GRID COMPUTING—A HANDS-ON EXPERIENCE

Leif Nixon and Patrick Norman

The Compass portable was the first computer of the Grid company (1984).

List of exercises:

1. Introduction to Nordugrid/Swegrid and ARC
2. Python programming on the grid
3. Program package development
4. Job scheduler
5. Watchdog

Coordinator:
Leif Nixon
National Supercomputer Centre,
Linköping University,
SE-581 83 Linköping, Sweden
e-mail: nixon@nsc.liu.se

Revision 1.22, 2005/01/05 10:38:41

Contents

A Introduction **3**
A.1 Computing Grids . . . 3
A.2 Swegrid and ARC . . . 3
A.3 Motivation . . . 3

B Notes on the exercises **5**

C Notes on the lab environment **5**

D Exercise 1: Introduction to Nordugrid/Swegrid and ARC **6**
D.1 Relevant documentation . . . 6
D.2 Authentication and proxies . . . 6
D.3 Advanced Resource Connector . . . 6
D.4 Getting started . . . 7
D.5 Submitting a Simple Job . . . 7
D.6 Monitoring jobs . . . 8
D.7 More Examples . . . 9

E Exercise 2: Python programming on the grid **12**
E.1 Relevant documentation . . . 12
E.2 Monte Carlo integration . . . 12
E.2.1 Side note: quasi-random numbers . . . 13
E.3 Exercises . . . 13

F Exercise 3: Program package development **14**
F.1 Relevant documentation . . . 14
F.2 Runtime environments and Program packages . . . 14
F.3 Random walk problem . . . 14
F.4 The `make` utility program . . . 15
F.5 Exercises . . . 16

G Exercise 4: Job scheduler **17**
G.1 Relevant documentation . . . 17
G.2 Scheduling of large number of tasks with distributed control . . . 17
G.3 Exercises . . . 20

H Exercise 5: Watchdog **21**
H.1 Relevant documentation . . . 21
H.2 Scheduling of large number of tasks with local control . . . 21
H.3 Exercises . . . 21

I Appendix: A basic random walk program **23**

A Introduction

A.1 Computing Grids

The exercises of this course will introduce you to the use of *computing grids*. A computing grid may in this context be considered as:[1]

> Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed "autonomous" resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements.

This definition of a grid is quite abstract and provide little insight into the problem of accessing the grid and to get your application to run on the grid resources. The level of abstractness is intentional, however, and makes the concept of a grid much wider than the concept of a cluster. If one compares a grid with a cluster, the main differences lies in the way the resources are managed. In case of clusters, the resource allocation is performed by a centralized resource manager and all nodes cooperatively work together as a single unified resource, whereas, in case of grids, each node has its own resource manager and does not aim at providing a single system view. It is therefore clear that a *middleware* is needed to bind together the nodes of the grid. The middleware needs to be concerned with the aspects mentioned above (availability, capability, performance, cost, and users' quality-of-service requirements) when it handles the brokering between the grid and the client. The middleware must also manage data and resources, monitor jobs, and provide other information services.

A.2 Swegrid and ARC

During the exercises you will get acquainted with a particular grid and a particular middleware. The abstract discussion in the previous section will thereby turn most concrete, and thereby lose some of its generality.

The grid we will work with is named *Swegrid*[2]. Swegrid is a Swedish national computational resource, consisting of 600 computers in six clusters at six different sites across Sweden, and the sites are connected through the high-performance GigaSunet network. This grid is relatively homogeneous with respect to its nodes and does not allow for parallel job scheduling across its nodes (clusters).

We will work with an open source middleware named *Advanced Resource Connector* (ARC), which is maintained and developed by the NorduGrid Collaboration[3]. Pre-built binaries are available for about a dozen Linux distributions and can be downloaded from the NorduGrid homepage.

We recommend the "Swegrid–User's guide"[4] for an introduction to using Swegrid with the ARC middleware. This documentation will also be instrumental for the exercises in this course.

A.3 Motivation

The installation of Swegrid was the starting point of a new phase in Swedish High Performance Computing (HPC). It is not clear what the situation in Swedish HPC will

[1] `http://www.gridcomputing.com`
[2] `http://www.swegrid.se`
[3] `http://www.nordugrid.org`
[4] `http://www.swegrid.se/downloads/swegrid_manual.pdf`

be ten years from today, but there is no reason not to believe that grid computing will play an important role in that future. As a student of the national graduate school in scientific computing you are likely to be among the users requiring HPC resources in your future work. The goal of this hands-on sessions of this course is to teach you to master the grid environment from a user's perspective in order for you to make the most out of your scientific applications on the grid.

It may be argued that the lifetime of Swegrid as of today is short, and that the middleware of coming grid generations may not be ARC. As true as that may be, it is still our belief that the generic notions will remain and that these exercises serve as an illustration of some general issues involved with grid computing.

B Notes on the exercises

You can do the exercises on your own or in pairs.

You don't need to document exercise 1; just reflect carefully on the questions. Feel free to discuss them with the instructor or other students.

Exercises 2–5 should be documented with code listings, example runs, etc, and handed in to the instructor, electronically or on paper. Your programs should preferably also be demonstrated live.

C Notes on the lab environment

Your desktop systems will be Sun SunRay thin clients running the Solaris operating system. The default language in the desktop environment is Swedish. To get an English environment instead, select the Options→Language menu on the login screen and choose the entry `en_US.ISO8859-1`. (You can of course select any language of your choice.) There is also a choice of two different desktop environments, Gnome and CDE. Choose whichever you think you will be most comfortable with. You can always change your selection of desktop environment in the Options→Session menu on the login screen.

The computer labs SU00 and SU01 are reserved for the course all weekdays between 13:00 and 21:00. There are no reservations during the weekend, just grab a free SunRay in any SU lab.

The actual work will be performed on an NSC system, `login-3.monolith.nsc.liu.se`. This machine is accessible from the SunRay system (and from the general Internet) by ordinary ssh login.

Please note that you have separate account names on the SunRay system and `login-3`.

D Exercise 1: Introduction to Nordugrid/Swegrid and ARC

Acknowledgment: Most of this exercise has been shamelessly stolen from Arto Teräs' and Juha Lehto's Nordugrid tutorial.

D.1 Relevant documentation

1. The NorduGrid User Guide
   `http://www.nordugrid.org/documents/userguide.pdf`
2. "Swegrid–User's guide"
   `http://www.swegrid.se/downloads/swegrid_manual.pdf`.
3. NorduGrid homepage
   `http://www.nordugrid.org`
4. "Learning Python" by Lutz and Ascher, O'Reilly (2003).
5. Python 2.3 Library Reference
   `http://www.python.org/doc/2.3.4/lib/lib.html`

D.2 Authentication and proxies

Once you have acquired a grid certificate and installed the ARC software you are ready to access the grid. Before doing so, however, you need to be able to authenticate yourself, a process that is handled with time-limited proxy certificates. The creation of a proxy is done with the ARC command `grid-proxy-init`. You will be prompted for a password that should match the one your private key is encrypted with.

The dictionary definition of the word proxy is "a document giving authority or power to act for another". In our case it gives your jobs the authority to act as you on the grid resources.

In this course we are using temporary certificates signed by a temporary Certificate Authority, used for this course only. This limits the number of available clusters, as these tutorial identities and certificates do not belong to any generally authorized Virtual Organization. Also, the usefulness of the Grid Monitor web interface is somewhat limited for these tutorial identities.

The certificates and corresponding private keys have already been placed in your home directory (the files are located in the directory `.globus`) on `login-3`. The password of the private key is "`ngssc`".

D.3 Advanced Resource Connector

The grid middleware provides the user with a number of commands to access and interact with the grid. Swegrid is based on the Advanced Resource Connector (ARC) middleware, and the command names for ARC is given in Table 1. Documentation of the ARC commands are given via the regular `man` command.

Table 1: Some of the more important ARC commands.

ngcat	ngls	ngsync
ngclean	ngsub	ngrenew
ngcopy	ngremove	ngtest
ngget	ngresub	
ngkill	ngstat	

D.4 Getting started

First of all, log in to `login-3` and get a copy of the example files by typing[5]

```
$ tar xvzf ~nixon/exercises.tar.gz
```

The example files for this exercise are located in `exercises/ex1`.

- Print your certificate in text form by typing `grid-cert-info`. What is your identity in the grid? Who has signed the certificate? How long is it valid?
- Logging in to the grid actually means creating a temporary access token called a "proxy certificate". Do this by typing `grid-proxy-init` and specifying the password of the private key. Print information about your proxy by typing `grid-proxy-info`. How long is it valid?

D.5 Submitting a Simple Job

Take a look at the file `hellogrid.sh`. It is a simple shell script which writes "Hello Grid" on the standard output and sleeps for a while before returning. You can try to run it locally by typing

```
$ ./hellogrid.sh
```

The job description file to submit this script to the grid is `hellogrid.xrsl`:

```
&(executable=hellogrid.sh)
 (stdout=hello.out)
 (stderr=hello.err)
 (gmlog=gridlog)
 (architecture=i686)
 (cputime=10)
 (memory=32)
 (disk=1)
```

Try to submit the job to NorduGrid:

```
$ ngsub -f hellogrid.xrsl
```

This may take a while to complete as the client first contacts the root information server, asks for clusters connected at the moment and then queries all the available clusters for their attributes etc. Then it starts preparing a job by transferring the input files. When the job is submitted, you should receive a message similar to

[5] The `exercises.tar.gz` file is also available from the course homepage at `http://www.nsc.liu.se/ngssc-grid-05`.

```
Job submitted with jobid gsiftp://benedict.aau.dk:2811/jobs/2837896291031006429
```

In this case, the job was submitted to `benedict.aau.dk` in Denmark and the URL `gsiftp://benedict.aau.dk:2811/jobs/2837896291031006429` is the reference to the job. (The last part is a session directory chosen randomly by the target system.) It is possible to check the status of the job using the ngstat command:

```
$ ngstat gsiftp://benedict.aau.dk:2811/jobs/2837896291031006429
Job gsiftp://benedict.aau.dk:2811/jobs/2837896291031006429
 Jobname: hellogrid
 Status: FINISHED 2004-03-29 16:15:18
```

In this case the job has been successfully completed. Other stages that the job may be in are described in the NorduGrid User Guide. Retrieve the results by typing

```
$ ngget gsiftp://benedict.aau.dk:2811/jobs/2837896291031006429
```

This downloads the result files and some statistics in the directory `2837896291031006429`. Take a look at the output (files stdout and stderr and diag file in the gridlog directory. What can you see?

Note that we made no reference to which cluster the job should go. If you would like to specify the cluster (or exclude some), it can be described in the xRSL file or on the command line:

```
$ ngsub -f hellogrid.xrsl -c datagrid3.csc.fi
```

- Try submitting the job with the command `ngsub -f hellogrid.xrsl -d 1` to see more information about the submission process. Even more info is available with `-d 2`. If a cluster is misbehaving and causing time-outs, the `-t` option for specifying a faster time-out is useful.
- Specify a job name by adding the line `(jobname=hellogrid_your_name)` to the file `hellogrid.xrsl`. Submit the job again. Now you can refer to the job with the name instead of the job ID when using the `ngstat` and `ngget` commands.
- Submit some more jobs and try the commands `ngkill` and `ngclean`.
- Specify three alternative clusters as accepted targets in the `hellogrid.xrsl` file. Try submitting the job. (Hint: Use the "`cluster`" attribute, see the User Guide for details.)
- Add a "`notify`" attribute in the xRSL file to receive email notifications of job status changes. See the User Guide for details.

D.6 Monitoring jobs

The command `ngstat` was introduced in the previous section. Take a look at available options by typing

```
$ ngstat -h
```

The status of jobs can also be seen via the graphical Grid Monitor, which can be found on the NorduGrid web site `http://www.nordugrid.org`. Click on the "Grid Monitor" link at the top of the page.

The main view of the monitor shows currently connected resources. Most of the elements are links, clicking on them opens a new window giving more information of that particular resource. For example, click on a cluster name to view more information about that cluster, on the process bar to view more information about jobs running on the cluster, and the "All users" icon to view a list of users authorized to run jobs in NorduGrid.

The jobs belonging to a certain user can be monitored through the Grid Monitor. To see what kind of information is available, you can for example select the NorduGrid Virtual Organization (click on the "VOs" icon and then "NorduGrid members") and click on the names of the instructors.

- What is the processor type in the Monolith cluster? How much memory is installed in the nodes?
- Which version of NorduGrid software and which runtime environments are installed in the Benedict cluster in Denmark?
- On which clusters is the user Balazs Konya (Balazs is a member of the NorduGrid VO) authorized run jobs?
- Which Storage Elements have more than a terabyte of free disk space?
- Using the "Match yourself" dialog, it is possible to see the amount of resources available for any user, including the tutorial identities which are not part of Virtual Organizations. In the "search" dialog, select "Resource/object: User" and click "Next". In the first row of the following dialog, select "Name", "~" (tilde) and fill in your name (spelled as in your certificate) in the last field. Then select the types of resources you want to see on the subsequent rows, for example "Free CPUs" and "Free disk space".

D.7 More Examples

Running a real application

This example demonstrates how to run a simple serial computation on the grid. The application is a first-principles real-space electronic structure program calculating the electronic structure of the CH_4 molecule.[6] In this case the (statically linked) executable is submitted to the grid as one of the job input files and no reference to Runtime Environments (software packages installed on the target cluster) is required. Basically we request a single i386 compatible PC. Go to the directory containing the material:

```
$ cd rspace
$ ls CH4_LUCKY.xrsl
INPUT  potentials  rspace-0.81_i386-linux_SERIAL
```

The job description is in the file `CH4_LUCKY.xrsl`:

[6] Thanks to Tuomas Torsti for providing the example.

```
$ cat CH4_LUCKY.xrsl
&(executable=rspace-0.81_i386-linux_SERIAL)
 (JobName=CH4_LUCKY)
 (inputFiles=(INPUT "")
             (potentials/C "")
             (potentials/H ""))
 (outputFiles=(energies "")
              (forces "")
              (WAVES_1 "")
              (POTENTIAL ""))
 (CpuTime=10)
 (memory=64)
 (disk=10)
 (stdout=stdout.txt)
 (stderr=stderr.txt)
 (gmlog=debugdir)
 (|(architecture=i386) (architecture=i686))
```

The first line defines the name of the executable. If it is not specified in the list of input files, it is automatically appended to it. Edit the job name from CH4_LUCKY to CH4_LUCKY_YOUR_FIRST_NAME so you more easily can separate out your job from the other students' jobs. Read from the User Guide how the location of the input and output files is resolved. That can be tricky with all the available access schemes. In the last lines some of the requirements for the job are specified, so that the client can select a suitable resource.

Submit the job!

```
$ ngsub -f CH4_LUCKY.xrsl
INPUT->INPUT 1 s: 0 kB 0 kB/s 0 kB/s ...
rspace-0.81_i386-linux_SERIAL->rspace-0.81_i386-linux_SERIAL 1 s: ...
rspace-0.81_i386-linux_SERIAL->rspace-0.81_i386-linux_SERIAL 2 s: ...
C->C 1 s: 0 kB 0 kB/s 0 kB/s ...
C->C 2 s: 64 kB 31 kB/s 32 kB/s ...
H->H 1 s: 0 kB 0 kB/s 0 kB/s ...
Job submitted with jobid gsiftp://ingvar.nsc.liu.se:2811/jobs/7009965451436415513
```

Monitor the job with ngstat and when it is finished, fetch the results[7] with ngget. The default time that the output files are kept on the remote site is 24 hours. In practice one would like to transfer the results back to some storage server (Storage Element, SE) automatically after the completion of the job. That's achieved by specifying the target location in the xRSL file. The files can then be moved between different SEs using for example ngcopy (see the User Guide for details). Interactive FTP clients with Grid authentication are also available.

- A Storage Element is available at storage2.bluesmoke.nsc.liu.se. Log in to the storage element by typing gsincftp storage2.bluesmoke.nsc.liu.se and create a directory named se3/ngssc/_your_name_ there. Then modify the CH4_LUCKY.xrsl file so that the output files are uploaded to the storage element (again, see the User Guide for details). Submit the job using ngsub and fetch results using ngget when it is completed. Now ngget should only fetch log files,

[7] These may or may not be intelligible, depending on whether you are a quantum physicist or not.

standard output and standard error. Log in to the storage element again to get the actual result files.

E Exercise 2: Python programming on the grid

E.1 Relevant documentation

1. "Swegrid–User's guide"
   `http://www.swegrid.se/downloads/swegrid_manual.pdf`.

2. The NorduGrid User Guide
   `http://www.nordugrid.org/documents/userguide.pdf`

3. "Learning Python" by Lutz and Ascher, O'Reilly (2003).

4. Python 2.3 Library Reference
   `http://www.python.org/doc/2.3.4/lib/lib.html`

E.2 Monte Carlo integration

To get a suitable problem to work with, we will look at using Monte Carlo to evaluate definite integrals. Suppose we wish to evaluate the integral

$$I = \int_a^b g(x)dx, \tag{1}$$

where a and b are the limits of integration. If we introduce a bounding box that encloses the function $g(x)$, then the integral of $g(x)$ can be understood to be the fraction of the bounding box that is also within $g(x)$. So if we choose a point at random (uniform distribution) within the bounding box, the probability that the point is within $g(x)$ is given by the fraction of the area that $g(x)$ occupies. The integration scheme is then to generate a large number of random points inside the box and count the number that are within $g(x)$ to get the area

$$I \approx \frac{n^*}{n} V \tag{2}$$

where n^* is the number of points within $g(x)$, n is the total number of points generated, and V is the generalized volume (depending on the dimension of the problem at hand) of the bounding box.

This method is *very inefficient*, many points are required for Eq. (2) to converge toward the true value of I with any degree of precision.

We will consider an alternative approach. It will be presented in the one-dimensional case, although its real use is in multi-dimensional cases, to which it can easily be extended.

We can get an approximation of I by adding together the areas of n boxes, where the width of each box is $(b-a)/n$ and the height of a box is $g(x_i)$, where x_i is a random value such that $x_i \in [a, b]$, i.e.

$$I \approx \sum_{i=1}^{n} \frac{b-a}{n} g(x_i). \tag{3}$$

It can be shown that estimates based on Eq. (3) converge toward the actual solution as $1/\sqrt{n}$, if the sample points are selected uniformly randomly.

E.2.1 Side note: quasi-random numbers

If uniformly distributed random numbers are used for the Monte Carlo evaluation of integrals then, because of the clumps and voids in any given sample, there will be regions of the integral that are under- as well as over-represented. In the long run it is not a problem since we know that the numbers represent a uniform distribution well. But "long run" means using lots of iterations.

To improve the situation we can try to pick sample points that fill the interval in a more uniform way and don't form clumps and voids along the way. Such number sequences are called "quasi-random" or "maximally avoiding" sequences. (Since the numbers in the sequence are no longer uncorrelated, they are technically not random at all.)

Using quasi-random numbers for choosing the points will cause the integration estimate to converge like $[\log n]^N/n$ (where N is the number of dimensions in the integral). This improved convergence is considerably better, almost as fast as $1/n$.

E.3 Exercises

1. Write a Python program called `integral.py` that can evaluate one-dimensional integrals, using the method in Eq. (3). The program should be controlled with an input file that specifies the number of iteration points, integration limits, and $g(x)$ (expressed in Python).

   There are basically two ways to get the information from the input file; either you can read the file with ordinary Python file I/O functions and use the `eval()` function to evaluate $g(x)$, or you can treat the input file as a Python module and `import` it. Both methods have their (dis)advantages. Be prepared to motivate your choice.

   Using uniformly distributed random numbers from the `random` module to select the sample points is fine. Optionally—if you feel ambitious—you can track down an algorithm for generating quasi-random numbers and implement it in Python.

   The output of the program should contain (i) the final result of integration, (ii) timing statistics (Hint: look up the function `os.times()`.), and (iii) convergence control. As a rough measure of convergence, we can check the relative change of the estimated integral value during the last 100 iterations:

$$R_n = \frac{I_n - I_{n-100}}{I_n} \tag{4}$$

2. Submit a grid job that uses the program to evaluate the integral[8]

$$I = \int_0^{\pi} \frac{x \sin x}{1 + \cos^2 x} dx. \tag{5}$$

   Your job should request the runtime environment `PYTHON` of version `2.3` or higher.

3. Write a Python program that submits the grid job ten times with different n. This means you will have to run `ngsub` from within your program. Consider the `os.system()` and `commands.getstatusoutput()` functions. Also look up the `-o` option to `ngsub` and think about how that can be useful. Plot the convergence of the integration with respect to n.

[8] The analytical value is $\pi^2/4$.

F Exercise 3: Program package development

F.1 Relevant documentation

1. "Swegrid–User's guide"
   http://www.swegrid.se/downloads/swegrid_manual.pdf.

2. "Learning Python" by Lutz and Ascher, O'Reilly (2003).

3. Python 2.3 Library Reference
   http://www.python.org/doc/2.3.4/lib/lib.html

F.2 Runtime environments and Program packages

Scientific computing is performed in projects concerned with research and development. The ultimate aim of the computation is to relate theoretical models to the development of advanced products or functional materials or to the understanding of complex interactions. Some applications fall into relatively broad categories whereas others are more specific in nature. In the former situation it is not uncommon that there are pre-existing program packages that are quite generic and can be utilized by a large number of scientist working on different projects; one example would be given by software available for the simulation of the dynamics of fluids. The software can be effectively adapted to specific applications by the control of user input, and there is typically no need for re-compilations inbetween different application runs. If this is the case, the most efficient use of the grid resource is to pre-compile the software and either include the binary executable in the job request or to pre-install the software on the grid resource. There are several general purpose software packages that are pre-installed on Swegrid, and the user can access these by the request of a certain *runtime environment*.

In the opposite situation, however, the situation may be characterized by specific software design and frequent software modification and on-going development. The use of pre-compiled binaries may be hampered by the inhomogeneity of the grid, and it may be necessary to use the allocated resource to first compile the program that is intended to be run in the application.

This exercise is intended to illustrate the latter situation, and give practice on how to perform runtime compilation in a convenient manner. We will first look at the well-known random walk problem in physics in order to have something to work with in the exercise.

F.3 Random walk problem

A random process consisting of a sequence of discrete steps of fixed length is known as a *random walk*. The random thermal perturbations in a liquid are responsible for a random walk phenomenon known as Brownian motion, and the collisions of molecules in a gas are a random walk responsible for diffusion. Random walks have interesting mathematical properties that vary greatly depending on the dimension in which the walk occurs and whether or not they are confined to a lattice.

In a plane, consider a sum of N two-dimensional vectors with random orientations. Use phasor notation, and let the phase of each vector be random. Assume that N unit steps are taken in an arbitrary direction, i.e., with the angle that is uniformly distributed (and opposed to the case of a lattice), as illustrated above. The position z in the complex

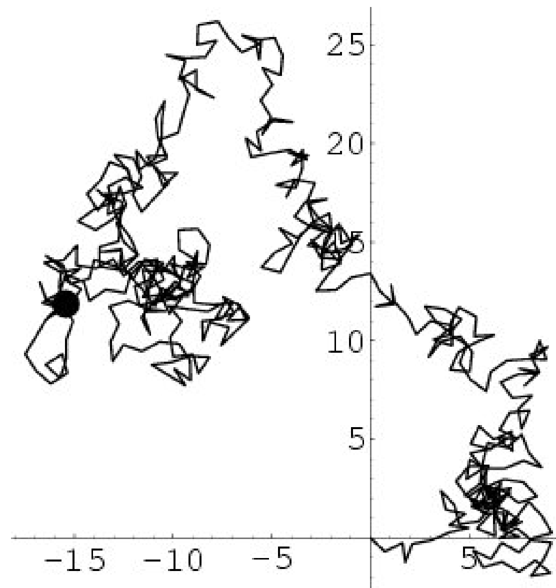

Figure 1: Illustration of random walk with unit step length and arbitrary direction. The angle distribution in the steps is uniform between $[0, 2\pi[$.

plane after N steps is then given by

$$z = \sum_{j=1}^{N} e^{i\theta_j}. \tag{6}$$

If we perform a large number of random walks, all starting at the origin, we would be interested in the mean distance from the origin after N steps. It is straightforward to show that the mean absolute square becomes

$$\left\langle |z|^2 \right\rangle = N. \tag{7}$$

With a step size l (corresponding to the mean free path in diffusion), we can determine the number of steps needed to travel a distance d:

$$N = \left(\frac{d}{l}\right)^2. \tag{8}$$

F.4 The `make` utility program

When the source code of a program is split into multiple files, it is convenient to use the `make` utility program. The `make` command will look for a file named `Makefile` in order to find directives for the compilation of your program—`make` will automatically check the timestamps on the involved files and keep track of whether each file needs to be re-compiled or not, a feature that is most beneficial when working with a large number of source code files. A file `random-walk.c`[9], which contains C source code, can be compiled with use of the following `Makefile` by simply typing `make` at the command prompt[10]. Executing `make clean` will clean up your directory from temporary files such as the object code.

[9]The source code of this program is given in the Appendix

[10]Lines beginning with # are comments. Lines 8, 19 and 22 begin with tabs. It *has* to be tab characters, not space characters. This syntactic silliness is probably the greatest single source of compilation errors in the history of Unix.

```
01:#
02:# Makefile for random walk program
03:#
04:all: random-walk
05:#
06:.SUFFIXES: .o .c
07:.c.o:
08:        $(CC) $(CFLAGS) -c $*.c
09:#
10:CC     = gcc
11:RM     = rm -f
12:CFLAGS = -O3
13:LIBS   = -lm
14:#
15:SRC = random-walk.c
16:OBJ = random-walk.o
17:#
18:random-walk: $(OBJ)
19:        $(CC) $(CFLAGS) -o random-walk $(OBJ) $(LIBS)
20:#
21:clean:
22:        $(RM) -f *.o *~
23:#
24:# End of Makefile
25:#
```

This Makefile is actually overly verbose. Most of these rules and specifications are already present as built-in defaults in make, but are included here for purposes of illustration.

Both Makefile and random-walk.c are available in your `exercises/ex3` directory.

F.5 Exercises

1. Create a gzipped archive file that contains the `Makefile` and the source code file `random-walk.c`. The tarball will function as our "program package". Write a script (in, for example, Python or sh) that is to be submitted to the grid and that will unpack and compile the "program package", and then run the application.

2. Modify the source code by changing the number of iterations. Record in a plot the convergence of the distance with respect to the number of iterations. How many iterations are required to converge the printed distances to within 0.1% of the theoretical values?

3. In many cases the Intel C compiler, icc, produces faster code than the GNU compiler, gcc. However, icc is not available on all systems. Modify your script so that it tries to find icc on the target system. If icc is available, override the value of CC in the Makefile to use icc for compiling the program instead[11].

[11] Check the make manual (do "`info make`") for details.

G Exercise 4: Job scheduler

G.1 Relevant documentation

1. "Swegrid–User's guide"
   `http://www.swegrid.se/downloads/swegrid_manual.pdf`.

2. "Learning Python" by Lutz and Ascher, O'Reilly (2003).

3. Python 2.3 Library Reference
   `http://www.python.org/doc/2.3.4/lib/lib.html`

G.2 Scheduling of large number of tasks with distributed control

A computational scientist frequently needs to perform a large number of calculations using an identical application software with varying input files. In principle, this is an ideal situation for grid usage, since the future vision of the grid is that of a supplier of a large number of compute units that today are used inefficiently. The issue of grid inhomogeneity and binary executables was discussed in the previous exercise, but there are other questions that need to be addressed. For example, one other complication that is caused by the inhomogeneity is the difficulty of pre-estimating the number of required CPU seconds.

In this exercise, however, we will focus on how to create a convenient interface for the grid user to run a large number of similar jobs. The straightforward way is to construct all the necessary input files and submit the jobs with a script. The disadvantage with this solution is that, by the time the jobs are submitted, there is no possibility for the further modifications of the job specifications. The user is also responsible for checking the completion of individual jobs and to re-submit unsuccessful calculations.

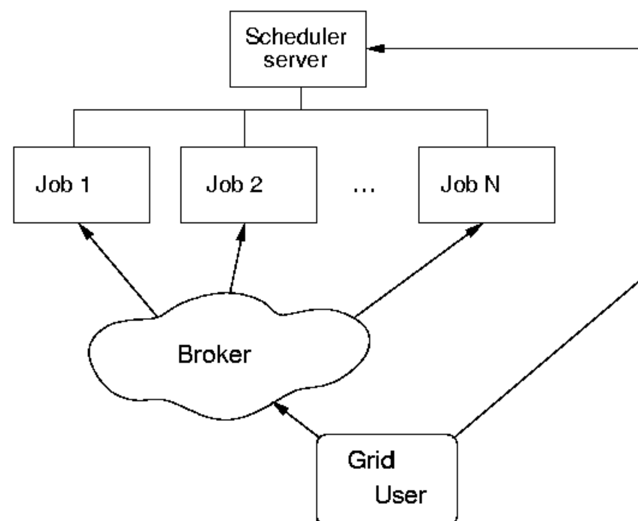

Figure 2: Scheduling of jobs.

Our strategy for scheduling of jobs is illustrated in Figure 2. The *grid user* does not stand in direct contact with the grid compute units—this contact is performed by the *broker*. However, both the grid compute units and the grid user stand in contact with a scheduling server, and the grid user can choose to communicate with the compute units via this scheduling server also after the jobs are submitted to the broker and even while

the job is running on the compute unit. The scheduling server will maintain a *task list* from which the compute unit requests a task specification and reports the status of the task taken.

The problem remains what form the communication between the jobs and the scheduler should take. One could imagine the scheduler to be simply a gridftp server, containing a file with task specifications, that the jobs download and pick the first unclaimed task from. However, that approach has problems of both a practical nature and a more theoretical, fundamental nature. Among the former problems is that (in ARC) jobs have no access to the user's proxy certificate once they have started executing on a compute node. Thus they cannot access a gridftp server as the user[12]. Additionally, there is no guarantee that the compute node even has any ARC software installed, so we do not know whether the job will have access to a gridftp client program. A graver problem is the race condition inherent in this tentative solution—there is no mechanism to stop several jobs from downloading the task specification simultaneously so that they all choose the same task.

It is obvious that this initial stab at a solution is insufficient; we will need something a bit more elaborate. Fortunately, Python comes with built-in support for an remote procedure call (RPC) standard called XMLRPC, which makes it very easy to write simple client-server systems. For example, Figures 3 and 4 show how write an XMLRPC server that provides a single function `add` that takes two arguments and return their sum, and a client that calls the function.

```
#! /usr/bin/env python

from SimpleXMLRPCServer import SimpleXMLRPCServer

def add(x, y):
    return x+y

server = SimpleXMLRPCServer(("", 8000))
server.register_function(add)
server.serve_forever()
```

Figure 3: `server.py`

```
#! /usr/bin/env python

import xmlrpclib

server = xmlrpclib.ServerProxy("http://localhost:8000")
print "The sum of 2 and 3 is", server.add(2, 3)
```

Figure 4: `client.py`

Using the XMLRPC tools it is relatively easy to write a scheduling server and corresponding client. However, there are some traps waiting for the unwary student.

[12]All such transactions are performed during the stage-in and stage-out phases, before and after the actual execution of the job.

Security

Remember that your server will be exposed to the entire Internet, and that there are bad, bad things out there. The client that connects to your server might not be your own well-behaved program.

We would like to be able to use the grid security infrastructure, GSI, and thereby get major parts of the security problem solved automatically, but that is currently not possible since, as mentioned previously, ARC grid jobs do not have access to the proxy certificate. Furthermore, the XMLRPC client in the Python standard library does not support GSI (but there are alternative implementations that do).

We will have to make do anyway. Please keep security in mind at all times. At the very least, incorporate a password argument in your function calls; that stops some of the most trivial attacks.

Port numbers

The XMLRPC server listens for connections on a specific port number, which is specified in the call to the SimpleXMLRPCServer constructor function. If you tell your server to listen on port 8000, your client should connect to the URL `http://login-3.monolith.nsc.liu.se:8000`. The port numbers that are available for user programs are 1024–65535.

To avoid clashes, each student have been assigned a port number range. Please pick port numbers within that range.

Under certain circumstances, the port[13] will remain reserved by the system for several minutes after the server process has exited (it is said to be in a TIME_WAIT state). If you try to reuse the port within that time period you will get an "address in use" error. This is to ensure that all connections to the old process are properly terminated before a new process can listen on the same port, which is normally an excellent idea. However, in a development situation, where you need to restart the server frequently, this can be irritating.

To avoid the TIME_WAIT state, you can tell the system that the port can be reused immediately. The simplest way is to subclass SimpleXMLRPCServer and set `allow_reuse_address` to a true value in the subclass (see Figure 5). (Technically, this specifies that the network socket should be created with the SO_REUSEADDR option set.)

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

class MyXMLRPCServer(SimpleXMLRPCServer):
    allow_reuse_address = 1

server = MyXMLRPCServer(("", 8000))
...and so on...
```

Figure 5: `Reusing port numbers`

[13]Actually, it is the combination of network address and port number which is reserved.

Compute node connectivity

In general, the compute node that a grid job runs on might not be directly connected to the Internet. However, there is an xRSL attribute called "`nodeAccess`" that you can use to request nodes with outbound and/or inbound Internet connection. See the xRSL manual for details.

G.3 Exercises

1. Write a Python script that is to be submitted to the grid N times, where N is larger than the number of jobs that is to be completed (if some jobs needs to be re-calculated). We thus submit an identical script multiple times.

   When the script is executed on a compute unit, it should contact the scheduling server and find the first task that has "to-do" status. It should mark the task as "taken" and construct the appropriate input file for the program `integral.py` from the first exercise. When the job has successfully completed it should mark the task as "done" and give the value of the integral.

   If a task is not "done" within a reasonable timespan from when it was "taken", it is considered to have failed, and should once again be marked "to-do". To test this feature, you might want to introduce something like

   ```
   if random.random() > 0.5:  sys.exit(1)
   ```

   into the client to make a certain proportion of the jobs fail.

   The list of tasks should contain the integrals:

   $$I_n = \int_0^{10} \frac{1}{1+x^n} dx, \quad n = \{1, \dots, 10\}. \tag{9}$$

H Exercise 5: Watchdog

H.1 Relevant documentation

1. "Swegrid–User's guide"
   `http://www.swegrid.se/downloads/swegrid_manual.pdf`.

2. "Learning Python" by Lutz and Ascher, O'Reilly (2003).

3. Python 2.3 Library Reference
   `http://www.python.org/doc/2.3.4/lib/lib.html`

H.2 Scheduling of large number of tasks with local control

In this exercise we will return to the issue of scheduling a large number of jobs. We discussed this issue in the previous section, but let us now choose a different perspective. This time we will focus on the tedious work of supervising grid calculations and retrieving the final results. If you run hundreds of calculations in a project you are likely not to be pleased by the situation of having to collect result files on an individual basis, or even to check for their completion. When run on a local resource, this management is easy since the output of the application is directly visible in your local directory, but, when run on a grid resource, the whole thing is a bit more cumbersome.

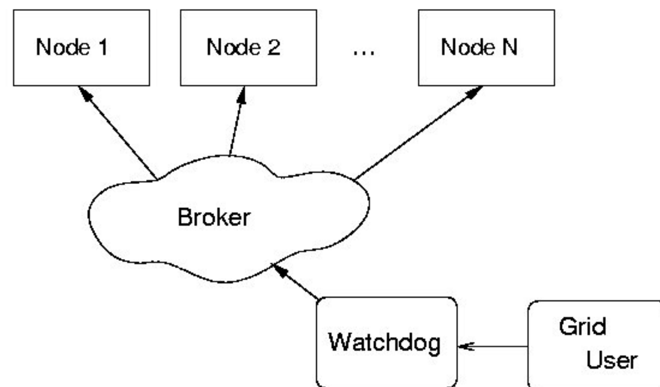

Figure 6: Scheduling of jobs with a watchdog program.

Let us improve on this situation by creating a *watchdog* program that will supervise our calculations. In this case we envisage that we have prepared input files [*.inp] in a local directory, and we desire to, with a minimum a manual labor, successfully run the application for these input files and leave the corresponding output files [*.out] in the same directory.

H.3 Exercises

1. Write a Python program `watchdog.py` that, as an argument, takes the name of an xRSL file. The program should submit the job to the grid, and keep watch of the status of the job. When the job has successfully completed the output file should be downloaded from the grid resource and placed in the same directory as the input file. If the calculations failed, the program should save the standard error message and place this file in the same directory. Other files are not to be kept.

The program should not call `ngstat` more often than once a minute, to avoid unnecessary load on the computing resource.

2. Modify your program to take an arbitrary number of xRSL files and track the jobs in the same way as above.

   Again, you should not run `ngstat` towards any given resource more than once a minute. Of course, if the jobs end up on n resources, you can run n `ngstat` calls per minute—one for each resource.

I Appendix: A basic random walk program

```
/* random walk simulation */
#include <stdio.h>
#include <stdlib.h>

#define NumberIter  10      /* number of walks */
#define NumberSteps 10*100  /* number of steps per walk */
#define pi 3.14159265358979

FILE *output;

main()
{
  int i,j;
  double x, y, random_angle;
  double xsum[NumberSteps+1],ysum[NumberSteps+1], rsum[NumberSteps+1];

  output= fopen ("walk.out", "w");
  srand( (unsigned)time( NULL ) );
  for (i=0; i<=NumberSteps; i++)
    {
      rsum[i]=0; xsum[i]=0; ysum[i]=0;
    }

  for (j=1; j<=NumberIter; j++)          /* average over walks */
    {
      x=0; y=0;                          /* start walk */
      for (i=1;i<=NumberSteps; i++)
        {
          random_angle = 2*pi*(double)rand()/(double)RAND_MAX;
          x += sin(random_angle);
          y += cos(random_angle);

          xsum[i] += x;
          ysum[i] += y;
          rsum[i] += x*x+y*y;
        }
    }
  /* write results into file */
  for (i=0; i<=NumberSteps/100; i++)
    {
      fprintf(output,"%i\t%f\t%f\t%f\n", i*100,
              rsum[i*100]/(double)NumberIter,
              xsum[i*100]/(double)NumberIter,
              ysum[i*100]/(double)NumberIter );
    }

  fclose (output);
}
```