} : '/dev/null';
$stderr =~ s{^\Q$grami{joboption_directory}\E/*}{};

# Start creating the output script.  Note that the script is created
# in-memory, instead of being written to file, part by part.  This is
# because we want to test for all I/O errors, and having just a single
# write means that there is only one place we have to test for write
# errors.
my $output = "#!/bin/sh\n";

# If the custom RSL attribute 'wrapperdebug' is set, enable command
# tracing (set -x) and list all files in the session directory.  (This
# output is sent to stderr.)
if (notnull($grami{joboption_rsl_wrapperdebug})) {
    $output .= "set -x\nexec &>\Q$stderr\E\nls -la\n";
}
```

```
# All environment variables in existence after sourcing the runtime scripts
# with argument 0 should be set in the wrapper script.
open EXPORTFILE, "<$exportfile" or die "$0: $exportfile: $!\n";
$output .= $_ for <EXPORTFILE>;
close EXPORTFILE;
unlink $exportfile;

# Source runtime scripts with argument 1.
$output .= $setrte1;

# Enable the executable bit for non-preinstalled executables.
if ($real_exe !~ m{^/}) {
    $output .= "chmod +x \Q$real_exe\E\n";
}

# Incomplete job command; arguments may follow.
$output .= "\Q$real_exe\E";

# Add optional arguments to the command line.
if (defined $grami{joboption_arg_1}) {
    for (my $i = 1; defined(my $arg = $grami{"joboption_arg_$i"}); $i++) {
        $output .= $arg ne '' ? " \Q$arg\E" : " ''";
    }
}

# Redirect stdout/stderr.  These variables are always set to something
# (/dev/null if unspecified), so it's safe to unconditionally add these
# redirections.
$output .= " >\Q$stdout\E";
# If we're debugging the wrapper script, we don't do stderr redirection.
if (!notnull($grami{joboption_rsl_wrapperdebug})) {
    if ($stdout eq $stderr) {
        # We're here if stdout and stderr is redirected to the same file.
        # This will happen when (join = yes) in the xRSL.
        $output .= ' 2>&1';
    } else {
        $output .= " 2>\Q$stderr\E";
    }
}

# Always a newline to terminate the job command.
# Preserve the job's exit code.
# Run runtime environment files with argument 2.
$output .= "\n_exitcode=\$?\n$setrte2";

# Delete all remaining files that are not listed in outputFiles.
if (notnull($grami{joboption_rsl_outputfiles})) {
    # The format of this variable is:
    # <filename1><SP>[<checksum1>]<SP>...<SP><filenameN><SP>[<checksumN>]
```

```
        # so split by / / and ignore the odd indexes (the checksums).
        # Also ignores zero length filenames and the gmlog.
        my $i = 0;
        if (!defined $grami{joboption_rsl_gmlog}) {
            # Avoid 'uninitialized' warning when gmlog is unset.
            $grami{joboption_rsl_gmlog} = '';
        }
        my @fileslst = grep { $i++ % 2 == 0 &&
                              $_ ne '' && $_ ne $grami{joboption_rsl_gmlog} }
                       split / /, $grami{joboption_rsl_outputfiles};
        # Quote special chars in filenames and put a ' ' between each name.
        my $files = join ' ', map { quotemeta } @fileslst;
        # Now generate code that removes everything but the requested output.
        # Note that, bashims have been avoided so that there are less strict
        # requirements on /bin/sh on the execute nodes.  (Note that the file
        # utilities used (mkdir, dirname, find, etc.) may still be
        # GNU-centric.  TODO: fix this if we're to support non-x86-Linux.)
        $output .= <<EOF;
# Make a temporary directory, keep.N, where N is chosen for uniqueness.
N=0
while [ -e keep.\$N ]; do
    N=`expr \$N + 1`
done
for i in $files; do
    [ -e "\$i" ] || continue
    destdir="keep.\$N/`dirname "\$i"`"
    mkdir -p "\$destdir"
    mv -f "\$i" "\$destdir"
done
# Wipe out anything that's not moved into keep.N.
find . -mindepth 1 -path ./keep.\$N -prune -o -print0 | xargs -0 rm -rf
# Move the output files back to their correct location.
find keep.\$N -mindepth 1 -maxdepth 1 -exec mv {} . \\;
# Done, now we can remove the keep.N directory.
rmdir keep.\$N
EOF
    }

    # Exit with the job's exit code.
    $output .= "exit \$_exitcode\n";

    # Create the actual shell script from $output.
    open EXE, ">$exewrapper"             or die "$0: creat $exewrapper: $!\n";
    print EXE $output                    or die "$0: write $exewrapper: $!\n";
    close EXE                            or die "$0: close $exewrapper: $!\n";
    chmod 0755, $exewrapper              or die "$0: chmod $exewrapper: $!\n";

    # Log the Condor job submission script in gmlog/errors.
    unless ($debug) {
```

```
        warn "$0: ----- begin wrapper script ($exewrapper) -----\n";
        warn "$0: $_\n" for split /\n/, $output;
        warn "$0: ----- end wrapper script ($exewrapper) -----\n";
    }
}

#
# Create a Condor job description that submits the wrapper script created
# above.  The Condor job description should mirror the xRSL as much as
# possible.
#
sub create_condor_job_description {
    # As above, the job description is created in-memory, so that only one I/O
    # operation has to be done when writing to disk.
    my $output = "Executable = $exewrapper\n" .
                 "Input = $grami{joboption_stdin}\n";

    if ($debug) {
        # Use a bogus name for the logfile if debugging -- it doesn't matter.
        $condor_log = 'job.log';
    } else {
        $condor_log = File::Temp::tempnam($grami{joboption_directory}, 'log.');
    }
    $output .= "Log = $condor_log\n";

    my @requirements;
    if (notnull($grami{joboption_memory})) {
        push @requirements, "Memory >= $grami{joboption_memory}";
    }

    if (notnull($grami{joboption_rsl_disk})) {
        push @requirements, "Disk >= " . $grami{joboption_rsl_disk} * 1024;
    }

    if (notnull($grami{joboption_runtime_0})) {
        for (my $i = 0; notnull(my $r = $grami{"joboption_runtime_$i"}); $i++) {
            $r =~ s/\W/_/g;
            push @requirements, "$r =?= True"
        }
    }

    # This is a custom RSL attribute used for debugging.  If the xRSL contains
    # (machine = foo), the job will only run on machine "foo".
    if (notnull($grami{joboption_rsl_machine})) {
        push @requirements, qq{Machine == "$grami{joboption_rsl_machine}"};
    }

    # TODO: Remove this once there's support for non-x86 CPUs!
    push @requirements, '(Arch == "INTEL" || Arch == "x86_64")';
```

```
if (@requirements) {
    $output .= "Requirements = " . shift @requirements;
    $output .= " && $_" for @requirements;
    $output .= "\n";
}

if (notnull($grami{joboption_rsl_inputfiles})) {
    # TODO: should I change the split pattern to / /, so that it doesn't
    # squeeze together multiple spaces?  If the size.checksum pair is
    # optional, the / / variant must be used.  If not, the ' ' variant
    # is better since it allows for variations in placing whitespace.
    my @tmp = split ' ', $grami{joboption_rsl_inputfiles};
    $output .= "Transfer_input_files = ";
    for (my $i = 0; $i < @tmp; $i += 2) {
        $output .= ',' if $i > 0;
        $output .= $tmp[$i];
    }
    $output .= "\n";
}

if (notnull($grami{joboption_env_0})) {
    $output .= "Environment = ";
    my $first = 1;
    for (my $i = 0; notnull($grami{"joboption_env_$i"}); $i++) {
        $output .= ";" if $i > 0;
        $output .= $grami{"joboption_env_$i"};
    }
    $output .= "\n";
}

if (notnull(my $rank = condor_rank())) {
    $output .= "Rank = $rank\n";
}

$output .= "GetEnv = True\n" .
           "Universe = vanilla\n" .
           "When_to_transfer_output = ON_EXIT\n" .
           "Queue\n";

if ($debug) {
    print $output;
} else {
    my $cmd_fh;
    ($cmd_fh, $cmd_filename) = tempfile('XXXXXXXX',
                                        DIR => $grami{joboption_directory},
                                        SUFFIX => '.cmd');
    print $cmd_fh $output  or die "$0: write $cmd_filename: $!\n";
    close $cmd_fh          or die "$0: close $cmd_filename: $!\n";
```

```
        # Log the Condor job submission script in gmlog/errors.
        warn "$0: ----- begin condor job description ($cmd_filename) -----\n";
        warn "$0: $_\n" for split /\n/, $output;
        warn "$0: ----- end condor job description ($cmd_filename) -----\n";
    }
}

sub submit_condor_job {
    return if $debug;
    chdir $grami{joboption_directory}
      or die "$0: chdir $grami{joboption_directory}: $!\n";

    my $condor_submit_exe = find_condor_submit_exe()
      or die "$0: Couldn't find the condor_submit executable!\n";
    warn "$0: running $condor_submit_exe $cmd_filename\n";
    my $cf = condor_config();
    my $submit_out = `/usr/bin/env - CONDOR_CONFIG=\Q$cf\E /bin/sh -c \\
                      '\Q$condor_submit_exe\E \Q$cmd_filename\E' 2>&1`;
    my $err = $?;
    warn "$0: $_\n" for split /\n/, $submit_out;
    die "$0: condor_submit failed!\n" if $err;

    warn "$0: appending local job id to grami file $gramifile\n";
    my ($localid) = $submit_out =~ /submitted to cluster (\d+)\./;
    open GRAMI, ">>$gramifile"                      or die "$0: $gramifile: $!";
    print GRAMI "joboption_jobid=$localid.condor\n" or die "$0: $gramifile: $!";
    print GRAMI "condor_log=$condor_log\n"          or die "$0: $gramifile: $!";
    close GRAMI                                    or die "$0: $gramifile: $!";
}

sub notnull {
    return defined $_[0] && $_[0] ne '';
}
```

B.3 finish-condor-job

```
#!/bin/sh

progname=$(basename "$0")

# This program assumes the role of /bin/mail, so it's called like this:
# /bin/mail -s '[Condor] Condor Job <job-ID>' <email-address>
# We extract the job-ID from the second argument.
#
# NOTE: The format of the email message is, unlike the job log, not guaranteed
```

```
# to remain unchanged in future versions of Condor, but since we need the job
# id to locate the log file, there's no way around this. :-(
lrmsid=${2##*Condor Job }
lrmsid=${lrmsid%%.*}.condor

: ${NORDUGRID_CONFIG:=/etc/nordugrid.conf}
if [[ ! -r $NORDUGRID_CONFIG ]]; then
    echo "$progname: Couldn't find nordugrid.conf!" >&2
    exit 1
fi

# Set variable "controldir" from GM config.
eval "$(grep '^[[:blank:]]*controldir=' "$NORDUGRID_CONFIG")"

# Find the proper GRAMI file.
grami=$(grep -l "^joboption_jobid=$lrmsid$" "$controldir"/job.*.grami)

if [[ ! -f $grami ]]; then
    echo "No GRAMI file for job $lrmsid could be found." >&2
    exit 1
fi

# Logfile used by Grid Manager.
gmlog=$controldir/$(basename "$grami" .grami).errors

# IMPORTANT: Never change the format of this line!
# It is used in LRMS_Condor.pm to delimit job info.
echo "----- starting $progname -----" >>"$gmlog"

# Find the Condor log.
condor_log=$(sed -n 's/^condor_log=\(.*\)/\1/p' "$grami")

# Use /dev/null if we couldn't find the log.  Should never happen.
if [[ ! -f $condor_log ]]; then
    echo "$progname: couldn't find Condor log file ($condor_log)"
    echo "$progname: using /dev/null as log file"
    condor_log=/dev/null
fi >>"$gmlog" 2>&1

mbody=/tmp/mailbody.$$
trap "rm -f $mbody" EXIT

# Dump mail body and Condor log into gmlog.
{
    cat >$mbody || echo "$progname: failed to write $mbody"
    echo "$progname: ----- begin condor job completion message -----"
    sed "s/^/$progname: /" $mbody
    echo "$progname: ----- end condor job completion message -----"
    echo "$progname: ----- begin condor log ($condor_log) -----"
```

```
    sed "s/^/$progname: /" "$condor_log"
    echo "$progname: ----- end condor log ($condor_log) -----"
} >>"$gmlog" 2>&1

echo "$progname: searching for exit code in $condor_log"
failure_reason=
i=0
maxtries=4
while ((i < maxtries)); do
    # Extract the program exit code from the Condor log.
    exitcode=$(sed -n '/Normal termination/{s/.*value \([0-9]*\).*/\1/p;q;}' \
                "$condor_log")

    if [[ -z $exitcode ]]; then
        exitcode=$(sed -n \
                    '/Abnormal termination/{s/.*signal \([0-9]*\).*/\1/p;q;}' \
                    "$condor_log")
        [[ $exitcode ]] && ((exitcode = exitcode + 128))
    fi

    if [[ $exitcode ]]; then
        break
    fi

    ((i = i + 1))
    echo "$progname: failed to get exit code (attempt $i/$maxtries)"
    sleep 15
done >>"$gmlog" 2>&1

if ((i == maxtries)); then
    echo "$progname: giving up on log; trying condor job completion message"
    exitcode=$(sed -n 's/.*has exited.*with status \([0-9]*\).*/\1/p' $mbody)
    if [[ -z $exitcode ]]; then
        exitcode=$(sed -n 's/.*has died on signal \([0-9]*\).*/\1/p' $mbody)
        [[ $exitcode ]] && ((exitcode = exitcode + 128))
    fi
fi >>"$gmlog" 2>&1

if [[ $exitcode ]]; then
    echo "$progname: job $lrmsid finished with status $exitcode"
    sessiondir=$(sed -n 's/^sessiondir=\(.*\)/\1/p' \
                    "$controldir"/$(basename "$grami" .grami).local)
    if [[ $sessiondir ]]; then
        echo "exitcode=$exitcode" >>"$sessiondir.diag"
    fi
    if [[ $exitcode != 0 ]]; then
        failure_reason='Job finished with non-zero exit code'
    fi
else
```

```
    echo "$progname: failed to get exit code; using 255 as fallback"
    failure_reason='Failed to get exit code of job'
    exitcode=255
fi >>"$gmlog" 2>&1

# Write exit code in job.ID.lrms_done to signal that we're done.
echo "$exitcode $failure_reason" \
              >"$controldir"/$(basename "$grami" .grami).lrms_done

echo "----- exiting $progname -----" >>"$gmlog"
exit 0
```

B.4 cancel-condor-job

```
#!/bin/sh

progname=$(basename "$0")
echo "----- starting $progname -----" >&2

if [[ ! -f $1 ]]; then
    echo 'No GRAMI file' >&2
    exit 1
fi

#
# Try to extract condor_location from nordugrid.conf.
#
: ${NORDUGRID_CONFIG:=/etc/nordugrid.conf}
if [[ -r $NORDUGRID_CONFIG ]]; then
    eval "$(egrep '^[[:blank:]]*condor_(config|location)=' "$NORDUGRID_CONFIG")"
    condor_rm=$condor_location/bin/condor_rm
    if [[ -x $condor_rm ]]; then
        echo "$progname: found $condor_rm using $NORDUGRID_CONFIG"
        echo "$progname: found $condor_config using $NORDUGRID_CONFIG"
    fi >&2
fi

#
# Search PATH for condor_rm.
#
if [[ ! -x $condor_rm ]]; then
    IFS=:
    condor_rm=
    for i in $PATH; do
        if [[ -x "$i"/condor_rm ]]; then
            condor_rm=$i/condor_rm
```

```
            echo "$progname: found $condor_rm using PATH" >&2
        fi
    done
    unset IFS
fi

#
# If the file /etc/sysconfig/condor exists, see if it defines the variable
# CONDOR_LOCATION and look for condor_rm in $CONDOR_LOCATION/bin.
#
if [[ ! -x $condor_rm ]]; then
    if [[ -f /etc/sysconfig/condor ]]; then
        . /etc/sysconfig/condor
        condor_rm=$CONDOR_LOCATION/bin/condor_rm
    fi
    if [[ -x $condor_rm ]]; then
        echo "$progname:" \
             "found $condor_rm from /etc/sysconfig/condor" >&2
    fi
fi

#
# If all else fails, use the optional SUID root program condor_master_location
# to search /proc/<pid>/exe for links to the condor_master program, and use its
# directory to find condor_rm.
#
if [[ ! -x $condor_rm ]]; then
    progdir=$(cd "$(dirname "$0")"; pwd)
    if [[ -x $progdir/condor_master_location ]]; then
        condor_bindir=$(cd "$("$progdir"/condor_master_location)"/../bin; pwd)
        condor_rm=$condor_bindir/condor_rm
    fi

    if [[ -x $condor_rm ]]; then
        echo "$progname:" \
             "found condor_rm from condor_master_location ($condor_rm)" >&2
    fi
fi

if [[ ! -x $condor_rm ]]; then
    echo "$progname: could not find condor_rm!" >&2
    exit 1
fi

if [[ ! -r $condor_config ]]; then
    condor_config=$(dirname "$condor_rm")/../etc/condor_config
    [[ -r $condor_config ]] && echo "$progname:" \
        "$condor_config found above $condor_rm" >&2
fi
```

```
if [[ ! -r $condor_config ]]; then
    condor_config=$CONDOR_CONFIG
    [[ -r $condor_config ]] && echo "$progname:" \
        "$condor_config found using CONDOR_CONFIG environment variable" >&2
fi

if [[ ! -r $condor_config ]]; then
    echo "$progname: could not find Condor configuration file!" >&2
    exit 1
fi

eval "$(grep '^joboption_jobid=' "$1")"
echo "$progname: canceling job $joboption_jobid with condor_rm..." >&2
export CONDOR_CONFIG=$condor_config
$condor_rm ${joboption_jobid%.condor} >&2
echo "----- exiting cancel-condor-job -----" >&2
exit 0
```

B.5 cluster-condor.pl

```
#!/usr/bin/perl
use lib "$ENV{PWD}/";
use lib "$ENV{NORDUGRID_LOCATION}/libexec";
use lib $ENV{NORDUGRID_LOCATION}."/libexec/nordugrid";
use lib "/opt/nordugrid/libexec";
use infosys_shared;
use InfosysCluster qw(%config &clusterldif);
use LRMS_Condor;
#use warnings;

# populates the nordugrid-cluster LDAP entry with real GM and config
# The GM interface is factored into the InfosysCluster module.


#<LRMSPORT
# automatically determine the LRMS name(flavour) and version
$lrms_type = nordugrid_cluster_lrms_type();
$lrms_version = nordugrid_cluster_lrms_version();
#LRMSPORT>


#<LRMSPORT
# Calculate the totalcpus & cpudistribution & number of used cpus
# by collecting information from the LRMS
# Consult the schema for the syntax of the $cpudistribution
```

```
# If $dedicated_node_string is set, nondedicated nodes, which are not marked
# with $dedicated_node_string should be skipped
$totalcpus = nordugrid_cluster_totalcpus();
$usedcpus = nordugrid_cluster_usedcpus();
$cpudistribution = nordugrid_cluster_cpudistribution();
#LRMSPORT>


#<LRMSPORT
# Calculate $lrms_queued, the number of jobs (both grid and nongrid)
# being queued in the LRMS
$lrms_queued = lrms_get_queued();
#LRMSPORT>


#<LRMSPORT
# Calculate the total number of jobs in the LRMS (running, queued, hold, being
# in any LRMS state; submitted by both grid and nongrid users): $lrms_total
$lrms_total = lrms_get_total();
#LRMSPORT>


# generating the cluster ldif
&clusterldif;


# some performance test
my $runtime =  time - $^T;
if ($config{loglevel} == 2) {
   &infosys_shared::write_log("execution time: $runtime");
}
elsif ($config{loglevel} == 1 and $runtime >= 4 ) {
   &infosys_shared::write_log("SLOW script: $runtime");
}


exit;
```

B.6 queue+jobs+users-condor.pl

```
#!/usr/bin/perl
use lib "$ENV{PWD}/";
use lib "$ENV{NORDUGRID_LOCATION}/libexec";
use lib $ENV{NORDUGRID_LOCATION}."/libexec/nordugrid";
use lib "/opt/nordugrid/libexec";
use infosys_shared;
```

```
use InfosysQJU qw(%config %hoh_gmjobs %users &queueldif &jobsldif &usersldif);
use LRMS_Condor;
#use warnings;


# generates everything under the nordugrid-queue-name=xz,
# creates the queue, job & user entries.
# Real GM and config values are used, but dummy values
# are set for LRMS-dependent  attributes (13, 66 or "dummy").
# The GM interface is factored into the InfosysQJU module.
# Use this template to port the infosys.
# Don't use 'my xyz' for the interfacing GLOBAL variables!


$timestamp = time;


#<LRMSPORT
# read the queue info such as limits, authorized users' list, etc from the LRMS
# "queue" command into %queue_info and  @acl_users (list of LRMS-authorized
# unix users, undefined list means every unix user is LRMS authorized)

# queue limits, setting dummy values
# don't set the values if they are not relevant to the LRMS
# nordugrid-queue-maxrunning
$queue_info{"max_running"} = nordugrid_queue_maxrunning();
# nordugrid-queue-maxinqueue
$queue_info{"max_queuable"} = nordugrid_queue_maxqueuable();
# nordugrid-queue-maxuserrun
$queue_info{"max_user_run"} = nordugrid_queue_maxuserrun();
# nordugrid-queue-maxcputime (in minutes)
$queue_info{"max.cput"} = nordugrid_queue_maxcputime();
# nordugrid-queue-mincput (in minutes)
$queue_info{"min.cput"} = nordugrid_queue_mincputime();
# nordugrid-queue-defaultcputime (in minutes)
$queue_info{"default.cput"} = nordugrid_queue_defaultcputime();

#LRMSPORT>



#<LRMSPORT
# read the job info from LRMS to a hash of hashes %hoh_lrmsjobs
# The LRMS jobid is the key of the hash, it MUST match the one stored
# in the GM's job.xx.local file as 'localid'
# determine the queue rank of a LRMS queuing job
# $hoh_lrmsjobs{$jobid}{"rank"} = 1

for (lrms_get_localids_running()) {
```

```
    $hoh_lrmsjobs{$_}{'job_state'} =
        lrms_get_job_status($_, $config{controldir});
    $hoh_lrmsjobs{$_}{'rank'} = nordugrid_job_queuerank($_);
    $hoh_lrmsjobs{$_}{'used.mem'} = nordugrid_job_usedmem($_);
    $hoh_lrmsjobs{$_}{'used.walltime'} = nordugrid_job_usedwalltime($_);
    $hoh_lrmsjobs{$_}{'used.cput'} = nordugrid_job_usedcputime($_);
    $hoh_lrmsjobs{$_}{'req.cput'} = nordugrid_job_reqcput($_);
    $hoh_lrmsjobs{$_}{'exec_host'} = lrms_get_job_executionnodes($_);
    $hoh_lrmsjobs{$_}{'comment'} = nordugrid_job_lrmscomment($_);
}

lrms_get_jobinfo_logged(\%hoh_gmjobs, $config{controldir});

#LRMSPORT>


#<LRMSPORT
# scan the LRMS and count the number of running jobs with their multiciplity
# and the number or queued jobs
# count the jobs per unix users and the number of queueing jobs per users
# These values may be needed for the calculation of the per user freecpus
# useful variables: $user_jobs_queued{$username}, $user_jobs_running{$username}
# required values for the queue entry: $totalrunning_in_the_queue, $totalqueued
# nordugrid-queue-running
$totalrunning_in_the_queue = nordugrid_queue_running();
# nordugrid-queue-queued
$totalqueued = nordugrid_queue_queued();
#LRMSPORT>



#<LRMSPORT
# scan the LRMS in order to find out the $totalcpus and the $usedcpus
# Take into account the $dedicated_node_string or the $queue_node_string
# for distinguishing grid and non-grid enabled nodes (skip nonededicated nodes).
# The $totalcpus and the $usedcpus may be needed for the calculation of the per
# user freecpus.
# TODO ???
#LRMSPORT>



if ($config{loglevel} == 2) {
   my $runtime = time - $timestamp;
   &infosys_shared::write_log("LRMS I/O time: $runtime");
}


#<LRMSPORT
```

```
# Count the running and queued GRID jobs in the LRMS (a subset of all LRMS jobs)
# running jobs are counted with their multiciplity
# you can loop through the hoh_gmjobs and use the
# "dummy13"=$hoh_gmjobs{$ID}{"localid"} and the
# $hoh_lrmsjobs{"dummy13"}{"job_state"} to find queuing and running grid jobs
# in the LRMS
# Provide the $gridrunning, $gridqueued numbers
#nordugrid-queue-gridrunning
$gridrunning = nordugrid_queue_gridrunning();
#nordugrid-queue-gridqueued
$gridqueued = nordugrid_queue_gridqueued();
#LRMSPORT>



#<LRMSPORT
# nordugrid-queue-status
# determine the $lrms_queue_status from the LRMS
# set it to 'active' if the queue can accept jobs
$lrms_queue_status = nordugrid_queue_status();
#LRMSPORT>


# loop over the mapped local unix users and determine the LRMS-dependent
# freecpus and queuelength for each of them.
# Use the %users hash which contains the grid user -> local unix user
# mapping info:  $users{Grid_User_SN}=mapped_unix_id
# Take into account queue limits (cpu & time), physically available cpus
# (values determined above) or use LRMS specific method
# Store the result in two hashes, take care of the syntax of the freecpus:
# cpus[:minutes] here the :minutes part is optional and used for specifying
# time limit on CPUs.
# $freecpus_for_mapped_users{unix_username}="13:33"
# $user_jobs_queued{$unix_username}=13

for (values %users) {
    #nordugrid-authuser-freecpus
    $users_freecpus{$_} = nordugrid_authuser_freecpus();
    #nordugrid-authuser-queuelength
    $users_jobs_queued{$_} = nordugrid_authuser_queuelength();
}

#LRMSPORT>

&queueldif;


&jobsldif;
```

```
&usersldif;



my $runtime =  time - $^T;
if ($config{loglevel} == 2) {
   &infosys_shared::write_log("execution time: $runtime");
}
elsif ($config{loglevel} = 1 and $runtime >= 4 ) {
   &infosys_shared::write_log("SLOW script: $runtime");
}
```

References

[1] *The ATLAS Experiment*
⟨http://atlasexperiment.org/⟩

[2] *CERN openlab for DataGrid applications*, Third annual report: June 2003–June 2004, IT Department CERN (2004).

[3] *The Large Hadron Collider*
⟨http://athome.web.cern.ch/athome/LHC/lhc.html⟩

[4] *ATLAS Data Challenges*
⟨http://atlasinfo.cern.ch/Atlas/GROUPS/SOFTWARE/DC/doc/AtlasDCs.pdf⟩

[5] *GEANT applications*
⟨http://wwwasdoc.web.cern.ch/wwwasdoc/geant_html3/node9.html⟩

[6] S. George, A. Lowe et. al., *Architecture of the ATLAS High Level Triggers Event Selection Software*, ATL-DAQ-2003-046.

[7] S. Fields, *Hunting for Wasted Computing Power*
⟨http://www.cs.wisc.edu/condor/doc/WiscIdea.html⟩

[8] *dhcpcd – DHCP client daemon*
⟨http://www.phystech.com/download/dhcpcd.html⟩

[9] *Condor Manual: Current Limitations*
⟨http://www.cs.wisc.edu/condor/manual/v6.6/1_4Current_Limitations.html⟩

[10] U. F. Mayer, *Linux/Unix nbench*
⟨http://www.tux.org/~mayer/linux/bmark.html⟩

[11] *BYTEmark Frequently Asked Questions*
⟨http://www.byte.com/bmark/faqbmark.htm⟩

[12] *Condor Manual: Security in Condor*
⟨http://www.cs.wisc.edu/condor/manual/v6.6/3_7Security_In.html⟩

[13] *PVM: Parallel Virtual Machine*
⟨http://www.epm.ornl.gov/pvm/⟩

[14] *Condor Manual: PVM Applications*
⟨http://www.cs.wisc.edu/condor/manual/v6.6/2_9PVM_Applications.html⟩

[15] S. Hebert, *Message Passing Interface (MPI) FAQ*
⟨http://www.faqs.org/faqs/mpi-faq/⟩

[16] *MPI: A Message-Passing Interface Standard*
⟨http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html⟩

[17] *Condor Manual: MPI Applications*
⟨http://www.cs.wisc.edu/condor/manual/v6.6/2_10MPI_Applications.html⟩

[18] *MPICH – A Portable Implementation of MPI*
⟨http://www-unix.mcs.anl.gov/mpi/mpich/⟩

[19] *The Globus Alliance*
⟨http://www.globus.org/⟩

[20] I. Foster, C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit*, Intl J. Supercomputer Applications, 11(2):115–128, 1997

[21] *NorduGrid middleware, the Advanced Resource Connector*
⟨http://www.nordugrid.org/middleware/⟩

[22] *The Norwegian GRID Project*
⟨http://www.norgrid.no/⟩

[23] *The Swegrid Organization*
⟨http://www.swegrid.se/⟩

[24] *Dansk Center for Grid Computing*
⟨http://www.dcgc.dk/⟩

[25] *Nordic Data Grid Facility*
⟨http://www.ndgf.org/⟩

[26] *The NorduGrid Collaboration*
⟨http://www.nordugrid.org/about.html⟩

[27] *Portable Batch Systems (PBS)*
⟨http://www.amestechnology.org/ames/technologyinformation.asp?ID=ARC-15006⟩

[28] A. Konstantinov, *The NorduGrid Grid Manager and GridFTP Server: Description and Administrator's Manual*
⟨http://cvs.nordugrid.org/cgi-bin/cvsweb/~checkout~/nordugrid/doc/gm/GM.pdf?rev=HEAD&content-type=application/pdf⟩

[29] B. Kónya, *The NorduGrid Information System*
⟨http://www.nordugrid.org/documents/ng-infosys.pdf⟩

[30] *The ATLAS Data Challenges*
⟨http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/DC/index.html⟩

[31] *Condor Configuration Instructions for NorduGrid/ARC Sites*
⟨http://www.nordugrid.org/documents/condor-config.html⟩

[32] *ATLAS DC2 job statistics at NorduGrid/ARC facilities*
⟨http://www.nordugrid.org/monitor/atlas/dc2stats.php⟩

[33] B. R. Martin, G. Shaw, *Particle Physics*, John Wiley & Sons, ISBN 0-471-97252-5 (1997).

[34] I. Foster, *The Grid: A New Infrastructure for 21st Century Science*, Physics Today Vol. 55 #2, p. 42, 2002

[35] R. P. Mount, *Workshop on New visions for Large-Scale Networks: Research and Applications*, Stanford Linear Accelerator Center.

[36] O. Smirnova, *xRSL (Extended Resource Specification Language)*
⟨http://www.nordugrid.org/documents/xrsl.pdf⟩