NPL
National Physical Laboratory

Report

NPL REPORT
DEM-ES 006

Software Support for Metrology - Good Practice Guide No. 17

Distributed computing for metrology applications

T J Esward, N J McCormick, K M Lawrence and M J Stevens

NOT RESTRICTED

March 2007

dti
FUNDED BY THE DTI

National Physical Laboratory | Hampton Road | Teddington | Middlesex | United Kingdom | TW11 0LW

Switchboard 020 8977 3222 | **NPL Helpline** 020 8943 6880 | **Fax** 020 8943 6458 | **www.npl.co.uk**

Software Support for Metrology
Good Practice Guide No. 17

Distributed computing for metrology applications

T J Esward, N J McCormick, K M Lawrence and M J Stevens
Mathematics and Scientific Computing Group

March 2007

ABSTRACT

This guide aims to facilitate the effective use of distributed computing methods and techniques by other National Measurement System (NMS) programmes and by metrologists in general. It focuses on the needs of those developing applications for distributed computing systems, and on PC desktop grids in particular, and seeks to ensure that application developers are given enough knowledge of system issues to be able to appreciate what is needed for the optimum performance of their own applications. Within metrology, the use of more comprehensive and realistic mathematical models that require extensive computing resources for their solution is increasing. The motivation for the guide is that in several areas of metrology the computational requirements of such models are so demanding that there is a strong requirement for distributed processing using parallel computing on PC networks. Those who need to use such technology can benefit from guidance on how best to structure the models and software to allow the effective use of distributed computing. This work aims to ensure that metrologists can, when appropriate, take maximum advantage of the significant improvements in computational speed that are offered by cost-effective distributed computing. In addition, it contains case studies and examples that demonstrate what can be achieved with distributed computing technologies.

ISSN 1744–0475

National Physical Laboratory,
Hampton Road, Teddington, Middlesex, United Kingdom TW11 0LW

We gratefully acknowledge the financial support of the UK Department of
Trade and Industry (National Measurement System Directorate).

Approved on behalf of the Managing Director, NPL
by Jonathan Williams, Knowledge Leader for the Electrical and Software team

Contents

1 Introduction **1**
- 1.1 The aim of this guide . . . 1
- 1.2 What is grid computing? . . . 2
- 1.3 What is distributed computing? . . . 2
- 1.4 The benefits of distributed computing . . . 3
- 1.5 The challenges that a distributed system represents . . . 4
- 1.6 A structured approach to developing and testing software . . . 4
- 1.7 The NPL distributed computing system . . . 5
- 1.8 The structure of this guide . . . 5

2 Background to parallel and distributed computing **6**
- 2.1 Introduction . . . 6
- 2.2 The limitations of serial programming . . . 6
- 2.3 Potential gains from parallel processing . . . 7
- 2.4 Parallel and distributed systems . . . 7
  - 2.4.1 Key features of parallel systems . . . 8
- 2.5 Models for parallel and distributed processing . . . 10
- 2.6 Modelling the performance of parallel systems . . . 11
  - 2.6.1 Communications aspects of parallel and distributed computing . . . 13
- 2.7 Defining scalability . . . 14

3 Overview of parallel program design principles **15**
- 3.1 Introduction . . . 15
- 3.2 Automatic parallelisation . . . 15
- 3.3 Identifying opportunities to parallelise a problem . . . 16
  - 3.3.1 Understanding the problem . . . 16
  - 3.3.2 What aspects can be parallelised? . . . 16
- 3.4 Partitioning the computation task . . . 17
  - 3.4.1 Domain decomposition . . . 18
  - 3.4.2 Functional decomposition . . . 20
  - 3.4.3 Reviewing the partitioning process . . . 21
- 3.5 Load balancing . . . 21
- 3.6 Communication . . . 22
- 3.7 Dependencies . . . 23

4 Establishing a distributed computing system **24**
- 4.1 Introduction . . . 24

4.2 Obtaining more computing power . . . 24
4.3 Is distributed computing the answer? . . . 25
4.4 Potential sources of management software . . . 27
4.5 How the server and management software works . . . 27
4.6 Likely performance gains . . . 28

5 Is an application suitable for distributed computing? 30
5.1 Introduction . . . 30
5.2 Suitability of applications . . . 30
5.3 The main application types . . . 31
5.3.1 Data parallelisation . . . 31
5.3.2 Batch applications . . . 32
5.3.3 MPI applications . . . 32
5.4 Communication versus computation . . . 32
5.5 Pre-processing and post-processing . . . 33
5.5.1 Response surface modelling . . . 33

6 System administration and quality issues in DC systems 34
6.1 Introduction . . . 34
6.2 System administration . . . 34
6.2.1 Scheduling jobs . . . 34
6.2.2 General advice on scheduling . . . 36
6.3 File formats . . . 37
6.4 Software quality issues . . . 38
6.4.1 Best practice guide for software development . . . 38
6.4.2 Grid-specific quality concerns . . . 39
6.4.3 Avoiding dependence on installed system resources . . . 42
6.5 Application certification procedure . . . 44

7 Random numbers and distributed computing 45
7.1 Introduction . . . 45
7.2 Random number generators: the background . . . 45
7.2.1 General principles . . . 46
7.2.2 The period of the generator . . . 46
7.3 The Wichmann-Hill generator . . . 46
7.3.1 The construction of the generator . . . 47
7.3.2 Distributed computing and random number generation . . . 47
7.4 Implementing the Wichmann-Hill generator on the NPL grid . . . 47
7.4.1 A *FORTRAN* version of the generator . . . 47
7.4.2 Testing the program locally . . . 48
7.4.3 Building the program module . . . 48
7.4.4 Building the data module . . . 48
7.4.5 Testing with the test agent . . . 49
7.4.6 Registering on the NPL grid . . . 49
7.4.7 Running on the test trid . . . 49

8 Distributing specific application examples 51
8.1 Introduction . . . 51
8.2 Java and distributed computing systems . . . 51
8.2.1 Installing Java on the donor machines . . . 51

8.2.2 Creating a stand-alone module . . . 52
8.2.3 Making the module general purpose . . . 53
8.2.4 Choosing the right VM . . . 53
8.2.5 Memory concerns . . . 53
8.2.6 Logoff issues and the Sun VM . . . 54
8.3 Comsol and distributed computing . . . 54
8.3.1 Comsol's Java API . . . 54
8.3.2 Licensing Comsol . . . 55
8.3.3 Links . . . 55
8.4 Finite element modelling with PAFEC . . . 56
8.4.1 Licence issues . . . 56
8.4.2 Interaction with donor machines . . . 56
8.4.3 Implementation and efficiency issues . . . 56
8.4.4 Input file generation . . . 57
8.4.5 Performance . . . 58
8.4.6 Advantages . . . 59
8.5 MPI applications on the grid . . . 59
8.5.1 Limitations and problems . . . 63
8.6 Distributing Numerical Algorithms Group routines . . . 63
8.7 Distributing MATLAB . . . 64
8.7.1 Creating a standalone executable . . . 64
8.7.2 Coding requirements . . . 64
8.7.3 Distribution . . . 65

9 Case studies and examples 66
9.1 Modelling electron transport in quantum dot systems using distributed processing techniques . . . 66
9.1.1 Introduction . . . 66
9.1.2 The problem . . . 66
9.1.3 Why use the NPL grid? . . . 69
9.2 Monte Carlo calculations for ionising radiation applications . . . 72
9.2.1 Scientific background . . . 72
9.2.2 Technical details of the implementation on the United Devices grid . . . 73
9.3 Adiabatic energy transfer in caesium atoms . . . 74
9.4 Using commercial FE software on the NPL Grid . . . 75
9.4.1 Input sensitivity analysis of finite element modelling – summary . . . 75
9.4.2 Finite element analysis using the NPL grid . . . 76
9.4.3 Input file generation . . . 76
9.4.4 NAFEMS benchmark LE5: Z section cantilever . . . 76
9.4.5 Piezoelectric analysis . . . 77
9.4.6 Conclusions . . . 78
9.5 Boundary elements for acoustic applications . . . 78
9.5.1 Introduction . . . 78
9.5.2 Problem formulation . . . 79
9.5.3 Results . . . 80
9.5.4 Extensions and lessons . . . 83

Bibliography 84

A Glossary of terminology relevant to the NPL grid **86**
A.1 Introduction . . . 86
A.2 Key definitions . . . 86
A.3 Administration software definitions . . . 88

B Grid program development, design and creation **90**
B.1 Can the NPL grid help my application? . . . 90
B.2 The Program Itself . . . 91
B.2.1 Some advice for development . . . 92
B.2.2 Making the Workunits . . . 92
B.2.3 Getting and interpreting the Results . . . 93
B.3 Using Buildmodule, Buildpkg, and running the Test Agent . . . 93
B.3.1 Using Buildmodule . . . 93
B.3.2 Other options for Buildmodule . . . 93
B.3.3 Using BuildPkg . . . 94
B.3.4 Using the Test Agent . . . 94
B.4 Requesting grid access . . . 95

C Developing software for the NPL distributed computing grid **96**
C.1 Prepare the program . . . 96
C.2 Installing the United Devices Software Developers' Kit (SDK) . . . 96
C.3 Test the program locally . . . 97
C.3.1 Build the Program Module . . . 97
C.3.2 Build the Data Modules . . . 97
C.3.3 Test the program using the Test Agent . . . 97
C.4 Set up an application on the test grid . . . 98
C.5 Install Windows Perl on the PC . . . 98
C.6 Run the application on the test grid . . . 98
C.7 Run the application on the live grid . . . 98

D Form for requesting grid access **99**
D.1 Request For NPL grid access . . . 99
D.1.1 Your details . . . 99
D.1.2 Application details . . . 99
D.1.3 Resource requirements . . . 100
D.1.4 End-to-End issues . . . 101

E Matlab Distributed Computing Toolbox 2.0 **102**
E.1 Introduction . . . 102
E.2 Key features . . . 102
E.3 Working with the Distributed Computing Toolbox and the MATLAB Distributed Computing Engine . . . 103
E.3.1 Creating jobs with the Distributed Computing Toolbox . . . 103
E.3.2 Dynamic licensing on the Distributed Computing Engine (DCE) . . . 104
E.3.3 Scheduling jobs . . . 104
E.4 Executing applications . . . 104
E.4.1 Distributed execution . . . 104
E.4.2 Parallel execution . . . 104
E.4.3 Working in managed interactive or batch mode . . . 105

E.5 Web links 105

F Use of DOE/RSM techniques **106**
F.1 Introduction 106
F.2 Choice of design 107
F.3 Running the experimental design and building response surface models 107

Chapter 1

Introduction

1.1 The aim of this guide

This guide was prepared as part of the Department of Trade and Industry's Software Support for Metrology programme 2004-2007. It aims to facilitate the effective use of distributed computing methods and techniques, and particularly PC desktop grids, by other National Measurement System (NMS) programmes and by metrologists in general. In addition, it focuses on the needs of those developing applications for distributed computing systems rather than the needs of systems administrators and superusers. The guide seeks to ensure that application developers are given enough knowledge of system issues to be able to appreciate what is needed for the optimum performance of their own applications, but it does not specifically address issues of concern mainly to administrators and IT support staff. It is likely that the best source of support on such topics, especially in the case of commercially developed grid systems, will be the vendors and their own support staff, and the extensive guides on installation and system administration that are likely to accompany commercial systems.

Within metrology, the use of more comprehensive and realistic mathematical models that require extensive computing resources for their solution is increasing. The motivation for the guide is that in several areas of metrology the computational requirements of such models are so demanding that there is a strong requirement for distributed processing using parallel computing on PC networks. Those who need to use such technology can benefit from guidance on how best to structure the models and software to allow the effective use of distributed computing. In the course of preparing this guide, a series of case studies was carried out to test the ideas put forward. In addition, a workshop was held to allow users of distributed computing systems to share experiences and comment on initial drafts of the guide. The results of the case studies and workshop have been incorporated into the final version of the guide.

This work aims to ensure that metrologists can, when appropriate, take maximum advantage of the significant improvements in computational speed that are offered by cost-effective distributed computing.

1.2 What is grid computing?

The use of distributed systems for the solution of large computing problems is a rapidly developing branch of technology, and the meanings of certain technical terms associated with the field appear often not to be fixed. It is therefore important to begin with a definition of what *grid computing* and *distributed computing* are considered to be for the purposes of this guide.

Grid computing refers to the establishment of an environment in which users can access computers, databases and experimental facilities without needing to be aware of where those resources are located. This is the definition that is used by the *RealityGrid* (`www.realitygrid.org`) which is funded as part of the UK's *e-science* initiative and aims to grid-enable the realistic modelling and simulation of complex condensed matter structures at the meso- and nanoscale levels. When the term *grid computing* is used in the sense in which it is employed here, it refers to computers and resources that are located in a number of organisations and under the control of different system administrators. They are accessed via the Internet and it should not be apparent to users that they are crossing boundaries between different organisations and systems when accessing the grid.

1.3 What is distributed computing?

Distributed computing in its most general sense is computing that allows a number of computers that are remote from each other to take part in solving a computational problem or to process or access information. In this sense the term refers to distributed client-server computing environments in which processes are located at the most suitable location for business or administrative purposes and are accessed by users at PCs that provide an interface to the resource in question. Thus, financial or accounting information, or database records, for example, may be stored and processed at one location, but the information may be accessed and amended by users working at remote PCs. This approach to distributed computing is well described in Khanna [10].

Another more recent use of the term *distributed computing* refers to collaborative initiatives in which large numbers of users of PCs allow some of their computer's spare processing capacity to be used to tackle a large computing problem, with the collaboration often being co-ordinated by means of the Internet. A well-known example of this is *SETI@home* in which computer owners offer their machines to work on the Search for Extraterrestrial Intelligence (SETI) project to download and investigate radio telescope data. More information about this project can be found at `http://setiathome.ssl.berkeley.edu/`. Similar initiatives have focused on drug discovery to screen molecules that may have the potential to be developed into pharmaceuticals to treat specific diseases.

It is this second definition of distributed computing on which this guide concentrates. However, the guide restricts itself to cases in which the computers that are donated to tackle a large computational problem are located within one organisation and are under the control of one systems administrator.

A typical distributed computing architecture consists of some kind of small, unobtrusive agent program that is installed on the computers that form the network and one or more dedicated servers that manage the distributed system. In addition, the system will also have its users or clients, that is, scientists and programmers who wish to submit computing jobs to the system.

The agent program will communicate with the managing server to advise it that it is available to process work tasks, receive application packages from the server, run these applications locally, send the results back to the server, and then request further work tasks from the server. This process should be effectively invisible to the user of the computer on which the agent software is running.

1.4 The benefits of distributed computing

In large organisations, such as the UK's National Physical Laboratory, there are large numbers of PCs that are used as desktop machines by scientists and administrators or are employed to control experimental apparatus. It is in the nature of the normal use of PCs that the full processing capability of such machines is very rarely utilised. Distributed computing is a way of utilising the "unused" power of desktop computers to tackle large processing problems. In its simplest form it can be regarded as a method of running a program on many machines in parallel in a co-ordinated manner. In this way the computing task can be completed many times faster than is possible with serial processing, depending on the number of computers available to take part in the calculation.

Distributed computing can therefore be regarded as a form of parallel processing. However, the main difference between parallel processing in its most general form and distributed processing is the degree of communication possible or allowable between parallel processes. In true parallel programming architectures there can be memory shared by all the parallel processes that acts as a common data area. This can allow communication between the parallel processes themselves, and it ensures that the results of calculations by one process can be available for other processes to use as required. In a distributed computing environment, the possibility of communication between PCs in the network does not exist, unless a Message Passing Interface is used. The allocation and timing of work units in a distributed system is left entirely to the system server, and this in turn depends on the availability of networked computers. Such availability cannot be guaranteed and there is thus no certainty that a given machine will be processing a required parcel of data at a given time to allow inter-process communication.

For a general introduction to the principles that underlie parallel computing, readers are referred to [12].

There are two main classes of application that can best take advantage of distributed computing systems. The first of these is data parallel applications, that is, programs that operate on different control files or input files, but without the need to change any aspect of the program itself, which means that identical executables can be sent to each machine in the network. In such a case, a program that has been written to operate in a serial processing environment can be transferred directly to a distributed computing

system. The second form of application is one where a large set of input data is divided up between machines so that each works on a small part of the data set. Here, it is likely that the executable for the distributed version will differ from that of the serial version.

1.5 The challenges that a distributed system represents

To use a distributed computing system efficiently, users must recognise the key features of the system. Firstly, the computers that form the network are likely to be a mixture of machines, with different processor speeds, different memory capacities, and perhaps from a range of manufacturers. In addition, machines may become unavailable at any time, as users may switch them off or re-boot them, or the machines may not remain connected to the network. Users must also recognise that most desktop PCs are not completely secure, and the onus is on them to ensure that their grid applications are able to withstand malicious attempts to interfere with their operation, and that they do not compromise other applications running on each machine.

Users must also ensure that grid applications are unobtrusive, that is, users of desktop PCs should not notice any deterioration in the performance of their machines when distributed tasks are being processed. Thus, it is advisable to avoid distributed applications that need to transfer very large data files between the server and the agent PCs either at the end or beginning of computational tasks. If this is not done, then there is the likelihood of adverse reactions from the users of donor PCs, which may lead to hostility to distributed computing developments within an organisation and requests to have PCs removed from the list of donor machines. Where there is a risk of some distributed applications prejudicing the performance it may be possible to provide a software tool for donors that allows them to suspend temporarily any distributed application that is running on their machine, or to opt out of the network temporarily. Following a request for such a facility, NPL's grid supplier, United Devices, provided one for NPL's use. It is, of course, important that such a software tool should not allow donors to disconnect permanently from the distributed network.

Finally, one should aim to make the best use of the available network bandwidth. This can be achieved by distributing applications whose ratio of computation to communication needs is high. Contributing machines should be spending almost all their time calculating rather than transferring data to and from the server.

1.6 A structured approach to developing and testing software

It is clear from the remarks above that to achieve the best performance from a grid computing system from the points of view of the system administrator, the programmer developing applications for the system, and donors of agent PCs, a structured approach to the development and testing of software is

needed. Guidance provided here takes into account the interests and concerns of all those associated with the distributed computing system.

1.7 The NPL distributed computing system

The case studies and examples described in this Guide are all taken from applications that run on the National Physical Laboratory's Grid MP system (Grid MP is the proprietary name for the United Devices software used to administer our distributed computing system). However, we have tried to make the examples as general as possible so that they demonstrate principles, rather than the details of a specific grid implementation. Whenever it has been necessary to describe features that are unique to the United Devices system that we employ, this has been pointed out. It has often been necessary to use terminology that may be specific to the United Devices system, and appendix A provides a short guide to key terms.

NPL currently uses the United Devices (UD) Grid MP 4.0 system and the accounts of our own experience with DC systems in this Guide relate specifically to this system. Grid technology is a fast-developing area and we recommend that readers who are contemplating the introduction of a PC desktop grid into their own organisation make their own additional enquiries of software and system vendors and do not rely completely on this Guide. For example, later releases of the UD Grid MP system allow users to combine desktop PCs, servers, workstations, clusters and other high-performance computing resources into a single grid.

1.8 The structure of this guide

This report begins with an introduction to some of the general principles underlying parallel and distributed computing. Chapter 2 sets out a general introduction to parallel and distributed computing and aims to give the reader enough background to be able to understand and use this good practice guide. An overview of parallel program design principles is given in chapter 3 and this is followed by chapters 4 and 5 on establishing a distributed computing system and deciding whether an application is suitable for distributed or parallel processing. Subsequent chapters give advice on developing applications for a distributed system, software quality processes specific to distributed computing, and programming for parallel and distributed applications. Random number generators and how these should be implemented in a distributed environment are described in chapter 7, and chapter 8 discusses distributing specific application examples. Finally, there is a series of short case studies that exemplify some of the topics discussed in the main body of the guide and give examples of applications that have been implemented successfully on the NPL grid. The guide also includes appendices that provide a glossary of distributed computing terms, present some specific details of system administration on the NPL Grid, and describe the MATLAB Distributed Computing Toolbox.

Chapter 2

Background to parallel and distributed computing

2.1 Introduction

This chapter gives a general introduction to parallel and distributed computing and aims to give the reader enough background to be able to understand the remainder of this good practice guide. It is based on textbooks written by Bertsekas and Tsitsiklis [2] and by Quinn [13] and the Introduction to Parallel Computing published on the Internet by the Lawrence Livermore National Laboratory in the USA (`www.llnl.gov/computing/tutorials/parallel_comp/`). The reader may wish to consult these for further details.

2.2 The limitations of serial programming

The traditional method of writing computer programs has been to perform computations serially, that is, programs are to be executed by a single computer with a single central processing unit (CPU). A serial problem is solved by following a series of instructions that are carried out one after the other by the CPU. Only one instruction can be executed at a time.

The limitations of this approach are obvious. The time it takes to solve a problem is governed by how many instructions there are to carry out and how quickly individual instructions can be performed. For large problems, the time required to perform a calculation may be so long that the task becomes intractable.

There are also physical and economic limits to serial processing. The speed of transmission of data depends on physical limits, such as propagation delays both between and within integrated circuits. To increase speeds, devices and systems increasingly get smaller, and each new generation of processor corresponds to a large research and development effort. It may be cheaper in some circumstances, therefore, to use several processors of more moderate speed if the problem is capable of parallelisation.

Parallel processing attempts to overcome these limitations by tackling

at least part of a problem using a range of computing resources simultaneously. At the most general level, such a system might consist of multiple processors within a single computer, a number of computers in a network, or combinations of such systems.

If there is to be a benefit in attempting to tackle a problem using parallel processing methods, then there are some specific conditions that must be met. It must be possible to divide the computing tasks into discrete pieces of work that can processed simultaneously and it must be possible to solve the problem in less time than it would take to solve it serially. This can only be achieved if it is possible to execute program instructions on different processors at the same time.

2.3 Potential gains from parallel processing

The main reasons for choosing a parallel computing solution to a problem are to save time and to solve larger problems than would be feasible serially. However, as the Lawrence Livermore guide emphasises, there are a number of other reasons for choosing parallel methods. These include being able to take advantage of non-local resources (such as computers linked by the Internet or on an Intranet – this is the model this guide addresses), and economic reasons, such as using a larger number of cheap resources rather than upgrading to a much more powerful supercomputer, for example. In addition, it may be possible to overcome memory limitations associated with single machines.

2.4 Parallel and distributed systems

In general, parallel computing may be considered to consist of three main processes:

1. Allocating tasks – breaking down the total workload into smaller tasks that can be assigned to different processors, and the proper sequencing of tasks. Correct sequencing is essential when tasks are interdependent and cannot be executed simultaneously.

2. Communicating interim results between processors, as is required when a particular process needs a result from another process before it can continue its own work.

3. Managing synchronisation. In synchronous processing each process waits at predetermined points for completion of certain computations or the arrival of certain data from other machines. In asynchronous processing there is no requirement for waiting at predetermined points.

The original motivation for developing parallel and distributed computing in science and mathematics can probably be attributed to the need to solve systems of partial differential equations for applications such as large computational fluid dynamics problems or for weather prediction. In such cases there is a large number of numerical computations to be carried out, coupled with a desire to solve more and more complex problems more rapidly. In this type of application one can decompose the problem along a spatial

dimension and calculate results for small regions of space, because the interactions between variables are local in nature. Another example of large scale computations is the analysis, simulation and optimisation of large scale interconnected systems. In practice these may be harder to divide into smaller, separate calculations. To overcome this limitation one uses fewer, more powerful processors, co-ordinated through a more complex control mechanism. A third area is information acquisition, extraction and control within geographically distributed systems. In this case the distributed system has to be able to operate correctly in the presence of limited and sometimes unreliable communication capabilities

The essence of parallel computing is that it consists of several processors that are located within a small distance of each other and whose main purpose is to execute jointly a computational task. In such an architecture, communication between processors must be reliable and predictable.

In a distributed computing system, processors may be far apart. Communication between processors may be problematic. There may be unpredictable communication delays and communication links may be unreliable. The topology of the system may change during operation as a result of failures or repairs, as well as the addition or removal of processors. Each processor may be engaged in its own local computing activities while at the same time co-operating with other processors in a computational task. The co-operative task is unlikely to be the reason for the network's existence.

2.4.1 Key features of parallel systems

Here some of the key features of parallel systems are defined briefly and some definitions of key terms are given.

The first issue to consider is the type and number of processors. In *massively parallel* systems many thousands of processors may be involved. In *coarse-grained parallelism* there will be a small number of processors, perhaps no more than ten, each of which may be relatively powerful. The processors may be loosely coupled so each may be performing a different task at any given time.

Another important feature is the existence or absence of some kind of global control mechanism. There is almost always some degree of central control but its form may vary. At one extreme the global control mechanism is only used to load a program and data to the processors and each processor then works on its own. (This is essentially the model of distributed computing that the remainder of this guide addresses.) At the other extreme the control mechanism is used to instruct each processor what to do at each step.

The terminology used to describe such control mechanisms arises from the work of Flynn and is referred to as Flynn's Taxonomy [8]. Flynn argues that there are four possible architectures that can be used to classify computer systems:

1. SISD (single instruction, single data). This refers to traditional serial computing in which only one instruction is carried out by the CPU at each clock cycle and there is only one data stream that is being used as input during any single clock cycle.

2. SIMD (single instruction, multiple data) is an example of a parallel processing operation in which all processors carry out the same instruction at any given clock cycle but each processor operates on a different data element.

3. MISD (multiple instruction, single data) is relatively unusual – the processors are carrying out different instructions but are operating on the same data. One could imagine that a code breaking algorithm might operate in this way.

4. MIMD (multiple instruction, multiple data) where many processors perform different operations on different data at different times.

Distributed computing, as described in this guide, most closely resembles the MIMD case. Although typically we expect that all processors in the network will be running the same executable, they may all be at different stages within the program and will each be operating on different data.

The next key question is whether systems operate synchronously or asynchronously. These terms refer to the presence or absence of a common global clock to synchronise operation of different processors. This common clock is by definition present in SIMD systems. Thus the behaviour of the system is easier to control and algorithm design is simplified, but it involves considerable overhead and synchronisation may ultimately be impossible to achieve. It is also difficult to synchronise a data communication network. A parallel computer operating asynchronously may be able to emulate a synchronous system. In a distributed computing system of the kind discussed in this guide, the clocks of the individual processors making up the network are not synchronised (or at least do not need to be synchronised).

Finally, the manner in which processors exchange information is significant. The two extreme cases are shared memory architectures (all processors can access the same memory) and message-passing architectures (there is no shared memory, but processors communicate by sending messages to each other), with a variety of architectures lying between these extremes. In the shared memory case, all processors can operate independently but the memory resources are shared and changes to items in memory are visible to all processors. This approach has an advantage for programmers in that it is easier to program such a memory configuration. The main disadvantage is the fact that performance does not scale with the addition of CPUs to the system. Additional CPUs can lead to increased data communications traffic along the connection between the CPUs and the shared memory.

In the distributed memory approach there is no shared memory, but each processor has its own local memory. There is no global address space that is available to all processors. Each processor, with its local memory, operates independently. If a processor needs access to data held by another processor, then it will normally be the responsibility of the person programming the system to determine how this is to be done. Note that in a distributed memory system the memory scales with the number of processors; increasing the number of processors increases the size of available memory.

It is, of course, possible for hybrid systems to combine both shared and distributed memory features.

2.5 Models for parallel and distributed processing

The descriptions of parallel processing architectures set out above lead naturally to a set of models that describe the manner in which parallel programs operate.

- *Shared memory model:* this follows directly from the shared memory architecture described above. Processors can read from, and write to the shared memory space asynchronously. For a programmer, this architecture is simple as it is not necessary to specify how data are communicated between tasks, so that the process of developing programs may be simpler.

- *Threads model:* in this approach a program begins as a single process that proceeds serially until it reaches a parallel construction. At this point a number of tasks (or threads) are created which are run by the operating system concurrently. Once these are completed the main program continues in serial fashion until the next parallel section is encountered. This can be seen as analogous to a program with sub-routines, but with the ability for sub-routines to run concurrently. Each thread may posses local data but also has access to the data and resources of the main program. Threads communicate through global memory. An example of how threads operate is found within the OpenMP FORTRAN language, in which a master thread is executed serially until the directive PARALLEL is encountered, at which point a team of threads is created. These threads are executed in parallel. When the END PARALLEL directive is reached, each of the threads is synchronised and the main program thread continues serially.

- *Message passing model:* in this approach each task uses its own local memory during a computation. Each task exchanges data by sending and receiving messages. Data transfer has to be a co-operative process – for each data transmission operation there must be a related receiving operation. Parallelism is defined by the programmer. Standardisation of Message-Passing Interfaces (MPI) was begun in the early 1990s by the MPI Forum (`www.mpi-forum.org/`) and the original MPI specification is available on the Internet at `www-unix.mcs.anl.gov/mpi/standard.html`. In the United Devices Grid MP system, it is possible to implement a version of message passing called MPICH, which is designed to be portable, and chapter 8 discusses the use of the NPL grid in this way. Information and downloads for MIPCH can be found at `www-unix.mcs.anl.gov/mpi/mpich/`.

- *Data parallel model:* in this model one is concerned with operations that are performed on a data set. Processors work collectively on the same data structure, but each performs the same computation on a different part of the structure. How the data are made available may vary: in the shared memory model all tasks can access the data, which is stored in global memory, while in a distributed system the data will be split up and only the relevant part of it will be stored in local memory.

- *Single Program Multiple Data:* this refers to the case of a single program being executed simultaneously by all processors. At any particular moment, the processors may not be executing the same instructions within the same programme and they may be working on different data. It is this model that the majority of applications on the NPL grid adopt.
- *Multiple Program Multiple Data:* in this case different processors may be executing different programs and working on different data.

Hybrid systems may combine features of the various individual models described above.

2.6 Modelling the performance of parallel systems

There are a number of models of the performance of parallel and distributed systems that describe the computing power of parallel systems and interprocessor information transfer mechanisms. However, for the purposes of this Good Practice Guide it is not essential to consider these in detail and there are many texts in the computing literature that deal with this topic. Nevertheless, we include a brief discussion of this question to help readers appreciate some of the issues that are involved in quantifying the performance of parallel and distributed systems.

Firstly, it is necessary to make some assumptions; typically these are that each processor is capable of executing certain basic instructions, such as arithmetic operations, comparisons and branching instructions (if ... then etc) and that there is a mechanism by which processors can exchange information. Assume that each basic instruction requires one time unit and that information transfers are instantaneous and cost free. Delays may be introduced into models as needed. Once these rules have been defined, it is possible to represent the flow of information through a system graphically. One technique for doing so is the directed acyclic graph. For more information about these, see Bertsekas and Tsitiklis [2]. Such graphs aim to show what operations are to be performed on what operands and show the precedence constraints on the order in which operations are to be performed.

Complexity measures are intended to quantify the amount of computational resources used by a parallel algorithm. Some measures are:

- the number of processors,
- the time until the algorithm terminates (time complexity),
- the number of messages transmitted in the course of the algorithm (communication complexity).

Following the approach of [2], complexity measures are often defined as a function of the size of the problem to be solved. An informal definition is the number of inputs to the computation, for example, in the case of the problem of adding n integers, n is a natural measure of problem size. If the problem size is held constant, it is still possible that the resources used depend on the actual value of the input variables. The usual approach is to count the

amount of resources required in the worst case over all possible choices of data corresponding to a given problem size.

In relation to time complexity it is conceivable that an algorithm has terminated, i.e. desired outputs are available at some processors but that no individual processor is aware of this fact.

To quantify speed-up and efficiency, one can consider a computational problem parameterised by the variable n representing the problem size. Time complexity is normally dependent on n.

For a parallel algorithm that uses p processors (p may depend on n) and that terminates in time $T_p(n)$, let $T^*(n)$ be the optimal serial time to solve the same problem, that is the time for the best serial (single processor) algorithm for this problem. The ratio

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \tag{2.1}$$

is called the *speed-up* of the algorithm and describes the speed advantage of the parallel algorithm compared to the best possible serial algorithm.

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T^*(n)}{pT_p(n)} \tag{2.2}$$

is the efficiency of the algorithm and measures the fraction of time that a typical processor is usefully employed. Ideally $S_p(n) = p$ and $E_p(n) = 1$, so that the availability of p processors speeds up the calculation by factor of p. This can only happen if no processor is ever idle or does any unnecessary work. This situation is practically unattainable. $T_*(n)$ is often defined differently. Other definitions include:

- let $T^*(n)$ be the time required by the best existing serial algorithm.
- let $T^*(n)$ be the time required by a benchmark serial algorithm. For example, for the multiplication of two dense $n \times n$ matrices, a time proportional to n^3 is a reasonable benchmark although algorithms with smaller time requirements exist.
- let $T^*(n)$ be the time required for a single processor to execute the particular parallel algorithm being analysed, i.e. let a single processor emulate the operation of p parallel processors. Note that this approach shows the efficiency as related to how well a particular algorithm has been parallelised but gives no information on the absolute merits of the parallel algorithm unlike the earlier two definitions.

The main difficulty in achieving the best performance arises from the fact that in the real world, and considering the whole computational task, which might include serial processing (including preparation of data for input to the parallel section of the program and combining and analysis of the results after the parallel part of the computation has been completed) bottlenecks might arise that limit performance. This is expressed in *Amdahl's Law* [1], that is, the computational process has two parts, one that is inherently sequential and another that is fully parallelisable, and if the inherently sequential section consumes a fraction f of the total computation, the speed-up is limited by:

$$S_p(n) \leq \frac{1}{f + (1-f)/p} \leq \frac{1}{f}, \forall p. \tag{2.3}$$

There has been some debate in the literature about how to use Amdahl's Law and especially how the serial fraction of a program should be defined. One of the assumptions on which the law is based is that the serial proportion of the program is independent of the number of processors to be used in the parallel part. Another is that a serial algorithm should retain its structure when parallelised, so that the same number of instructions is carried out by both the serial and the parallel version for the same input. The simplest way to avoid these difficulties is to use processing times as the basis for determining the serial fraction of a computational process. However, in practice this approach also has limitations when the serial proportion is obtained from computational experiments by recording the total time taken for a computation and the time taken for the parallel-only part of the computation. All the computational overheads, such as communication, synchronisation, data input and output and memory access will be allocated to the serial percentage. If instead of timing the computation, one counts the number of serial and parallel instructions in a program, then all these overheads are ignored. For more information see the on-line paper by Yuan Shi entitled *Re-evaluating Amdahl's Law and Gustafson's Law* (`http://joda.cis.temple.edu/~shi/docs/amdahl/amdahl.html`).

2.6.1 Communications aspects of parallel and distributed computing

For many algorithms and systems, the time spent on interprocessor communication is a sizeable fraction of total time needed to solve a problem, i.e. the application experiences substantial communication penalties or communication delays. The communication penalty CP can be defined as:

$$CP = \frac{T_{TOTAL}}{T_{COMP}}, \tag{2.4}$$

where T_{TOTAL} is the time required by the algorithm to solve the given problem and T_{COMP} is the corresponding time that can be attributed just to computation, i.e. what would be required if all communications were instantaneous.

Communication delays have four components (assuming that packets are not divided and recombined en route to the eventual processor):

1. communication processing time: time required to prepare information for transmission, i.e. assembling information in packets, appending addressing and control information, selecting a link on which to transfer each packet, moving the packet to appropriate buffers.

2. queuing time: once information is assembled into packets for transmission on a communication link it must wait in a queue prior to the start of transmission, e.g. the link may be temporarily unavailable because other information packets or system control packets are using it or scheduled to use it, or because contention resolution is under way. In the case where packet transmission errors are non-negligible the queuing time includes time for packet re-transmissions to correct the errors.

3. transmission time: time to transmit all the bits of the packet.

4. propagation time: time between the end of transmission of the last bit of the packet and the reception of the last bit at the the receiving processor.

Depending on the given system and algorithm one or more of these times may be negligible.

2.7 Defining scalability

In relation to the above discussions the concept of scalability of a computation is important. The definition of scalability given by Quinn [13] and which is used in this guide, is:

- an algorithm is scalable if the level of parallelism increases at least linearly with the problem size;
- an architecture is scalable if it continues to yield the same performance per processor, albeit used on a larger problem size, as the number of processors increases;
- data-parallel (single operation on a data set) algorithms are more scalable than control-parallel (different operations on different data sets simultaneously) algorithms because the level of control parallelism is usually a constant, independent of the problem size, whereas data parallelism is an increasing function of problem size.

In considering whether there might be any benefit in converting an existing serial application to a distributed or parallel application, questions of scalability should be considered. For data parallel applications such as those that can be run most easily on the NPL grid, there are always likely to be scalability benefits provided there is a reasonable ratio of computation to communication time.

Chapter 3

Overview of parallel program design principles

3.1 Introduction

This chapter sets out a brief introduction to some of the main principles of parallel program design, and provides some general background to the more detailed discussions relating specifically to distributed and grid systems that take place in later chapters. It is not intended to be a comprehensive guide to developing parallel programs for all types of parallel computer – there are a large number of texts and Internet resources that can provide the necessary assistance. The aim here is to summarise some of the more general principles and to indicate those that have the greatest relevance to grid and distributed computing systems.

3.2 Automatic parallelisation

Several tools are available to assist in the conversion of serial programs to parallel form. These parallelising compilers and parallelising pre-processors work by analysing the serial source code to identify parallelisation opportunities. The most obvious candidate constructions for parallelisation in this manner are FOR and DO loops. Some software tools are fully automatic. Others allow the programmer to include directives to the compiler as to how to parallelise the code. Available tools can easily be located by performing Internet searches using the terms “parallelizing compiler” or “parallelizing pre-processor”.

There is a risk in employing automatic parallelisation techniques that one may not get the most efficient parallelisation, that there may be no noticeable improvement in performance over the serial version, and that only the most obviously parallelisable parts of the code will be addressed by the compiler or pre-processor. It is likely therefore that most application developers will wish to consider manual parallelisation methods. In the case of the data parallel applications that users of distributed systems of the kind discussed in this guide will be using, and where application development is

likely to begin from some existing serial version of the software, manual methods are likely to be preferable.

3.3 Identifying opportunities to parallelise a problem

3.3.1 Understanding the problem

As with any software development task, the initial stage is to understand the problem to be tackled. Typically one might begin with written specifications of the problem (often called the *user specification* in software quality procedures) and how the software is to solve it (often known as a *functional specification*). There are a number of general rules that can be set down for developing software, and any organisation that is involved in software development will have its own quality procedures that should be followed for both serial and parallel applications. Chapter 6 summarises the key points to be borne in mind when preparing and carrying out software quality plans and meeting software quality requirements.

Often one will begin with an existing serial program. Here it will be essential to understand this program before one can begin to tackle its parallelisation. Clearly it is very likely that the parallel version will inherit any weaknesses or bugs from the serial version, unless the software is to be substantially re-engineered. Another issue that is likely to arise in practice when one begins with existing serial code is that the developer of the parallel version may well be the person who wrote the serial version. This person is likely to be best placed to understand what the existing serial version does. However, it may be worth considering whether someone else should be allocated the task of writing the parallel version. Experience in writing serial code does not mean that one is necessarily going to be expert in writing parallel applications. A specialist in parallel applications may be able to identify opportunities to parallelise that a non-specialist might miss. In any case, there are advantages in involving someone who is not necessarily committed to the methods and programming solutions employed in the serial version. Even if the developer of the serial code becomes the developer of the parallel version, he or she should seek advice from any colleagues who have prior experience of developing parallel and distributed applications.

3.3.2 What aspects can be parallelised?

The first stage is to identify those computational tasks that can clearly be parallelised, beginning with FOR and DO loop constructions, especially where one iteration of the loop does not depend on values calculated during a previous iteration. Of course, some iterative processes cannot be parallelised. The Lawrence Livermore guide (`www.llnl.gov/computing/tutorials/parallel_comp/#top`) cites the calculation of a Fibonacci series by the use of the following formula as such a non-parallelisable problem. The series is defined by:

$$F(0) = 0,$$

$$\begin{aligned} F(1) &= 1, \\ F(n) &= F(n-1) + F(n-2), n > 1. \end{aligned} \tag{3.1}$$

Clearly, if one uses this formula, the calculation of each number in the series requires knowledge of the previous two numbers. Note that algorithms exist for calculating Fibonacci numbers that can avoid at least part of the requirement to calculate previous numbers in the series in this manner. If there are computations that exhibit the data dependence demonstrated in the Fibonacci series computation, then alternative algorithms should be investigated. This is always good general advice when developing parallel programs. Algorithms that are suitable for serial computation may be inappropriate for parallel computation. In addition, alternative algorithms may be better at taking advantage of the benefits of parallelism.

Next, study the code to identify the key computations in the program. Much of any sizeable program will have extensive sections that deal with issues such as reading in, saving and printing out data, graphical display, interaction with the user, documenting the code itself, and so on. Thus, the important work of the program, its mathematical or computational core, may be limited to only a few key parts of the code. Thus, the aim is to identify those parts of the code which account for most CPU usage. Attempts to parallelise the software should concentrate on these key areas.

Having identified the key computations, review those aspects of the program that limit performance or can be seen as potential bottlenecks. The need to communicate extensively between processes or to engage in extensive input and output activities, including writing large data sets to files or transmitting them across a network, can limit the potential gains from parallelisation. In the kind of distributed computing system that NPL has adopted, it is not possible for the user to interact with the distributed software while it is running, nor can the software display graphical output or other information that might require user interaction during the computation. Thus, it is essential that the parallel code avoids all operations of this kind.

The choice of algorithm will also depend on the architecture of the parallel system and the model of parallel processing that one is using. Shared memory systems are, in general, easier to program. If processes are required to communicate data, then distributed systems that use the Message Passing Interface (MPI) will require different algorithms from those which do not allow communication between individual processors.

3.4 Partitioning the computation task

The aim of partitioning a computing task is to identify all the opportunities that exist to exploit parallelism in a computation. One seeks to define a large number of small computational tasks. There are, in principle, two aspects to this, the first is concerned with data and the second with the computational task itself. Ideally one needs both to divide up the data and the computation. One also seeks to obtain a partition arrangement that avoids duplicating either the computation or the data. However, in some cases, duplication may be unavoidable or may be necessary to reduce the amount of communication required in an application.

These two main ways in which a computational task may be divided up to take advantage of parallel processing are known as *domain decomposition* and *functional decomposition*.

3.4.1 Domain decomposition

In the *domain decomposition* approach to parallelisation it is the data associated with the problem that are decomposed so that each processor can work on a small part of the complete data set. Ideally, the individual small data sets should be of similar size. It is necessary both to identify the data and the tasks to be performed on the data. Often the nature of the task to be performed can determine how the data is to be divided, as the matrix multiplication example later in this section shows.

It is also important to recognise that the data to be decomposed need not simply be input data. It could be output data (e.g. each processor may be given a complete data set but may perform calculations and deliver results only on part of the data), or it may be intermediate data that is derived part way through a computational task and is then sent to other processors. Suppose, also that a computation on a data set proceeds in stages. The same data may need to be decomposed in different ways for each stage of the complete computation.

The complexity of the decomposition task depends on the dimensionality of the data and the nature of the computation to be performed on the data. For a one-dimensional data set, it may be sufficient simply to divide the data into separate shorter lengths for computation by each processor in the system. As a simple example, take the case of calculating the mean of a very large data set: one could divide the data up into chunks, ask each processor to sum the data, collect the results of all these summations and then divide by the total number of data values.

In the case of two-dimensional data sets, a number of decomposition options may be available. However, it is likely that each option will have its own advantages and disadvantages and lead to differing run times and differing memory requirements. This is demonstrated in the following simple example: consider a task involving the multiplication of two square matrices. For the sake of definiteness, we will examine the multiplication of two 6×6 matrices. This problem is written out in full below.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} & b_{16} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} & b_{26} \\ b_{31} & b_{32} & b_{33} & b_{34} & b_{35} & b_{36} \\ b_{41} & b_{42} & b_{43} & b_{44} & b_{45} & b_{46} \\ b_{51} & b_{52} & b_{53} & b_{54} & b_{55} & b_{56} \\ b_{61} & b_{62} & b_{63} & b_{64} & b_{65} & b_{66} \end{pmatrix}$$

$$= \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} & c_{16} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & c_{26} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} \\ c_{51} & c_{52} & c_{53} & c_{54} & c_{55} & c_{56} \\ c_{61} & c_{62} & c_{63} & c_{64} & c_{65} & c_{66} \end{pmatrix}.$$

Suppose we wish to process this matrix in parallel on three separate

processors that can communicate with each other. We can divide the first matrix into three pairs of rows, a_{1j} and a_{2j}, a_{3j} and a_{4j}, and finally, a_{5j} and a_{6j}. Similarly, we divide the second matrix into three pairs of columns, b_{i1} and b_{i2}, b_{i3} and b_{i4}, and finally b_{i5} and b_{i6}.

The calculation could then proceed as follows:

1. Distribute the partitioned matrices to the processors as follows:
   (a) The first processor receives the first pair of rows (a_{1j} and a_{2j}) and the first pair of columns (b_{i1} and b_{i2}).
   (b) The second processor receives the second pair of rows (a_{3j} and a_{4j}) and the second pair of columns (b_{i3} and b_{i4}).
   (c) The third processor receives the third pair of rows (a_{5j} and a_{6j}) and the third pair of columns (b_{i5} and b_{i6}).
2. Each processor now calculates the four individual c_{ij} values where the rows and columns intersect, and stores the results. In the case of the second processor, these would be c_{33}, c_{34}, c_{43} and c_{44}.
3. Now the processors need to communicate with each other to exchange the column data from the b matrix. Suppose that processor one sends processor two the data for the first pair of columns of b, the second processor sends the third processor the data for the second pair of columns of b, and the third processor send the data for the third pair of columns to the first processor.
4. Each processor now calculates and stores the c values for the new intersection between their own pair of rows from a and the new columns from b.
5. There now needs to be a final transfer of b column data and another calculation of the c values for the final intersection of rows and columns.
6. At the end, each processor has all the c values for the pairs of rows they were allocated at the start of the process, which they can communicate to each other or to a central processor for further calculations.

Now consider an alternative approach to the problem. This time we still use three processors but we divide a into pairs of columns and b into pairs of rows. This time, the calculation proceeds as follows.

1. Send data from a and b to the processors as follows:
   (a) Send the first processor the data for the first two columns of a, (a_{i1} and a_{i2}) and the first two rows of b, (b_{1j} and b_{2j}).
   (b) Send the second processor the data for the second pair of columns of a, (a_{i3} and a_{i4}) and the next two rows of b, (b_{3j} and b_{3j}).
   (c) Send the third processor the data for the final pair of columns of a, (a_{i5} and a_{i6}) and the final two rows of b, (b_{5j} and b_{6j}).
2. Each processor then performs the matrix multiplication of its own data sets in the order $a \times b$. This produces a 6×6 matrix, in which each element is a fraction of the total sum for the complete problem.

3. A central processor now collects the individual 6×6 matrices from each processor and forms the result by summing the three partial 6×6 results.

A third possibility also exists. One can send the complete data sets for the two matrices a and b to each of the three processors, together with instructions as to which rows and columns they are to operate on.

Note how the various approaches require different amounts of data to be received and transmitted by processors and different numbers of calculations to be performed. In the case of distributed systems such as the NPL Grid, it is the second and third methods that would be more appropriate because these do not require communication between processors at intermediate stages of the computation, as the final results are assembled from the contributions of the individual processors at the end of the calculation.

In a test of the first two methods that was published on-line, Garrison of the Department of Mathematics at the University of Colorado reported that the method using rows from a and columns from b, was more efficient and faster than multiplying columns of a by rows of b, (`http://cauchy.math.colostate.edu/Projects/Garrison/paper.html`).

3.4.2 Functional decomposition

In *functional decomposition* one concentrates initially on the computational tasks to be performed rather than on the data. If it is found that the computation can divided up into separate tasks, then the next stage would be to study the data requirements for each task. If the data requirements do not overlap with other functional tasks, then one has successfully decomposed the whole problem. Often this will not be the case so that extensive communication processes may be necessary to avoid duplication of data.

Examples of computational tasks that can be decomposed in this way as suggested by the Lawrence Livermore Laboratory's on-line guide (`www.llnl.gov/computing/tutorials/parallel_comp/#DesignPartitioning`) are:

- modelling an ecosystem, where each processor tackles the modelling of a specific population group within the ecosystem and where the behaviour of a particular population depends on that of its neighbours. At each time step each process calculates the current state of its population and then shares the result with neighbours, before the next time step is calculated.

- signal processing, where a signal has to be processed sequentially by three independent filters. Segments of data must pass through the first filter before proceeding to the second, and so on. Once the third data segment has passed through the first filter, then all three filter processes are occupied in filtering data.

- climate modelling, where each process handles a different aspect of the total modelling problem, for example, atmospheric, hydrology, ocean and land surface models. Data will, of course, have to be transferred between the separate functions.

3.4.3 Reviewing the partitioning process

Foster [9] gives a short check-list of points to review at the end of a decomposition process. These are:

- Has the partition produced at least an order of magnitude more tasks than there are processors in the system? If this is not the case, there is little room for flexibility in later design stages.
- Are there redundant computation and storage requirements? If this is the case, the algorithm may not be scalable to large problems.
- Are the tasks of comparable size? If not, load balancing may be a problem – see the next section.
- Does the number of tasks increase with problem size? Preferably, an increase in problem size should lead to an increased number of tasks rather than an increase in the size of individual tasks. If this is not the case, the application may not be able to take advantage of the availability of more processors.
- Are there a number of alternative partitions? If so, this allows flexibility in later stages of the application development.

3.5 Load balancing

Load balancing is the process of ensuring that work is distributed across tasks and processors so that as far as possible all processors and tasks are kept active all the time, that is, it seeks to minimise idle time within the system. The slowest tasks determine the overall performance of the system.

There is extensive literature on the general problem of load balancing in parallel computation, with examples of partitioning algorithms. Most of them are not appropriate for the kind of parallel processing that can be carried out with grid systems of the NPL type, that is, data parallel computation. A good introduction to load balancing methods and algorithms can be found in [9].

For grid systems, the main method of achieving load balancing is to try to ensure that each processor has a similar work load, by creating tasks of the same size. This is typically done by distributing equal amounts of data among processors. Note however, that it is important to recognise that in many applications, especially iterative calculations that continue until convergence has been reached, computations may take varying times to complete, even when they begin with the same amount of input data. In addition, on a grid with heterogeneous resources the performance of the available processors may vary. If there are fewer tasks than available processors then the server will allocate jobs straightforwardly to processors and it will be the slowest processor that limits system performance. If, however, there are many more tasks than processors, it is likely that some of the faster machines will be able to complete several tasks while the slower processors are still calculating their first task. In general, if the aim is the fastest completion of a particular task, then one should consider generating more relatively short work tasks than there are processors available, rather than define fewer, larger tasks.

It will also be the case that aspects of the system performance will not be under the control of individual application developers. There may be several different applications being processed at the same time (as occurs on systems like the NPL grid) and the individual developer may have no knowledge of the other applications and what their computational demands are likely to be.

3.6 Communication

All parallel applications, including those that are data parallel, require communication of some kind. In the data parallel case, it is necessary for the central server to send data packages to each processor and to collect results from each processor at the end of the computational task. Even if there is no communication between the individual processors, it is still necessary to recognise that some part of the complete computational problem must be devoted to communication. In the data parallel model, with no communication between processors, it is possible for the computing task to be communication-limited, if the data sets to be sent to each processor are large but the actual computation time is short.

Another possibility that arises in the data parallel case is that the overall computational task may be divided into stages, each of which may have its own data requirements. One could imagine that the first stage involves sending the same executable and different data sets to the processors, each of which returns a results data set. These results may be combined and then partitioned in a different manner from the original data set, and the new data may then be sent out to the processors with an executable that is relevant to the second stage of the complete process. Combining the first stage results and partitioning them for subsequent calculation are likely to be performed serially, or even off-line by the user or developer of the application.

The main communication issues that developers of parallel applications need to consider are summarised briefly below.

- Two key terms are *latency*, which is the time taken to transmit the smallest possible message between two points, and *bandwidth*, which is the amount of data that can be transmitted and received in one time unit. Bandwidth is normally expressed in megabytes per second. If one sends many small messages then the latency figure may dominate communication performance.

- Communication always involves some cost. The time spent communicating is time that is not available for computation. In processes where synchronisation of communication is important, there is always the possibility that some processors or tasks may be "waiting" rather than calculating.

- If one is using the message passing model of parallel computing, it is likely that communication processes will be visible to, and under the control of the programmer or application developer. In the case of data parallel computing, although the programmer may define the data packages and the work units, the process by which the server

communicates them to the individual processors, and indeed which processors receive which tasks, will generally not be under the control of the programmer.

- In systems which require synchronous communication, a form of "handshaking" will be needed between processors. In some cases this may need to be defined by the programmer and in others it may happen at a lower level that is not amenable to access by the programmer. The need for synchronous communication of data means that it is not possible to begin the next stage of a computation until the stage that requires synchronous communication has been completed. Asynchronous communication allows other computing work to be done during communication processes.

- There are two main models of communication between processors or tasks. The first is where two processors communicate with one processor acting as the producer or transmitter of data and the other as receiver or user of data. The second model is data sharing between tasks. Processors may be able to access common memory blocks, for example.

3.7 Dependencies

Finally, in this chapter, the way in which dependencies inhibit parallelisation must be considered. Two forms of dependency are important. The first relates to program statements and refers to the case where the order in which program statements are executed affects the results. A simple example is the case of mathematical operations that are not commutative. A commutative operation is one that does not depend on the order in which the objects are operated on, so that $ab = ba$. However, vector cross products are non-commutative, so that $\overrightarrow{p} \times \overrightarrow{r} = -\overrightarrow{r} \times \overrightarrow{p}$. When operations are not commutative it is necessary to ensure that they are carried out in the correct order and that the correct input data values needed for the operation are available in memory.

Data dependence arises when the same locations in memory are used by different tasks. An example is where the current value of a variable is overwritten in memory by another value. The following simple loop demonstrates the problem:

```
for i=2:1000
a(i)=a(i-1)*a(i-1);
end;
```

Here it is necessary to know the value of $a(i-1)$ before $a(i)$ can be calculated, and $a(i)$ then becomes the $a(i-1)$ for the next iteration. It is very difficult to parallelise such a loop. For example, in this case, assume that processor 2 has stored $a(i)$ and processor 1 has stored $a(i-1)$. In the distributed memory architecture processor 2 must obtain the value of $a(i-1)$ from processor 1 after processor 1 finishes its calculation. For the shared memory case, processor 2 must read $a(i-1)$ after processor 1 has updated it

Chapter 4

Establishing a distributed computing system

4.1 Introduction

It is not our purpose to give advice on how to make a financial case for establishing a distributed computing system within an organisation. Nor do we aim to discuss how one may demonstrate whether it is both feasible and worthwhile to establish such a system. Nevertheless, it is possible to provide some relevant general advice based on our experience at the National Physical Laboratory.

Firstly, it is necessary to consider whether the organisation has a requirement for a large computing resource that exceeds what is currently available and whether a distributed computing system might provide the answer. Here it is essential to identify not only existing large computing tasks, but the potential for new developments that might exist once a system has been established. Typical issues that one might want to consider are summarised below.

4.2 Obtaining more computing power

There are several ways of obtaining more computing power. These can be characterised as:

1. A monolithic high-end computer that may consist of a single cabinet or several cabinets linked together such as those marketed by Cray Inc.

2. Multiple computers may be linked together to form a cluster. Such systems may contain PCs or Unix workstations and storage media, which are linked together so that they appear to users to be a single system. Cluster computing provides a relatively cheap parallel computing resource, especially for scientific and mathematical work. One of the best-known examples of a cluster for scientific applications is the *Beowulf* project, which began in 1994 in the United States at the Center of Excellence in Space Data and Information Sciences (CESDIS). This

system linked 16 DX4 processors and its success led to the growth of such systems in scientific research. One of the essential features of Beowulf systems is that their compute nodes should be mass produced commodities that are easily available "off the shelf", and are therefore relatively cheap. A book by Robert Brown of Duke University Physics Department, North Carolina, describes Beowulf systems and also provides useful guidance on parallel computing in general. The book is entitled *Engineering a Beowulf-style Compute Cluster* and it can be accessed on-line [3] (`www.phy.duke.edu/~rgb/Beowulf/beowulf_book/beowulf_book/beowulf_book.html`).

3. Finally, a distributed computing environment of the kind discussed in this guide may be suitable for the needs of the organisation. As has been explained briefly in the first chapter of this guide, the concepts of *grid computing* and *distributed computing* are closely related. Many commentators consider distributed computing to be encompassed within grid computing. In their most general sense both terms refer to the concept of pooled computing resources that are available on demand to users. The concept of grid computing seems to have arisen among communities of research scientists who were seeking methods of combining computers across a network to form a distributed supercomputer that would be able to tackle complex computations. Within individual organisations, especially companies and other commercial entities, a local grid or distributed computing system is seen as a way of maximising the use of the organisation's computing resources by making them shareable across applications. Such a concept is occasionally referred to as a *virtual computer* or *virtual supercomputer*.

4.3 Is distributed computing the answer?

It is likely that organisations and groups who have a need for either of the first two solutions outlined above will already have recognised that such approaches are what they require, either because they have specific, ongoing requirements for access to dedicated high-powered computing resources or are likely to have such requirements very soon, or they realise that their competitor organisations, typically other research groups or large research facilities, have already established such systems and they themselves need to do so in order to remain competitive. However, it may be the case that the organisation does not wish or does not have the need to invest in such systems, but nevertheless recognises that aspects of its computing work would benefit from the performance speed-up that distributed computing might have to offer. In such a case, it is worth considering the following questions:

- Does the organisation have a reasonably large resource of networked computers? These could be a mixture of PCs with a range of operating systems, unix workstations and so on. There is no requirement in a distributed system for the donated resources to be identical machines. The important point to consider here is whether the potential resource is large enough to meet the requirements. Clearly, the maximum theoretical gain in computing performance that could arise in an ideal

system in which there was no communication overhead, and 100% of the CPU cycles of a donated machine were available at all times, is limited by the number of machines available.

- How are the existing networked machines used? Are large numbers of them desktop PCs being used by staff mainly for word processing, spreadsheets, and commercial software packages? If so, there is likely to be a potentially large unused computing resource that could be harnessed by a distributed system. A feasibility study carried out at NPL to establish the potential for distributed computing throughout the organisation revealed that for large periods of the day 90% of NPL's computing resource was available for exploitation by a distributed system.
- Is there an established computer support system with expert staff within the organisation who can set up and maintain the central management servers that will be required for the system?
- Is there a nucleus of staff with the appropriate computing skills and interests who have the potential to become expert users of the system and who can be available to help others develop and test applications for the system?
- Is there a range of potential applications that could benefit from access to increased computing resources? A typical application that falls into this category might be a Monte Carlo simulation, where higher quality or more reliable results could be obtained by running the simulation with many more repeats than are feasible at the moment with a serial approach.
- Are scientists and programmers complaining that they could achieve much more from their existing programs and computers if they could get faster throughput of results? An example here might be a processor-intensive or memory-intensive calculation that ties up a single PC for long periods of time, preventing it being used efficiently for other work.
- Are there serial applications that are limited in their output by long run times but which could be distributed relatively straightforwardly? Examples here might include programs that need to be run many times, each with different input data, such as an optimisation or a sensitivity analysis, where faster access to results could be beneficial.
- Is software used for prototyping activities, or to test "what-if" scenarios? More rapid access to results may reduce development times and allow more efficient, less wasteful use of resources and or more "what-ifs".
- Does the organisation have an established Intranet system that is used for internal communication and which may be used for accessing and controlling the distributed system? Web access to the system is not essential but web interfaces greatly increase usability and provide a straightforward method of running and monitoring distributed computing resources and applications.

If positive answers can be provided to the majority of questions listed above, then it may be worth an organisation investing in a distributed system as there are likely to be substantial performance gains for a relatively limited investment in additional computing resources. It is also important to recognise that a distributed system in fact upgrades itself over time. As an organisation renews and replaces its PCs with faster, more modern machines, these devices automatically upgrade the performance of the distributed system of which they are part, with no additional costs to that system.

4.4 Potential sources of management software

It is not the role or responsibility of this guide to recommend specific providers of software for managing distributed systems of the type discussed here. Firstly, it is important to recognise that this is a rapidly developing market and that new providers are likely to enter it. At the time of writing, probably the main commercial software vendors that the reader may wish to consider are IBM, Platform Computing, Parabon Computation, Data Synapse, Sun, Avaki, and United Devices.

In addition, organisations may wish to consider the open source software toolkit provided by the Globus Alliance (`www-unix.globus.org/toolkit/about.html`).

4.5 How the server and management software works

To understand how a typical distributed computing system works, it is necessary to appreciate the role of the server and its management software. Here a general, high-level description of typical tasks will be given. In practice, the detailed operations of a particular vendor's software may differ.

In the first instance it is necessary to have some means of taking a large processing task and dividing it into smaller tasks that can run on individual machines in the network. Although it is possible to imagine that this role might be performed by the central server, it is likely to be the case that the division into smaller work tasks is carried out by the person requesting an application to be run, or by specialist software used to submit tasks and work requests to the server. In cases where the program executable to be distributed is identical for all machines to be used for a particular distributed computing task, and the only variation between machines is likely to be the specific input data to be used by the executable, the definition of individual work units for each machine can most efficiently be left to the user, who, in such cases, has full control over the size of the work units to be distributed. The simplest case might be that a single work unit consists of a program executable and a single input data file that will be read by the executable. The expected program output may also be a single output data file containing the results associated with the particular input data set. Another possibility is that users register with the management server particular executables for distribution, create work packages consisting of input data files, which they submit to the system, and the management

software links each input data file with the required executable which it then combines into a work package to be sent out to a machine on the network.

The central management software sends work tasks and some client management software to machines that are idle and are requesting work from the central server. The server then monitors the progress of each job being run by machines on the network, collects and assembles completed work packages ready for the user to download them.

Management software will also handle those cases in which work packages fail to complete, perhaps because a user of a donor PC has switched it off or is using it intensively. The management software may send the work task to another available machine. An alternative approach is to send out the same work package to several machines at once, on the assumption that one set of results may be returned quickly. This approach is not the most efficient, as ideally one would prefer to process each work unit only once, and for each machine to be working on different tasks. However, it is a suitable method for ensuring that large processing tasks with many work units are completed successfully. Other tasks that the management software is likely to perform include:

- managing security and access, including authenticating users;
- resource management, including defining and managing groups of resources that may be allocated to specific applications because they are high-speed or large memory machines or because they have specific operating systems (Unix machines will be grouped separately from those running Windows). A common reason for defining more than one work group is that some machines may be set aside as a testing or training environment on which applications can be tested before they are allowed access to the full grid.
- policy management such as managing user and job priorities and allowing access to specific groups of resources;
- encryption and secure sandboxing;
- providing a user interface to the system, perhaps via web pages.

It is worth noting that in addition to the software required to manage the distributed system itself, users will need access to tool kits and libraries for developing and testing new applications, and vendors are likely to provide these as part of their complete software package.

4.6 Likely performance gains

It will be clear from the discussion in the above section that there are limits to the performance gains that can be expected from distributed computing systems.

Even if one has available a large network of machines all with equally high performance and which are being used only minimally for routine computing tasks, the performance gains will not simply be determined by the number of machines in the network. One has also to consider such factors as

the time it takes to up-load work units to the system and to send these out to donor machines and the time required to recover and access completed output files. Users who wish to optimise their applications will need to consider such issues as the ratio of computation time to communication time for their individual work units. However, creating work units that have long computation times also has risks. Not all machines in the network may be equally fast, so there is a risk that it will be the slowest machines that govern the rate of completion of an application, especially when there are only a few remaining work units to be completed from a large processing task. In addition, the owners of machines in the network may switch them off while they are tackling a particular work unit. If this occurs towards the end of a long computation task, then the work unit will have to be sent out again.

If there are fewer work units than there are machines in the network, it will be the slower machines that govern completion rates, although, in such cases it is possible that faster machines will be able to do more than one work unit while slower machines are still processing their single work unit.

The key issue for application developers to appreciate is that the network is a heterogeneous resource, which they do not have much control over, and that individual machines may become unavailable to the network at any time.

Chapter 5

Is an application suitable for distributed computing?

5.1 Introduction

This short chapter identifies the main questions to take into account when considering whether an application might benefit from the use of a grid or distributed system. It makes suggestions about how serial programs may be reconfigured to take advantage of distributing computing. See also appendix B for extracts from the advice NPL provides to application developers.

5.2 Suitability of applications

The question of the suitability of an application for grid computing depends on a number of factors. Perhaps it is simplest to consider first those types of application that would not be suitable for a distributed system. The most important of these are:

1. Applications that require interaction with the user via the keyboard or mouse during execution, or other requests for input.
2. Executables that produce graphical output.
3. Applications that depend on the availability of licences to be called at run time, e.g. commercial finite element software packages.
4. Applications that require specific applications, libraries or run-time environments to be installed on agent machines (in some cases this restriction can be circumvented, as discussed later in this section).

In the case of the first two items listed above it should be possible in a large number of cases to rewrite the program so that user interactions and graphic displays are avoided. Typically, input parameters will be read from files and required output will be written to files. Graphical displays of results will be confined to post-processing stages.

If applications require access to multiple licences, then it may be worth discussing the requirements with the provider of the software. It may be

possible to negotiate special arrangements to allow enough licences to be made available to the agent machines for the duration of the distributed applications in question. One of the case studies presented later in this report includes an example of this using the PAFEC finite element software.

In the fourth case listed above, it may also be possible to circumvent this restriction. For example, some commercial software packages allow the user to create stand-alone executables that do not need access to the licensed software itself at run time. Typically the executable may simply need access to certain dynamic link libraries at run time. These DLLs can be packaged with the distributed application executable and sent to each agent as part of the work package.

5.3 The main application types

There are three main types of application that can be run successfully on distributed computing systems such as those offered by United Devices. These are:

1. data parallel applications,
2. batch applications,
3. parallel applications using the MPI protocol.

United Devices, in addition to the applications listed above, are now able to support transaction-based applications, web-services like applications, data intensive applications and applications running within Virtual Machines.

5.3.1 Data parallelisation

Data parallelisation is probably the simplest class of distributed application to understand and to set up. In this case the same operation is carried out on separate elements of a data structure. The executable remains fixed but the data on which it operates changes. Typically each agent works on a small part of a much larger data set, so that each agent can be regarded as performing a distinct, independent task.

Examples of this kind of data parallel application include Monte Carlo calculations where, for example, it is only the seed for the random number generator that varies between agents and this seed may be the only information in the input file that is associated with the executable on each agent. Another example is searching a large database or a large amount of text. Indexing a block of text is an instance of this: each agent may be searching for a different word in the same block, and recording its location. In post-processing, the results of the individual searches can then be combined into an index. Data mining is another instance of this kind of problem.

The aim in this kind of distributed application is simply to speed up computation times so that results can be obtained more rapidly than serial processing of the same task.

5.3.2 Batch applications

It is possible to use distributed systems such as the NPL Grid simply to run batch processes, that is, a sequence of jobs to be run in serial fashion, perhaps overnight or at other times when demand for computing resources is low. Users run such tasks from the command line, specifying an executable and input data, which are then automatically packaged and submitted to the management software as a job. This enables users to submit both code and data on a single command line, and the use of such methods is intended for small jobs or testing programs that may change frequently.

5.3.3 MPI applications

The Message Passing Interface method of parallelisation allows devices on the grid to communicate with each other as well as with the grid server. This is typically done to allow individual processors to provide intermediate results to other processors that need them for further computation. If an application needs such an approach, rather than the data parallel method described above, the requirements should be discussed with the system administrator to ensure that suitable devices can be configured for use with MPI protocols. The application development manuals for the particular grid management software that the system uses will contain advice on how to submit MPI jobs.

5.4 Communication versus computation

In the model of distributed computing that we have adopted at NPL, the desktop PC grid, communication between the central server and the agent PCs takes place essentially at the beginning and end of each work unit. At the start the agent machine receives a programme executable and a data file to work on and at the end of the computation it returns a results file to the server. It is important, therefore, to consider the length of time that the system will be devoting to communications activities as opposed to computation. If work units consist of data sets on which calculations can be performed and results returned very quickly, the user will not be getting full benefit from the system and it may be that in extreme cases the distributed task will take longer to complete that a serial version of the task, especially when one takes into account the time needed to create work units and to assemble the returned results. In the development stages of a new application the user should experiment, if possible, with the size of input data sets and the length of the computation process to ensure that the ratio of communication to computation time remains as low as is reasonably achievable. It may also be necessary to take into account other applications that may be running on the system at the same time, as work scheduling will also have an impact on the total length of time that it takes to complete a task. There is a more detailed discussion of scheduling issues in section 6.2.1 of chapter 6.

5.5 Pre-processing and post-processing

In assessing whether there are any benefits to be gained from distributed computing one needs not only to take into account the ratio of communication time to computation time for work units but also how much time is needed to prepare work units for the system and to collate and study the results. Both these activities may be time consuming and additional programs may have to be written both to prepare the work units and to assemble and analyse results. See appendix B, section B.2.3 for the specific advice that is given to NPL program developers.

5.5.1 Response surface modelling

The combination of Design Of Experiments (DOE) and Response Surface Modelling (RSM) techniques can provide a powerful means of exploring the effects of varying the inputs to a process (the control factors) on the process outputs (responses). One company that uses these methods to design work tasks for a distributed computing system and to study the results from such systems is Bookham Technology. This guide includes an appendix F that explains their approach.

Chapter 6

System administration and quality issues in DC systems

6.1 Introduction

This chapter reviews aspects of the administration and operation of a distributed computing system that are likely to be of interest to application developers. Job scheduling issues, for example, concern not only system administrators but also system users who want to ensure that their jobs run in an efficient manner and return results in the shortest possible time, and with no disruption to other users of the grid or to those who donate their PCs to the grid. In addition, it introduces some of the key concepts of software engineering good practice that should be taken into account in all software development projects, both serial and distributed.

6.2 System administration

6.2.1 Scheduling jobs

Above the level of individual work unit scheduling – the bread and butter of a distributed computing system – there is the question of scheduling entire jobs. In a mixed-use environment, where some users are looking to run very large jobs and others are looking for a quicker resolution of small jobs, the decision as to which jobs should be prioritised over others cannot necessarily be decided automatically. Various strategies can be employed:

Simple time slicing

This is the default scheduling strategy of some distributed computing systems, such as the United Devices Grid MP platform. All open jobs that have been submitted are distributed to the donor computers with an equal priority, although there may be some methods of modifying this, depending on the system. The United Devices system allows users to set priority for their own

jobs, but this does not affect priority between users – jobs submitted by different users will run in parallel, but users who submit two jobs can order them to run in sequence.

Time slicing, although it is a reasonably fair strategy, is inferior to a simple queue if both users are not interested in intermediate results. Assume that two jobs are sent to the system, either of which would run in one day if it had the entire grid to itself. With time slicing, both jobs will finish after approximately two days, and four man-days will have expired while waiting for the results. With queuing, the first job will finish after a day, the second after two days; three man-days are spent waiting for results. If user 1 is interested in seeing the results of the job when they are 50% complete, he will see them in one day with time slicing, and in half a day with queuing. If user 2 is interested in seeing the results at a similar point, she will see them in one day with time slicing, and in one and a half days with queuing. Therefore, if both jobs are equally important, and both need to see intermediate results, time slicing is fair (although still inefficient). In all other cases, queuing provides a much better response to user 1, with no slowdown for user 2.

Donor computer partitioning

If the grid is large enough, it may be possible simply to partition the donor computers so that, for instance, half of them are available only for long jobs, and the other half are available only for relatively fast jobs. This ensures that long jobs can run without being interrupted, and a low latency is available for people who want to run fast jobs. The main disadvantage of this strategy, of course, is that it wastes a lot of available processing power – if there are no fast jobs, the long jobs will be running half as quickly as they should be. The opposite is also true, although it may be acceptable to allow quick jobs to run on the entire grid. This would ensure that, when there were no slow jobs running, the low-latency jobs would be able to access the whole power of the grid, but would cause slow-downs to large jobs if many small jobs were being run at the same time.

Partition by time

If the system is likely to be used by people in only one time zone, it is safe to assume that fast, low-latency jobs will probably be required only during business hours. If this is the case, a system can be set up to partition jobs by time of day – the grid can be available for low-latency jobs during the day, and run the larger, slower jobs at night only. Although this has a similar disadvantage to the donor computer partitioning strategy – if no small jobs are run during the day, larger jobs will take longer to finish than they should – it has one other advantage. With the larger jobs running out of hours, it is possible to relax many restrictions on user experience. Users of networked PCs who have gone home will not notice large amounts of disk-swapping, network use and so on. As long as any individual work unit is finished or suspended before the user starts work in the morning, there is no reason why the user should experience any adverse effects of the system. In this way, it may be possible to run jobs that simply cannot be engineered to behave themselves on the donor computers. The large PAFEC job mentioned in the case study in

chapter 9, section 9.4, was run in a time-partitioned model, since it was too memory-intensive to run during office hours. No jobs were sent out before 6 p.m. or after 7 a.m., and at 8 a.m. a script ran through any computers that were still processing work units they had received before 7 a.m. and cancelled them. Work units had been designed to be an hour or less in length, although the nature of the code meant that some work units could take much longer. The cancelling script ensured that any jobs that had started just before the final dispatch time did not remain around long enough on the donor machine to cause the user any irritation.

Job interruption

In a system in which jobs can be interrupted quickly, it may be possible to create a separate scheduling system, in which jobs can be marked as high or low latency. High latency jobs can then be suspended when a low latency job is created, and resumed when it is over. If jobs can only be interrupted relatively slowly (that is, an entire work unit must finish before the agent will check to see what it should be doing next) this may lead to an unacceptably slow resolution of the urgent job. If jobs can be interrupted quickly, however, this can provide performance that will appear to the smaller job as though it has been submitted to an unused grid. Nevertheless, the larger job may lose a lot of work – particularly if interrupted jobs must be completely restarted (which is usually the case). The loss of efficiency in the larger job must be weighed against the utility of allowing urgent jobs to take over the system temporarily and run immediately. However, this can be a much better option than some of the others, since it allows large jobs to run at all times that the system is not otherwise required, ensuring an almost complete utilisation of the processing power available.

6.2.2 General advice on scheduling

Any of these techniques on their own may meet the requirements but, if required, a more sophisticated system can be developed with relatively little effort. The types of jobs required should be considered carefully – are low-latency small jobs a priority? Are many large jobs expected to be run at once, and if so, do deadlines allow the jobs to be run in sequence on a first-come first-served basis? The principles behind any decisions should be well documented, both to save arguments later and to ensure that decisions do not become fossilised by habit if the way the grid system is used changes in the future. Where possible, users should also ensure that they fully understand how their chosen distributed computing system will schedule if left to its own devices. The vendor should provide adequate documentation to support this understanding – if the user is unsure of how the scheduler works, then the system could behave in unexpected ways when more than one job is submitted.

With clever scripting, it will be possible to limit some jobs in time while allowing others to run quickly and uninterrupted. It is worthwhile automating this process where possible, but allowing some manual override. If the user has daemons monitoring the distributed computing server, they can be configured to maintain a queue of jobs, bumping them up in priority when required. In the United Devices Grid MP Platform, for instance, jobs can be

submitted and closed without starting automatically. This allows a script running every few minutes to determine when one job has ended and another should be started. Users could submit their jobs in the usual way and forget about them until notified that the job has ended.

6.3 File formats

The files used by the grid application as input and output should be both dense (containing no unnecessary information) and easily manipulable. Although most grid systems will automatically compress data going to and coming from the donor computers, the compression algorithms used cannot distinguish useful from useless data. It may be that a results file returned from a grid application is compressed from 10 Mb down to only 2 Mb, but if only a few lines from that 10 Mb file are actually used, it could have been sent far more economically even uncompressed.

It can be tempting, especially with third-party applications such as finite element (FE) packages, to use the output file formats they supply without much in the way of analysis. When running a single FE job, for instance, a user might set up the problem geometry, run the problem, and then look at the results file in a mesh display on the PC. To run a hundred jobs with varying parameters, the simplest possible thing to do would be to simply upload those hundred jobs to the grid and (after a short while) retrieve the hundred results files from the system. But what would the user do with the results? Examining one hundred meshes by eye is a lot of work – and remembering and relating to the input the differences in the hundred results files would be an extremely complicated task.

It is important when designing a grid application that users have a clear picture of what they will be looking at when analysing the results. If looking at the effect of differing input parameters on a mechanical FE model, for example, they will generally be looking for the difference in displacement of a few representative nodes (or even a single node). If the application can be written to return those few displacements instead of the displacements for the entire model, enormous savings can be made in network access, storage on the server, and the time it takes to process the results.

This clear picture will help with the design of the application, since it is important to take a wide view of how the grid application will work.

- How will the job be divided up?
- How will the input files be created?
- How will the application run?
- How will the application result files be turned into a usable result?

The easiest file formats to manipulate will be text files - either simple text files, such as CSV (comma-separated value) files or XML files. Nearly all languages are capable of text file manipulation to the degree necessary to generate input files automatically, and indeed some are particularly suited to this task (for example, Perl and PHP). FORTRAN's rudimentary string-handling capabilities make it rather awkward to use for text

manipulation, but other programs can "massage" data into and out of the formats that FORTRAN is happy with.

XML might seem a rather heavyweight solution to many input and output problems, but it can be particularly useful for hierarchical data. It is can be read and written easily with a variety of helper libraries for different programming languages, and on the server can be transformed using XSLT (eXtensible Stylesheet Language Transformation) into various other formats. Although XML has a reputation for bulkiness compared to flat text files, this is largely unwarranted – XML files are designed to compress well. The same data stored in an XML file and a CSV file will often differ very little in size once zipped – and if there are hierarchical components to the data, the XML file may even be smaller once compressed if it is specified carefully. XML files can also be more tolerant of change than flat text files – a well designed XML parser will ignore elements that it does not know about (assuming they are allowed by the Document Type Definition (DTD), if one is supplied), allowing newer versions of the grid application to run with older input files or vice-versa. This may be necessary when verifying the results of an upgraded version of an application against a previous one. Instead of providing two input files, the same one can be used on both versions, removing one potential difference that may invalidate the comparison.

6.4 Software quality issues

The methodology required to develop programs for parallel and distributed applications is in principle no different from that which is employed for serial program development. There are some basic rules that should be followed for all software projects to ensure the production of software that is of good quality and fit for purpose.

6.4.1 Best practice guide for software development

Previous Software Support for Metrology programmes have led to the publication of a Best Practice Guide on the development and validation of software for test and measurement systems. The methodologies advocated in this guide can be adopted for the development of parallel and distributed applications. The guide is *Best Practice Guide No. 12: Guide for Test and Measurement Software*, which is downloadable from `www.npl.co.uk/ssfm/download/bpg.html`. The guide combines general advice on software development with advice on specific software packages and languages.

The approach of the guide is to base advice on what is called the *life cycle* of software development. Essentially, this approach divides the work of software development into the following stages:

1. Gather user requirements: this stage includes understanding what the user needs from the software, which includes the accuracy of the results, and defining the software environment in which the code will be developed, including any tools that will be employed to aid in software development and validation.

2. Specification: record and agree the functional requirements for the software.
3. Risk level assessment: establish the criticality of use of the software and its level of complexity.
4. Design and code: select the languages and techniques that will be used in writing the software and write the software itself.
5. Iteratively verify, validate and review the code that is written.
6. Deliver the code to the user.
7. Use and maintain the code.

This "software life cycle" approach can be employed for the development of distributed and parallel applications, provided the following points are taken into account.

1. A serial version of the software may already exist that has been written by the person who will be producing the distributed version, or has been obtained from some other source.
2. In such cases it is likely that the distributed version will inherit all the faults and weaknesses of the serial version of the code.
3. Thus, for critical applications it may be advisable to start again from the beginning of the software life cycle and prepare a new set of user requirements and specifications.
4. Even if one begins from a previous serial version it is likely that some changes will be needed to the software or at least to the process by which the data is assembled, simply because of the need to prepare sets of work units for distributed processing and to collate and process the completed work units.

6.4.2 Grid-specific quality concerns

Preparing an application to run on a distributed computing system will often mean changing the focus of the efforts put into software quality. When a scientific application is running on a single computer, the developers are usually interested in two aspects of quality – correctness and speed. A mathematician running a curve-fitting algorithm will care whether the algorithm fits the curve accurately, and whether the answer is available immediately, after a cup of coffee, or tomorrow. Users may be mildly put out if the computer grinds to a halt on all other tasks, or writes to the disk constantly, but they will accept that they are getting the result, and if the computer can do nothing else they will have to forego email or web-browsing for a while.

It should be fairly obvious that distributed computing applications cannot be allowed to behave in a similar manner. Correctness is still important, because if the answer is incorrect it doesn't matter how fast or how many times the software is run, it is still unlikely to be particularly useful.

Speed is still important – there is very little point in running an application two hundred times to speed it up if it could easily be made two hundred times faster on a single computer – but speed is now no longer the major worry of the programmer. The distributed computing system is providing a solution to the problem of speed, and a problem that the programmer does not need much input into – the grid administrators can make the application 10% faster just by adding 10% more donor computers. However if users are denied permission to run their applications on the grid, they will immediately lose an enormous speed boost.

Equal (if not greater) effort should therefore be given to user experience for the owner of the donor PCs. Unlike in Graphical User Interface (GUI) design, however, the aim in distributed computing development is not to ensure a pleasant donor PC user experience, but to try to ensure no user experience. The perfect distributed computing system should be completely undetectable when running – users should not be able to tell the difference between using their computer normally and using it with a grid application running in the background. In practice, there are likely to be little slowdowns (perhaps a noticeable network slowdown for a minute while returning a result package, or the sound of a disk running when the user is not accessing the disk), but a focus on making the software as unobtrusive as possible will yield good results in the long run. If donor PC users are complaining about their work being interrupted, it can be politically awkward to argue for the benefits of the system. If they ask that their PCs be removed from the system, then distributed applications will run slower as a result. Clearly, user experience and speed are linked in grid applications.

Error Handling

Because the application is running on a remote system, extra care must be taken with error handling. Although most distributed computing agents will have a sandbox that prevents the distributed application from overwriting data on the donor machines or taking up too much in the way of resources, a crashing application may tie up resources on the donor machine, and it may be very difficult to examine the cause of the crash if there is no easy access to the donor machine (as there very often will not be). It is therefore important to ensure that applications to be run on the grid are capable of gracefully exiting in the event of error, and that where possible they keep a log of their actions that can be returned to the user or controlling application for later analysis. Although most agents will provide an error log, it will usually be limited to the actions the agent can detect – error levels, messages reported on STDERR, and so forth. A log kept by the application itself can provide much more detailed and useful information.

It is also important to insure that error reporting is all handled through the log, and not through pop-up error boxes. Although a well-designed agent should block pop-ups, it is better to adopt a belt-and-braces approach here and ensure that the program can run in a mode that does not attempt to report errors by pop-up – this should not be a problem if the rest of the program has been designed to run from the command line or otherwise run without a GUI. Pop-up error boxes, even in user applications, can often be confusing and unhelpful for the user – imagine

how much more confusing an error box will be if it pops up for an application that the user of the PC did not even know was running.

Avoiding potentially malicious or damaging behaviour

Distributed computing modules may be able to perform a number of malicious behaviours on the donor PCs, which fall into two main groups: local I/O abuse, and resource hogging.

Local I/O abuse is the more serious of the two behaviours, but also the hardest to do by accident. Examples of this type of behaviour would be modifying configuration files on the donor PC, reading information from local files, deleting files which the donor system requires to run, popping up cryptic error message boxes on the donor user's screen, connecting to local networked machines, and so forth. Some of these are more likely than others, of course – it is unlikely that a programming mistake could cause local files to be read. This would almost certainly have to be programmed deliberately and maliciously, and so is extremely unlikely. It is worth bearing in mind that the donor PC users might still worry about this, since they are unlikely to trust the development team implicitly.

Most local I/O exploitation will be prevented by the agent, but it is still necessary to review code to ensure that such exploitation is not relied on by the program or introduced accidentally. Local disk I/O can be prevented by reviewing all file open code to ensure that only relative paths are used, and that no "up level" path components (such as "..\") are used within the code or allowed by any scripts the code might call. This will help ensure that all I/O will be performed relative to the agent sandbox, and will not attempt to alter files outside this area. Network access can be tested by running the application on a machine without access to the network.

Likewise, GUI code – pop-up error windows, etc. – should be prevented by the agent, but should be tested by eye both by the developer and by those responsible for checking module quality. Since GUI code has no purpose other than soliciting user interaction, it should be obvious that it cannot be required by a distributed computing application running as a background task, and any use of it in an application should be able to be removed without affecting the principal task of the system.

It is also possible that the malicious behaviour might not be performed by the distributed computing module, but by the donor PC or the donor user. If the donor PC is infected by a virus or turned into a "zombie" controlled by a malicious hacker, it is possible that it may be able to gain access to certain information about the distributed computing module, or its inputs or outputs. In corporate situations, where the donor PCs can be controlled absolutely, it is vital to ensure that they are protected from viruses and zombification by regular virus checks, an adequate firewall, and appropriate tools to prevent email Trojans. If distributing to a grid composed of public computers, this control may not be possible. In this case, all the data distributed and collected by the system should be encrypted as much as possible – certainly during transit, but also on the machine during processing if a suitable encryption wrapper is available. This will also protect against deliberate tampering by the donor PC's user, which although rare, may occur in situations where the distributed computing system is installed on corporate

PCs without taking the wishes of the individual computer users into account.

Computer users may resent their computer being used for distributed computing, especially if the agent programs cause a noticeable slowdown in their normal use of the computer. In this case, most tampering with the operation of the system is likely to take the form of users manually ending the process that is running the distributed agent, or disconnecting the machine during peak processing hours (either by turning off the machine or simply disconnecting it from the network so that it is unable to request additional work units). This is easy enough to monitor with the appropriate tools. Machines which are constantly unavailable to the system should be reviewed, and the users consulted. It may be that they have a legitimate reason for wishing to be removed from the grid.

Ultimately, most poor donor experiences of distributed computing systems are caused by badly written applications or poorly configured agents, and can be solved or prevented by appropriate measures such as making sure that applications do not behave maliciously themselves, setting appropriate "back-off" levels for agents to ensure that they do not use up more memory or processor power than is appropriate, and so forth. It is also useful to provide some way in which the donor users can request the system to stop processing for a short while if they are experiencing poor performance which they think might be due to the agent. In some cases this may also show whether or not the distributed application is actually responsible for the slowdown in machine performance, and can prevent the distributed computing system from being used as a scapegoat for all computer problems.

In a similar vein, keeping the donor users informed of what work their computer is doing, and how useful it will be (to the company, and where possible to society and to them) can help keep them kindly disposed to the system and prevent malicious behaviour on their part.

6.4.3 Avoiding dependence on installed system resources

In order to ensure that an application will run with the resources available to the agent and those supplied in the program module, it is vital to perform some test runs in environments other than those used for development. There are several options that will allow this.

Test agents

Since it is important for other reasons to test the program modules using a test agent, if one is available for the chosen distributed computing system, this should be one of the first ports of call. If possible, the build environment should have an option for packaging the complete system and loading it into a test agent. Setting this option up the first time will be slightly more complicated than simply building the first module; but it will be easier to use later (particularly much later, if the user has to return to development of the system after a few months), and it can be integrated into the build procedure more fully – with text-mode test agents it is possible to include an "agent test" task in the build process, which will automatically generate a distributed computing module and test it on the test agent every time a build is performed. This will give instant feedback on whether the changes have broken

anything in the final module, and will ensure that, provided the system is left in a working state, the user will quickly be able to regenerate the latest version of the module no matter how long it has been since it was last worked on.

Of course, if this test agent is running on the development machine, it can be difficult to ensure that a program running is not relying on resources found on the development path, or DLLs that are installed on the local machine as part of the development environment. Where possible, the test agent should be run on a separate drive or at the very least in a separate directory from the development system. This cannot eliminate the possibility that the module relies on external system resources, but it can minimize it.

Other computers

An administrator, if there is one in charge of applications on the distributed computing system, should (as part of the quality procedure) test every module that is to be used on the system at least once, before allowing it to be uploaded and distributed. If this test is carried out on a machine that is kept free of development software and packages, the test agent run can be carried out with a reasonable assurance that it will fail if it depends on anything not present in a normal system environment. It may not be possible to dedicate a machine for this purpose unless it is going to be in regular use in this way, in which case the administrator's own machine, or a designated machine in the development labs can be used.

If the administrator's machine is also being used for software development it might not be possible to guarantee that the machine is free of the external dependencies that might be lacking on the donor PCs. Still, it is much better than testing on the development machine, where those dependencies are guaranteed to be available.

VMWare and similar programs

VMWare and similar software allow complete images of operating systems to be simulated in sandboxes on another computer. For instance, an image of a Windows 98 machine could be running on an Apple Mac. Since these images can be stored on disk and quickly reloaded from the disk file in a known configuration, they can provide the ultimate test for a distributed computing module.

An agent run on such a virtual machine can be run with a "vanilla" installation of the target operating system – i.e. an installation with nothing that would not always be found. Not only that, but a single computer can test the module on many different operating systems without rebooting or tampering with the configuration of the host computer. Finally, the sandbox provided by these virtual machines can be extremely secure, preventing any non-standard access to the host hardware which might allow malicious operation or might be dangerous if relied on.

If the virtual machines are run on machines with a different hardware configuration from the target machine (x86 Linux modules running under Linux on a Mac, for instance, or Windows modules running on an SGI server), this abstraction provides a foolproof way of testing for unusual methods of accessing the host hardware – because the host hardware that a badly-written

application is exploiting does not even exist on the simulator machine. This is quite an extreme test, and in most circumstances will not be necessary, but is worth considering where the potential impact of bad behaviour of the program would be extremely undesirable (for instance, where a program is expected to be run on very large numbers of machines, or on machines provided by the general public).

6.5 Application certification procedure

We recommend using application certification procedures for obtaining approval to run new software on the grid. Essentially this means the user must demonstrate that a series of tasks has been completed before the system administrator will allow the application to run on the grid.

At NPL, we have developed a questionnaire which has to be completed and submitted to the administrator to register a new application. The main questions are listed in appendix D. In addition, we do not normally allow new users of the system immediate access to the full grid. Initially new applications are restricted to a "Training Grid" which a sub-set of machines on the NPL Grid (typically 20 in number) that are dedicated to software testing and development purposes.

Chapter 7

Random numbers and distributed computing

7.1 Introduction

Simulations using Monte Carlo methods are often good candidates for parallel and distributed processing, especially when simulations have to be run many times or when very many random numbers need to be generated. However, the process of parallelising random number generation, a key element of Monte Carlo methods, can introduce artefacts into the results. These arise if insufficient care has been taken to ensure that the sets of random numbers generated on each processor can be regarded as truly independent. It is necessary to ensure that there are no overlaps and no correlations between the random number series produced on each processor. To avoid these problems one needs to satisfy two requirements:

- the period of the generator must be of sufficient length for the application of interest; and
- one needs a seeding strategy to ensure that the number generation processes on each of the machines in the network can be regarded as independent.

In this chapter we describe an algorithm which meets these requirements and which we believe at the time of writing this guide is the only algorithm that can be shown to do so.

7.2 Random number generators: the background

Although pseudo-random number generation algorithms vary in complexity (they may involve straightforward arithmetic, or complex bit-manipulation), the operation is not computationally-intensive and there is no advantage to be gained from parallelising the process itself. However, some processor-intensive jobs that are run on distributed systems make use of pseudo-random numbers,

so it is important that a good generator is available for use on a distributed platform.

Implementing a random number generator on the grid can, in fact, prove to be an ideal small-scale project for a newcomer to the grid and this guide describes in detail what is required. The procedures involved in implementing this example on the NPL United Devices Grid MP 4.0 system can readily be applied to jobs that are more complex.

7.2.1 General principles

Pseudo-random numbers are needed in certain types of computation. For example, they are used in the Monte Carlo method for evaluating uncertainties. Advice and information about this topic can be found in the report by Cox and Harris [5]. Generators vary in complexity of operation, ease of use and ease of implementation in different languages. They also vary in the speed at which they operate, and the "randomness" of the numbers they generate. The randomness is important because it affects the quality of the results of the dependent computation.

Generators are available for producing numbers from different distributions, which can be an important consideration in some applications. For example, the ziggurat generator `www.doornik.com/research/ziggurat.pdf` produces numbers from a normal distribution, while the Wichmann-Hill generator [15, 16, 17], which is the generator recommended by NPL's Mathematics and Scientific Computing Group, produces numbers from a rectangular distribution.

A simple way to evaluate the randomness of a set of numbers would be to count the proportion generated either side of the mid-point of the range. This is a useful check that can easily be made when developing a new implementation of a generator. In practice, there are much more thorough tests of randomness available. In particular there are test suites known as *DIEHARD* and *TESTU01*. The latter contains some individual tests plus three batteries of statistical tests known as *Small Crush*, *Crush* and *Big Crush*, which the best generators are expected to pass. *Big Crush* uses 2^{38}random values, and can take over 24 hours to execute.

7.2.2 The period of the generator

The period, or cycle length of a generator, is an important consideration for lengthy jobs that require vast quantities of pseudo-random numbers. In distributed computing, it is important that the sequences of numbers generated on each processor do not overlap, and this means that the period must be long enough that widely separated starting points in the sequence generated can be chosen for each processor, with no likelihood of an overlap occurring.

7.3 The Wichmann-Hill generator

This generator was developed in the early 1980s [15, 16, 17]. It has been widely used, and is incorporated, for example, into the Microsoft Excel spreadsheet

package. The cycle length is around 7×10^{12}, which is now considered inadequate for some applications, and it has recently been found to fail some tests of randomness. As a result of this, the authors, Hill and Wichmann, have updated the generator and produced an improved version. More information about the new generator, including source code that implements the generator in *C* and *Ada* can be found on the NPL web site at `http://www.eurometros.org/gen_report.php?category=distributions&pkey=21&subform=yes`.

The new generator has a cycle length of approximately 2×10^{36} and passes the *Big Crush* battery of tests.

7.3.1 The construction of the generator

The generator is straightforward to program. It works by taking four separate seed values, and every time a new pseudo-random number is to be generated, these values are updated in a defined way using simple arithmetic. The pseudo-random number is produced by adding fixed proportions of these terms.

7.3.2 Distributed computing and random number generation

Wichmann and Hill considered the use of the generator on distributed computing systems, where it is important that the numbers are random both within each processor and between the processors. They considered the problem of undertaking a Monte Carlo simulation on a highly parallel system with a hundred or more processors and produced a variation of their basic algorithm that can be used for this purpose.

It turns out that the generator can be used in a distributed environment provided the seeds for each processor are chosen carefully. The authors provide a simple algorithm for generating a new seed from a known seed. By repeating the process, a suitable set can be constructed for use on all the processors. Readers who wish to implement the Wichmann-Hill generator on a distributed system are referred to the NPL web site link given above.

7.4 Implementing the Wichmann-Hill generator on the NPL grid

To run a job on the grid it is necessary to farm out a copy of the program to each processor involved, together with a file of data that the program is to process. A version of the program is needed that reads the seeds from a file, and writes the generated random numbers out to a result file. The input file also needs to contain a count of the number of random numbers that the program is required to generate. It was also decided to include a processor identification number in each data file.

7.4.1 A *FORTRAN* version of the generator

Hill and Wichmann supplied source code for a comprehensive generator package, both in the *C* language, and in *ADA.* Since a *FORTRAN 95*

compiler was available, a decision was made to implement the generator in *FORTRAN*. Accordingly the *C* version was converted by hand to *FORTRAN*. This was seeded in the same way that the *C* version had been, and the results of the *FORTRAN* version were checked against those produced by the *C* version, which had also been provided.

Having produced a *FORTRAN* version that generated the correct results, this was then adapted to read an identification number, the seed values, and the number of pseudo-random numbers required, from an input file *seed.txt*. The resulting numbers were written out to an output file *random.txt*, while the identification number and any error messages, or a message indicating successful completion, were written to a log file *log.txt*. The .exe file for this version of the program, called *mp_DISTRAND*, was then suitable for farming out to the processors on the NPL grid. Note that it is a local NPL system administration requirement that all .exe files to be run on the grid must have their names prefixed with "mp_".

7.4.2 Testing the program locally

Before a program can be farmed out to run on the grid, it must be tested using the United Devices Test Agent. This runs locally on the PC, and checks that the software is likely to run on the grid without any problems. In order to run on the test agent, a Software Development Kit (SDK) is required, that has to be installed on the PC. The SDK provides command line programs for building the program module and a data package. At NPL we make the SDK available to all potential users of the NPL Grid via a shared network drive. The example commands below are those that would be required on the NPL system and are used for illustrative purposes only. However, similar commands that take into account local directory and file naming conventions would be required on comparable distributed computing systems.

7.4.3 Building the program module

- In order that the module can be built, the folder containing the .exe file must also contain a copy of *loader.exe*, which is provided with the SDK in the folder *United Devices SDK\UDsdk_v4.0\tools\build* on the NPL network.
- The program module, named *DISTRAND*, is built using the MS-DOS command *buildmodule -e -orandom.txt -olog.txt DISTRAND*. Note that the *-e* option, which calls for no encryption, proved necessary, to circumvent a bug causing spurious characters to appear in the output files when the program was run.

7.4.4 Building the data module

- Only one data module is required for testing on the test agent. An input file, *seed_1.txt*, is prepared.
- The data module is built using the DOS command *buildpkg datamodule_1 seed_1.txt=seed.txt*. Here, *datamodule_1* is the name of the data module created by the SDK, *seed1.txt* is the input file for program

DISTRAND, and *seed.txt* is the name of the file from which *DISTRAND* is programmed to read the seed numbers.

7.4.5 Testing with the test agent

- The test agent is run under windows rather than DOS, by double clicking on *testagent.exe*, which is provided in the folder *United Devices SDK\UDsdk_v4.0\tools\testagent*
- On running the test agent, the user must select the program module and the data module using the buttons provided.
- The name of a result file must be specified, e.g. *result.tar*.
- The RUN button is selected.
- If no errors occur, the results file *result.tar* is created in the same folder as *testagent.exe*, unless the program folder is chosen for this purpose, and the program can be judged a safe candidate for running on the grid.

7.4.6 Registering on the NPL grid

- This is done by a grid administrator with appropriate access on the server.
- The user provides a name for the application (*DISTRAND*), and a name for the program(*WichmannRNG*).
- The grid administrator sets up two accounts, consisting of administrator and personal usernames and passwords to the grid server machine. The administrator account enables the addition and removal of programs. The personal account can be used for making grid runs and monitoring progress; but a set of useful Perl scripts is also available for such purposes.
- Windows Perl must be installed on the PC. It can be downloaded from `www.activestate.com/store/languages/register.plex?id=ActivePerl`
- The NPL Grid Perl utilities must be installed on the PC, e. g. in a folder named *NPL Grid Perl Utils*.

7.4.7 Running on the test trid

- On the NPL system, the Test Grid links a restricted set of some 20 PCs, and is used for checking that programs run correctly before they are promoted to the Live Grid, which links some 200 PCs.
- The file *NPL Grid Perl Utils**options.ini** must be edited to set the username, password, program name (*WichmannRNG*), and workunit name (*seeds.txt*).
- The required input files for all the workunits must be generated, and copied into the folder *NPL Grid Perl Utils**workunits*.

- In a DOS window, the command **CreateJob.pl** is used to start the job running. Any errors will be reported, but provided the command is successful, a job number will be displayed. This number needs to be jotted down.
- The command **GetJobComplete.pl** < *jobnumber* > displays the proportion of the job that has been completed.
- When the job is finished, the command **GetJobResults.pl**< *jobnumber* > is used to retrieve the result files. These are copied into the folder *NPL Grid Perl Utils\results* as a set of files called *results* < *workunitnumber* > *.tar*.
- The command **ExtractFileFromResults.pl random.txt** is used to unpack the results file.

Chapter 8

Distributing specific application examples

8.1 Introduction

This chapter describes methods of distributing applications that are based on specific software. All the software examples given have been implemented on the NPL grid. Our approach to distributed computing is that as far as possible we aim to help metrologists develop applications using software they are already familiar with so that they do not need to recode problems in a different language or learn how to work with new software packages in order to solve the problems that interest them.

8.2 Java and distributed computing systems

As with most interpreted or semi-interpreted languages, compiled Java classes require a runtime system (in Java's case, a Virtual Machine or VM). There are various schemes by which a set of Java classes can be compiled into a native executable; but since they work by carefully packaging up the runtime system, they offer very little benefit in terms of ease of deployment onto a distributed computing grid.

8.2.1 Installing Java on the donor machines

If a large number of Java programs are expected to be run on the grid, it may be best to install a Java VM on each of the donor machines. This means a fairly large up-front installation, but allows the sending of much smaller program modules, enabling the development of "agile" small-scale grid applications. Although this is probably the simplest solution for a grid system which will be supporting Java applications, there are a number of problems:

- The grid system must support a limited access to the resources of the donor machine. If the security model of the grid agent prevents executing programs installed on the donor computer outside the agent's

sandbox, the installed Java virtual machine cannot be called, and the application classes cannot be executed.

- It requires an agreement as to which virtual machines will be supported. Although in theory Java's "Write-Once-Run-Anywhere" byte code should work equally well on any virtual machine, in practice there are occasionally problems with library support for newer features. All developers of applications for the grid system should agree on a set of virtual machines (ideally a set of one!) that will be supported, and should be prepared to test their code on all virtual machines in that set. The grid administrators should also see to it that the virtual machines installed on the donor machines are suitable, and are kept up to date if the developers require it.

8.2.2 Creating a stand-alone module

If it is not possible to install a single VM on all the donor machines, it will be necessary to create a stand-alone module containing the classes of the application and a VM. This approach is simpler. It guarantees that the application will be run on the specified VM – but requires more preparation, and results in a larger program module.

Most VMs can be run locally to a directory structure. The Sun Java Runtime Environment can be transplanted from its install directory and run without problems, and it does not require any registry keys or files saved in user directories. This means that the whole directory can be packed into a module and unpacked at the other end by the agent. Typically, however, these installations are quite large - Sun's VM is 60Mb, and results in a packed module of 26Mb if compressed in its entirety. There are several steps that can be taken to reduce the size of the VM module:

- Carefully paring-down the VM's files will often allow unnecessary weight to be eliminated. Sun's VM has a number of security-related executables which are not required under most circumstances, and many of them will not be required for a given Java application to run. Similarly, localisation files, bootstrap images, and various other files may not be required to run most applications on a particular VM.

- If using Sun's VM, consider using the Pack200 and Unpack200 commands to compress *jar* files before adding them to the module. Pack200 is a compression method specifically designed for deploying *jar* files across the network. Because it is specifically designed for compressing *jar* files it can achieve much greater compression ratios than general-purpose compression algorithms. For instance, by far the biggest file in the Sun VM is the runtime jar file *rt.jar*. In Sun's Java Runtime Version 5, it weighs 32Mb – approximately half the size of the entire runtime installation. Compressing it with *Winzip* will reduce it to 9.9Mb, but compressing it with *pack200* reduces it to 4.3Mb. This technique can be used on any jar files, although just compressing *rt.jar* reduces a typical compressed program module for Java from 18Mb to 12Mb. The downside is that the *jar* must be unpacked on the agent machine with each work unit, and the unpack program (a 124Kb native

executable that does not require Java to run) must the included in the module. Unpacking is typically very fast (a couple of seconds for the 32Mb *rt.jar* file), and saves (6Mb times the number of agents without the module cached) in data transmission per program. It also reduces the storage space load on the Grid server slightly, although this is a trivial gain for most systems. If used on all *jar* files distributed with a module an even smaller module might be created.

8.2.3 Making the module general purpose

Because of the modular nature of Java applications, one possibility when developing multiple applications is to treat "Java Applications" as a program, and test the application classes (which will typically be very small compared to the VM module) as job data – preferably as persistent data which can be sent out once per job rather than once per work unit. This allows a distributed computing system with a program cache to use the same Java module for many different programs.

8.2.4 Choosing the right VM

The performance of a Java application can also be affected by the choice of VM. The Sun Java Runtime Environment is still the most commonly used VM, but for some tasks, the IBM VM has shown better performance. Even sticking with the Sun VM, it may prove useful to explore the distinction between their Client VM (installed by default), and the Server VM (which is included only as part of their development kit, but can be dropped in to replace the client VM – particularly easy if you are creating a stand-alone module). The Client VM is optimised for quick start-up time and good performance in GUI tasks, whereas the Server VM is optimised for object creation and destruction at the cost of a slower start up. This optimisation is intended to help with the large numbers of object creations that web sites or enterprise applications perform, but can also help speed up the processing of tasks with large data sets – and as no user interaction is allowed in distributed computing tasks, there need be no optimisation of GUI. On an inverse modelling task running on a single machine, an analysis that took an hour to run with the Client VM took only thirty minutes on the Server VM.

8.2.5 Memory concerns

Memory control in Java programs can be very simple, but should be approached carefully. Most VMs allow a strict control on the maximum memory that can be allocated to an application – for instance, the Sun Client VM defaults to capping memory usage at 64Mb. If the running application requests more memory than this, the application will fail with an *OutOfMemoryError*. If the application requires less memory, the memory cap can be set lower to ensure that less memory will be used on the donor machine during a run. However, if the memory requirements are hard to predict, the memory cap may have to be set higher – although again, there will have to be a compromise between the needs of the application and the user experience required on the donor machine. Simply put, though, this upper limit on

memory usage in the VM ensures that reasonable limits on memory usage can be enforced and that the donor machines are not forced into using virtual memory, with the inefficient disk-swapping that that entails.

8.2.6 Logoff issues and the Sun VM

The Sun VMs hook into the Windows OS signalling mechanism to detect LOGOFF, CTRL_C, and SHUTDOWN events. The default handling for LOGOFF events is to end the VM and report an error level 143. Java-based distributed computing modules may receive this event too, terminating the job if the user logs off. This is, clearly, not the desired behaviour (since the VM should be running in the system space as a background task, not in the user space). To prevent this, distributed Java jobs using the Sun VM should use the -Xrs command-line flag when calling Java, which instructs the VM to ignore OS signals.

8.3 Comsol and distributed computing

Comsol is a multiphysics modelling package capable of modelling heat transfer, structural mechanics, electromagnetic fields, and other phenomena, depending on the installed modules. Originally intended to work in concert with Matlab's solvers, Comsol is becoming increasingly independent, and is now just at the point where it can be used in a distributed computing system without including Matlab.

Comsol has no intrinsic support for MPI, and so (as with most finite element packages) a single large job may not be able to benefit from running on a grid. However, many problems will be amenable to distribution, particularly those involving a Monte Carlo simulation, or those in which a single problem must be run against all permutations of a known group of variables.

Comsol have begun to replace the Matlab solvers with their own systems, developed in Java. The majority of the Comsol system is now Java-based, meaning that many of the problems encountered when distributing Comsol are generic, and the solutions are discussed in the section on distributing Java.

8.3.1 Comsol's Java API

A Comsol API allows the running of Comsol .m commands, and potentially entire scripts, from within Java applications. An *Eval* command attached to an object from the Comsol API will evaluate Comsol commands, so an entire .m file can be encapsulated within a Java application. Alternatively, a single Java application can be written which will read in a .m file and evaluate it line by line. If the .m file is written so that the results will be in a known variable at the end of the calculation, the Java application can then extract it and write it to a results file using the Java I/O commands. Due to the relatively small size of most .m files, this technique could easily be used to create a generic module that will run many Comsol scripts – although of course this has to be weighed against the possible disadvantages of allowing scripts to be run by modules, as discussed in the section on developing applications for the grid.

At the time of writing there are some features that have not yet been ported to Java, which means that care must be taken when converting a Comsol .m script to run on a grid. Version 3.1i of Comsol does not support a means of dumping an entire Comsol structure to disk (as per the Matlab *save()* function), so the output from the converted program must be extracted either one variable at a time, or in the form of a two-dimensional matrix. Programmers may be used simply to running one model at a time, dumping the data, and then reading it into Comsol to display the results in a graphical format. This cannot be accomplished with the current version, although Comsol say that upcoming versions will support a dump of the Comsol structure in a similar manner to the Matlab *save()* function, so this may not be a problem in the future. However, the lack of a simple dump function can act as a reminder that the proper analysis of the data provided by many runs of a particular simulation cannot be accomplished in the same way that analysis of a single run can be. As mentioned above, one of the problems with planning a distributed computing run of a given model can be the temptation to focus on the processing itself and not with what will be done with the results. Programmers forced deliberately to extract one or more results from a model are at least thinking about what results they want, and what to do with them.

8.3.2 Licensing Comsol

Comsol's licensing model, like Matlab, is based on FlexLM – although with a certain amount of flexibility one can circumvent the problems that might otherwise be experienced. If the agent program being used blocks network access to machines other than the grid server (as most do), the networked licensing system cannot be used. Comsol allows licensing from a local file – this nodelocking system does not require network access or privileged machine information (some agents may block system queries such as local disk labels or network MAC addresses, which are often used to create node identifiers because of their relative uniqueness). A solution similar to that used with the PAFEC licensing system (see section 8.4) might be used easily: the Comsol program module would contain a licence file with licences for every machine on the grid, and a helper application would split out the correct licence before running the main program. This requires the ability to run an identification program on all the grid machines before the module is created (not always a trivial problem, as mentioned in the section on PAFEC), and can be rather inflexible if the population of donor machines is expected to be in some flux (which in most organisations it almost always should be if the best performance is to be got out of the distributed computing system). However, it is a reasonable model if maintained correctly, and most of the solutions mentioned in the PAFEC section will also be helpful with Comsol.

8.3.3 Links

More information about Comsol, which until the end of 2005 was known as Femlab, can be found at the Comsol Homepage (`www.euro.comsol.com/`).

8.4 Finite element modelling with PAFEC

8.4.1 Licence issues

NPL has a special agreement with Pacsys Ltd to run multiple licences for a slightly modified version of the PAFEC finite element analysis system and this is the basis of the observations in this section. However it is expected that the observations here could apply to other combinations of FEA software and underlying grid systems. There are licences for 200 machines, but for various reasons related to operating system, processor type and memory considerations this results in 148 machines licensed and capable of running PAFEC. The average specification of these machines is: 488 MBytes memory, 105 Mflops which is equivalent to a 2.17 GHz Pentium processor, split 50:50 between Windows XP and Windows 2000 operating systems. Our first use of PAFEC in this way to be reported in the scientific literature can be found in Macey et al [7].

8.4.2 Interaction with donor machines

The paging of large intermediate files can be a problem for particular combinations of Windows and hardware, but the latest machines using Windows 2000 or XP and hyperthreading provide minimal donor machine interference, when the DC job is the only background process. It was necessary to ensure that the NPL grid system correctly identified hyperthreaded machines as only having one processor rather than the default two. To further minimise donor machine dissatisfaction and to allow other grid users to develop models during the day, the finite element runs were run out of hours. This was achieved by using a cron job that enables DC jobs at 7 p.m. each weekday and another cron job that suspends jobs at 8 a.m., and triggers another process that iteratively kills any workunit still running at 8.45 a.m. The FEA jobs run all weekend and hence utilisation of the NPL grid by FEA is about 70% with very little degradation by donor use of the machines.

8.4.3 Implementation and efficiency issues

While implementing the PAFEC FEA package from Pacsys Ltd we found that the following issues were important, and it is expected that similar considerations would apply to other FEA packages.

There were some issues running in the NPL grid sandbox on the donor machine, but these were rectified by using relative file names for library files etc. Additionally the FEA system needs to be configured to run exclusively from the command line with no user interaction, any terminal output is suppressed by the agent on the donor machine and if the system is waiting for this user input then it will suspend indefinitely.

To improve control of the system, MS-DOS batch files were used to run the sequence of programs required, including creating individual licences on the fly from a master file. After a run had completed, post-processing was carried out so as to minimise the size of the returned result and to improve the efficiency of the whole job by running this stage in parallel. The batch file was also capable of automatically running many jobs in a downloaded workunit

sequentially, so helping to maximise efficiency by helping diminish the ratio of data download time to the time taken to process the workunit.

By judicious pruning of the PAFEC system it was possible to reduce it to a command-line-only system of 33 Mbytes in size. This compresses down to 12 Mbytes, for network transit, which is an acceptable size. This program module will be stored on donor systems as persistent data, so it is cached efficiently and this will minimise network traffic. The use of a near universal system (using the batch file to control execution) to run workunits made up of a variable number of FEA runs, minimises the number of versions of the program module and reduces cache flushing and network traffic.

It was found that generation of runs which required more than 1 Gbyte of intermediate files produced deleterious effects on the donor machines which, whilst temporary, caused the users of those machines some anxiety. This could be dealt with by better education or by limiting the absolute size of jobs being run in parallel on the NPL Grid.

To deal with errors there are several strategies available. The types of errors are:

- agent errors: e.g. machine turned off;
- program errors: e.g. lack of memory;
- licence errors: unlicensed machine;
- theory errors: input parameters are inconsistent.

It is possible to ask the system to process multiple results for each workunit: there is a much greater chance that a workunit will be returned without error if it is tried on multiple donor machines. However, this is inefficient in terms of throughput, so a simpler scheme of identifying workunits with errors and re-running them as a separate sub-job is preferable. For some kinds of error, such as licence errors, an unfavourable situation can easily arise when most donor machines take an appreciable time to process workunits, but a few machines fail quickly after the licence stage and are then ready to process a new workunit. In this situation 60-80% of all workunits can be unsuccessfully executed by one machine in the grid, so it is essential to eliminate fail-fast types of error. The problem is usually attributable to specific machines and so marking them as unavailable by the NPL grid for FEA analysis is usually sufficient. We have been advised by the developers of the PAFEC that with additional small modifications to the PAFEC FE program, it should be possible to identify many of the program, licence and theory errors and report the conditions back to the server.

8.4.4 Input file generation

Currently the most efficient use of the NPL grid for FEA is to run many FE input files rather than one big analysis. in particular, for studies of the sensitivity of outputs to changes in inputs, a system for generating many input files automatically is required. So, for example, to investigate input sensitivity, we need to create input files with a range of input variables e.g. -20%, -10%, -5%, 0%, 5%, 10%, 20% around an unperturbed value. A Perl script was used

to process a template that read data from a file containing a list of variables and the type of variation was needed. A fragment of the template file might be:

```
MATERIAL
MATE E NU
 21 var1 var2
```

where var1 and var2 are variables for substitution according to the following variable file rules:

```
#E for Epoxy
var1 array {1E9,2E9,4E9,5E9,8E9}
#nu for epoxy +-20% around 0.3
var2 range start=0.24 end=0.36 step=0.03
#Sxx
var3 fixed value=16.5E-12
```

and the variables are replaced with items in an array, or from a range or with a fixed value respectively.

As part of this process, the system generates a directory containing workunits and a directory containing variables to allow easy relinking of results during post processing. However, these directories can be very large and contain many thousands of files, making them very cumbersome to work with. For very large runs, i.e. containing 500,000+ workunits, it is efficient to break a job into smaller sub-jobs. Apart from making file handling more tractable, it also allows the sub-job to start more quickly. Whilst the first sub-job is running, workunit file creation for subsequent sub-jobs and uploading onto the Grid server can take place in parallel. After the first sub-job has been completed it can be downloaded in parallel with running the second sub-job. This means the total time for the job is time to upload the first sub-job + time to run all sub-jobs + time to download last sub-job, which is shorter than launching one very large job.

A further development may be to send simply variables and template files as workunits to the agent PCs and to allow batch processing on each agent to set up local input files and create results, but this may be difficult to debug, as it will need ancillary software on each agent and is only necessary for very large runs.

8.4.5 Performance

For FEA runs it was found that the average increase in speed was 85x over a single processor. The maximum possible was 148x because there were 148 machine available for FEA analysis, so this means that the analysis was using nearly 60% of the available resource. In trials it was found that the "speed-up" varied from 9x for triangular shell element runs, which are mainly limited mainly data I/O requirements because of the very short time needed to do the calculations, up to 112x for a full 3D element run, which has a more favourable ratio of computation to data I/O.

It was found that it was possible to carry out in a month jobs that would take nearly 7 years for a single computer, producing 10 Gbytes of processed output data from 6 Terabytes of result files in the process.

8.4.6 Advantages

Many finite element runs can be carried on the NPL grid out to investigate model input sensitivity in an efficient fashion that can substantially reduce the time to do this on a single machine. Additionally, mistakes in input files are discovered rapidly and can be easily corrected. The ability to re-run a series of analyses rapidly means faster model development. Additionally, the system is useful to the FEA vendor since it allows thorough testing of FEA software. It is possible that a single run of one million FE analyses is many times more than have ever been done before using the FEA software.

8.5 MPI applications on the grid

MPI applications use the MPI or *Message Passing Interface*, standard for message-passing libraries. This interface allows the distribution of work amongst a network of machines with communication between any machines. It differs from the normal distributed computing paradigm used, since it is not restricted simply to communication with a server at the start and end of processing. It is particularly well suited to applications where there is a requirement for interprocessor communication whilst processing work packets, although it can also be used for master-slave configurations.

There are several software instances of the MPI communication protocol, although our experience has been with the MPICH implementation which is the most popular freely available portable MPI implementation.

To produce an MPI application, specific MPI calls must be added to the software to allow parallel running of the software and interprocessor communication. The system can be used for shared memory, clusters and fully distributed networks of machines. The following code fragment illustrates some of the steps required to create an MPI program. This particular example shows how the same program can be used on all the machines in the MPI cluster, but machine id=0 is special since it coordinates the other jobs.

```
char processor_name[MPI_MAX_PROCESSOR_NAME];
                        /* dimension array to store names */

MPI_Init(&argc,&argv); /* initialise with command line arguments*/
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
                        /* find out how many processors are available */
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
                        /* get the id for this processor */
MPI_Get_processor_name(processor_name,&namelen);
                        /* get this processor's name */

n=100;      /* number of runs*/
while (!done)
{
    if (id == 0)
    {
     startwtime = MPI_Wtime();
                       /* master gets start time*/
    }
```

```
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
                      /* broadcast out number of runs from master */
                      /* also used as control semaphore*/

    if (n == 0)done = 1;  /* stop all processes in lockstep */
    else
    {
        mypi = function(id, n);
                      /* the calculation depending on number */
                      /* of runs and the id of the process */
                      /* (so no duplication)*/

        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                   MPI_COMM_WORLD);
                      /* sum all inputs from other processes */

        if (id == 0)
        {
            endwtime = MPI_Wtime(); /* master gets end time*/
            print pi;           /* print results*/
        }
    }
  if (id == 0)
    {
   n=0;  /* set semaphore to close down all processes */
  }
}
MPI_Finalize();  /* close all programs in an orderly fashion */
```

The above program will run on all the designated machines, with one machine as master using n, the number of runs as a semaphore. When each processor has carried out the function then n is set to zero by the master, the results are gathered and summed in this case, and then printed out. This semaphore causes all processes to terminate in an orderly fashion.

On the NPL grid there are two ways of submitting MPI jobs: **ud_mpirun** and **mpsub**.

ud_mpirun allows MPICH-based parallel jobs to be run on the NPL drid. This interface enables any arbitrary MPICH executable to run on platforms with a supported operating system. When using **ud_mpirun**, input can be specified at the command line, or a configuration file that contains **ud_mpirun** input can be specified.

The **mpsub** batch submission utility is a method of submitting both the executable and the data in a single unit of work. It is intended for use with computational requirements that may change frequently. Application users run **mpsub** by submitting all the programs and input data for the application to the NPL grid as a job.

Table : mpsub vs. ud_mpirun on the NPL grid

MPI using mpsub	MPI using ud_mpirun
Uses mpsub to submit the Job	Uses ud_mpirun to submit the Job
Can use any 3rd-party MPI version	Must use MPICH 1.25
Can still run the executable if it is a commercial product and the user does not have access to the source code and cannot recompile and relink it.	Must recompile using MPICH 1.25.
Only the head node must have the MP Agent installed.	Grid MP Agents are installed on all nodes. Results and Errors are on a per-node basis.
You have to decide which computers to use, list them in "host file", and be sure that they are all available dependent upon the third-party MPI implementation.	The Dispatcher picks which of the available Devices to use as nodes based on targeted Device Groups.
If MP Agents are installed on other nodes, administrators may want to set thresholds for the Devices to back them off if the Device resources are in use by another program so that MPI Jobs run serially. This is recommended to optimize performance.	The system monitors polling and can detect when a Device stops polling and becomes dormant. The MP Agent monitors the processes and can shut down the processes if the Job is aborted or fails early.

Using MPI with **ud_mpirun** allows jobs to coexist with other DC jobs on the NPL Grid and allows errors to be displayed and appropriate control to be exercised from the user console. **Mpsub** jobs, however, cannot be controlled using the console, and they require that other DC jobs on the machines involved be explicitly stopped. Because of these limitations NPL has exclusively used **ud_mpirun**.

The user submitting the job created by **ud_mpirun** must have *Create Job*, *Update Application*, and *Read Application* Privileges on the MPI application and also have the *Program Creator* role, and the *Read Device Group* privilege.

For cross platform compatibility we use the Perl version of **ud_mpirun** and use a configuration file *uduserconf*. A required library file *mpichd.dll* is also packaged with the executable. A flag is set to identify the processor type for execution.

The output from a typical job is shown here:

```
ud_mpirun.pl -w -c uduserconf -np 10 -f mpichd.dll  cpi.exe
Creating new MPI Program.
Creating new ProgramVersion.
Creating new ProgramModule.
Creating new ProgramModuleVersion.
Created ProgramModuleVersion: cpi.exe1121326537
Created Data/Workunit 0
Created Data/Workunit 1
Created Data/Workunit 2
Created Data/Workunit 3
Created Data/Workunit 4
Created Data/Workunit 5
Created Data/Workunit 6
Created Data/Workunit 7
Created Data/Workunit 8
```

```
Created Data/Workunit 9
Created Job 806
Waiting for Job 806
0% complete; 0 results
Workunit  2 (E0CF3714-F439-11D9-8DB3-000BDB9274A0)
                  sent to Pan.npl.co.uk (139.143.25.70)
Workunit  4 (E18DDD2C-F439-11D9-8DB3-000BDB9274A0)
                  sent to andalusite (139.143.75.44)
Workunit  7 (E2432FA6-F439-11D9-8DB3-000BDB9274A0)
                  sent to halite (139.143.75.45)
Workunit  5 (E1C97468-F439-11D9-8DB3-000BDB9274A0)
                  sent to Baji.npl.co.uk (139.143.75.32)
Workunit  1 (E08DE4F8-F439-11D9-8DB3-000BDB9274A0)
                  sent to Mirka.npl.co.uk (139.143.52.80)
Workunit  6 (E2067B7E-F439-11D9-8DB3-000BDB9274A0)
                  sent to Lead.npl.co.uk (139.143.35.30)
Workunit  9 (E2BC12EA-F439-11D9-8DB3-000BDB9274A0)
                  sent to Morse.npl.co.uk (139.143.200.11)
Workunit  8 (E2811942-F439-11D9-8DB3-000BDB9274A0)
                  sent to Radium.npl.co.uk (139.143.35.18)
Workunit  3 (E10E99A4-F439-11D9-8DB3-000BDB9274A0)
                  sent to CHESHIRE.npl.co.uk (139.143.15.20)
Workunit  0 (E00B5B46-F439-11D9-8DB3-000BDB9274A0)
                  sent to Pinkpanther.npl.co.uk (139.143.114.23)
0% complete; 0 results
successful result saved in result-267163-index-000004-job-000806.tar
successful result saved in result-267164-index-000007-job-000806.tar
successful result saved in result-267165-index-000005-job-000806.tar
successful result saved in result-267166-index-000003-job-000806.tar
successful result saved in result-267167-index-000000-job-000806.tar
successful result saved in result-267168-index-000002-job-000806.tar
successful result saved in result-267169-index-000001-job-000806.tar
successful result saved in result-267170-index-000008-job-000806.tar
80% complete; 8 results
Workunit  2 (E0CF3714-F439-11D9-8DB3-000BDB9274A0)
                  done on Pan.npl.co.uk (139.143.25.70)
Workunit  4 (E18DDD2C-F439-11D9-8DB3-000BDB9274A0)
                  done on andalusite (139.143.75.44)
Workunit  7 (E2432FA6-F439-11D9-8DB3-000BDB9274A0)
                  done on halite (139.143.75.45)
Workunit  5 (E1C97468-F439-11D9-8DB3-000BDB9274A0)
                  done on Baji.npl.co.uk (139.143.75.32)
Workunit  1 (E08DE4F8-F439-11D9-8DB3-000BDB9274A0)
                  done on Mirka.npl.co.uk (139.143.52.80)
Workunit  6 (E2067B7E-F439-11D9-8DB3-000BDB9274A0)
                  done on Lead.npl.co.uk (139.143.35.30)
Workunit  9 (E2BC12EA-F439-11D9-8DB3-000BDB9274A0)
                  done on Morse.npl.co.uk (139.143.200.11)
Workunit  8 (E2811942-F439-11D9-8DB3-000BDB9274A0)
                  done on Radium.npl.co.uk (139.143.35.18)
Workunit  3 (E10E99A4-F439-11D9-8DB3-000BDB9274A0)
                  done on CHESHIRE.npl.co.uk (139.143.15.20)
Workunit  0 (E00B5B46-F439-11D9-8DB3-000BDB9274A0)
                  done on Pinkpanther.npl.co.uk (139.143.114.23)
```

```
successful result saved in result-267171-index-000006-job-000806.tar
successful result saved in result-267172-index-000009-job-000806.tar
100% complete; 10 results
Finished Job 806 successfully.
```

The *mpresult* script is used to retrieve results, and show the status of submitted MPI jobs. This can also be done using the system console.

8.5.1 Limitations and problems

For a successful MPI run, all the machines needed must be available for the duration of the job, unlike the normal DC model used on the NPL grid. This means the job cannot start until enough free machines are available, and the job may pause or even terminate if nominated machines become temporarily unavailable. If an MPI job requires a lot of interprocessor communication then, if there are significant performance mismatches between processors, overall throughput will be dominated by the performance of the slowest machine. The performance can be influenced by other activity on the machine so it is expected that optimum performance will only be available out of hours.

Some problems have been found using the *mpich-1.2.5* libraries and the errors reported are opaque and unhelpful. Other *mpich* libraries have been tried including *nt-mpich* but these have also caused problems. Other libraries are undergoing investigation.

A problem with GridMP software version 4.2 has meant that some machines have attempted to run more than one MPI instance at any time even though they have only one processor. This problem remains to be resolved, but may be rectified in GridMP version 5.0. The fault resulted in communication problems between processors, and eventual failure of the MPI job.

8.6 Distributing Numerical Algorithms Group routines

For organisations that subscribe to the Numerical Algorithms Group's (NAG) libraries of mathematical software, it is relatively straightforward to include calls to NAG routines in distributed executables. The key task is to identify all the Dynamic Link Library (DLL) files that will be needed at run-time and to package these with the application. In the case of the NPL grid and United Devices software this is done using the -f commands in the *buildmodule* routine.

For further information about including DLL files in programs, see sections 6.4.3 (Avoiding dependence on installed system resources) and 9.5.4 (Boundary elements for acoustics applications: extensions and lessons) of this Guide.

In the case of the boundary element example described in section 9.5.4, which employed a NAG library routine, we list here for the sake of completeness all the files that had to be included with the module by means of the -f commands are set out below:

- Dforrt.dll
- DFORRTD.DLL

- DLL20DDS.dll
- DLL20DDS.lib
- libguide40.dll
- mkl_def.dll
- mkl_lapack64.dll
- mkl_p4.dll
- MSVCRTD.DLL

8.7 Distributing MATLAB

Using the MATLAB compiler it is possible to compile MATLAB code into standalone Windows executables. Note that the methodology described here requires access to the MATLAB Compiler and that the coding advice refers to MATLAB 7.1 and MATLAB Compiler 4.3 only.

For readers whose organisations make substantial use of MATLAB and who may wish to consider use of the MATLAB Distributed Computing Toolbox instead of the approach described here, appendix E provides information about the toolbox.

8.7.1 Creating a standalone executable

Compilation is carried out by entering a single command **mcc -m filename**. Using the command **mcc -m -v filename** outputs the compilation steps and can be useful in identifying errors in coding, e.g., the inclusion of a function that cannot be compiled.

The compilation process generates two files: a component technology file (ctf) file and an executable (exe) file. Both are required in order to run the executable.

8.7.2 Coding requirements

1. A MATLAB script cannot be compiled and should instead be converted to a function by adding a function declaration line
   function[] = function_name
   to the beginning of the code.

2. A function ("**fun1.m**") that is called only by another routine (e.g., an ODE solver) must be explicitly declared by a line
   %#function fun1
   following the function declaration line. Alternatively the **-a** option can be used when compiling the function using the **mcc** command.

3. There are a number of functions that are not supported by the MATLAB Compiler. A list of such functions may be found on *The MathWorks* website at `www.mathworks.co.uk`. The use of the support pages provided by *The MathWorks* is recommended.

8.7.3 Distribution

In order to run MATLAB executables on the grid, the compiled program requires access to the MATLAB Component Runtime (MCR). There are two options for installing the MCR - it can be installed manually on every grid machine, or it can be included in the program module. On running a MATLAB executable, the system path is searched for an instance of the MCR. If the MCR has been installed on the donor machine, the system path will have been updated as required. If the MCR is distributed with the program module, the system path will have to be updated to include the module's version of the MCR before the MATLAB executable runs. This can be carried out using a batch file and so that the system path on the donor computer is unaffected.

The advantage of installing the MCR on each donor computer is that it is large (around 200 Mb) and much transfer time is saved compared to distributing it with every program. Packaging of each MATLAB program is also made quicker and easier (see below). On the other hand, administrative access is required to install the MCR on every machine on the network, and this access may not be available.

Packaging the MCR with the program is more flexible (different versions of the MCR can be distributed if the program has a specific requirement), and requires no administrative access to the donor computers. The disadvantage is that the MCR must be redistributed to the donor machines with every MATLAB program. The MCR compresses well, but is still relatively large (78 Mb on average). For large jobs, caching on donor computers may make this acceptable, but it does increase the communication to computation ratio. There may also be problems with the sheer number of files that need to be included - on the United Devices system used for the NPL grid, the *buildmodule* command is used to create program modules. The number of files in the MCR is so big that listing them all produces a line bigger than the maximum command line size in Windows. To get round this, a simple program module (containing just the executable and ctf file) can be built, and then the other files added later using a ZIP program. Note that because the *buildmodule* executable includes a loader program to the program module, the ZIP program used must be able to preserve the loader when it adds the files. WinZip version 8 has been shown to respect the loader in this way, and can be used to create a module containing the MCR.

Distributing the MCR like this is very similar to distributing Java programs with the Java runtime, and in fact the MCR contains a Java runtime system within it. Some parts of the Java section of this Guide[8.2] may prove useful in determining which method is used to distribute the MCR.

Whichever route is taken to accessing the MCR, the ctf file generated by the MATLAB Compiler needs to be included along with the executable. Both files are required at run-time, and must be packaged with the application. In the case of the NPL grid and United Devices software, this is done using the **-f** option in the *buildmodule* routine, e.g., **buildmodule -oOUTPUT.DAT -ffilename.ctf -e module1 filename.exe**.

Chapter 9

Case studies and examples

This chapter presents a number of case studies of the use of the NPL Grid. The aim is to exemplify some of the principles of software development and good practice described in the Guide. We hope that readers may be encouraged to consider developing distributed applications of their own by comparing their own computing challenges with those presented here.

Note that some of the figures in these case studies are in colour and are best understood in the on-line version or by means of a colour print version.

9.1 Modelling electron transport in quantum dot systems using distributed processing techniques

9.1.1 Introduction

This case study presents the results of a collaboration between NPL and the Department of Physics at Imperial College, in which NPL assisted a PhD student at Imperial College with a Monte Carlo simulation of electron transport in quantum dot systems.

9.1.2 The problem

Semiconducting materials such as zinc oxide (ZnO) can be prepared as nanocrystals small enough to exhibit quantum confinement effects. In such cases they are known as quantum dots (QDs). Quantum dots have discrete energy levels instead of the energy bands of bulk crystalline solids. They can be regarded as artificial atoms. By means of quantum dots, materials can be synthesised with "made-to-measure" electronic properties that are controlled through the nanoparticle size. Examples of such applications are novel photovoltaic devices and single-electron transistors.

To interpret recent experimental data, a robust theoretical model is required and the work described in this case study is based on the theory of Roest [14]. This model defines an *electron addition energy*, that is, the energy needed to add one electron to a quantum dot of diameter d in an assembly of

electrochemical potential μ with $N-1$ electrons in its conduction band, as:

$$E_N(d) = \varepsilon_N(d) + (2N-1)E_C(d), \tag{9.1}$$

where $\varepsilon_N(d)$is the kinetic confinement energy, and $E_c(d)$ is the charging energy. $E_C = A/d$ where A is a fitting parameter. Thus, the quantum dots that are modelled are assumed to have a diameter-dependent energy level structure. The expression for the electron addition energy produces a family of curves, one for each value of N. The spacing between the curves is greater for larger values of A.

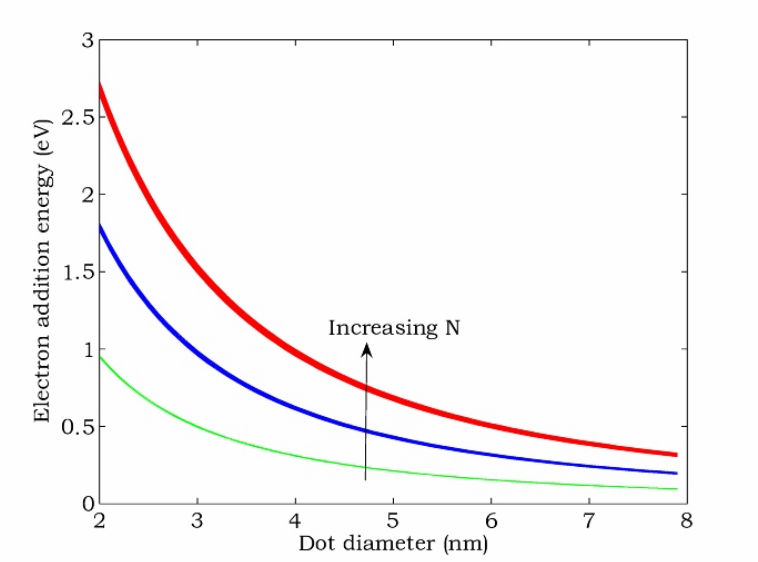

Figure 9.1: Quantum dot energy level structure: electron addition energy for $A = 0.01$ as function of dot diameter

Figure 9.1 shows the electron addition energy as a function of dot diameter for $A = 0.01$ and figure 9.2 presents the addition energy for $A = 0.28$. Note that the figures are in colour in the on-line version of this Guide.

Experimental measurements to study the relationship between the conductance in a ZnO quantum dot device and the number of electrons per dot [14] showed that conductance increased linearly with $\langle n \rangle$ between $0 = \langle n \rangle = 1.5$ and $2.5 = \langle n \rangle = 8$, with a steeper slope in the second range. The mobility was found to be stepwise in these ranges and the mobility in the first and second ranges was attributed to tunnelling between the S and P electron orbitals of adjacent dots respectively. This is shown in figure 9.3.

It was anticipated that a conducting-to-insulating transition would occur at $\langle n \rangle = 2$ due to all the S levels being occupied. No such transition occurs, possibly due to a variation in dot size causing overlapping of S and P orbitals. For a theoretical assembly of identical quantum dots we expect a conducting-to-insulating transition to occur. See figure 9.4 for a comparison and the theoretical and experimental results for identical dots. The quantum

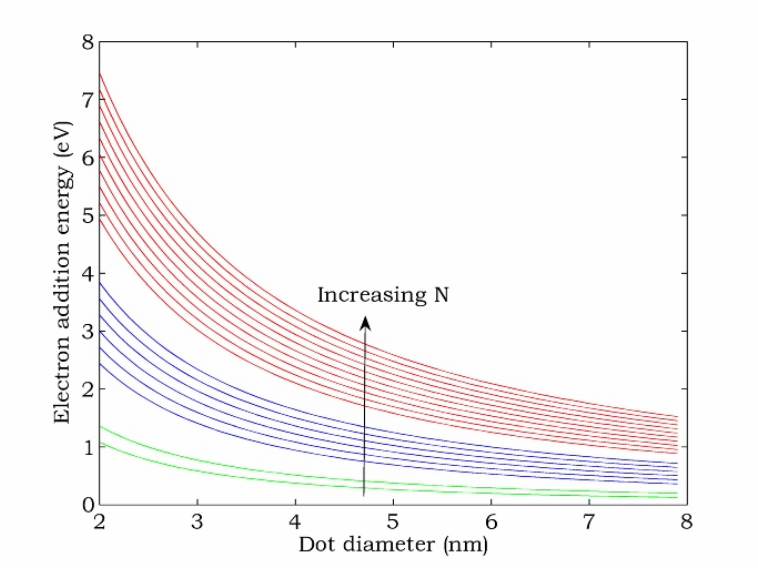

Figure 9.2: Quantum dot energy level structure: electron addition energy for $A = 0.28$ as function of dot diameter

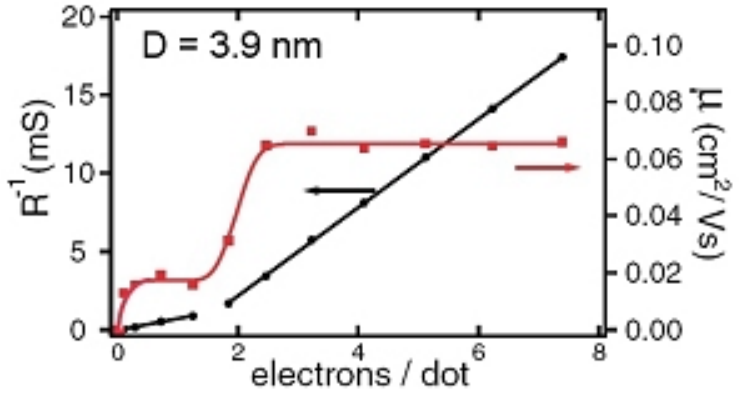

Figure 9.3: Conductance and mobility as a function of electrons per dot

dot model that was implemented on the NPL grid tests the idea that overlapping S and P orbitals are responsible for the observed behaviour.

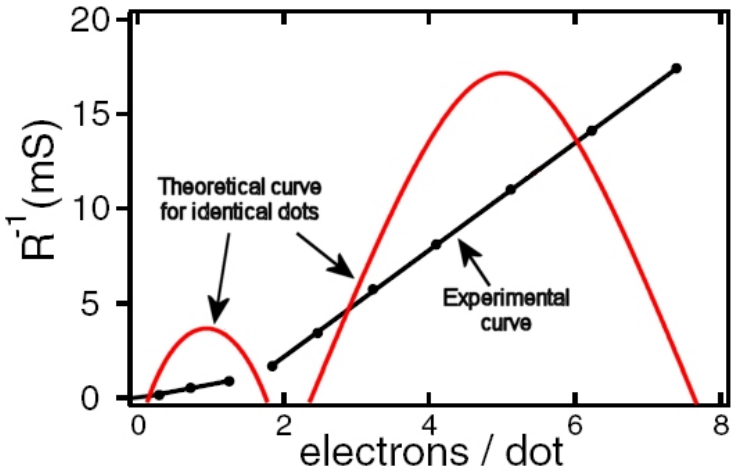

Figure 9.4: Theoretical and experimental conductance compared

The key parameter for the study is the interdot electron transition frequency, which can be expressed as follows:

The transition frequency from a to b $\propto$ probability of level a being occupied $\times$ probability of level b being empty $\times$ tunnelling probability $\times$ probability of levels a and b being aligned $\times$ Boltzmann factor.

This is based on Einstein's quantum fluctuation model [6]. Only resonant tunnelling is possible but quantum fluctuations occasionally bring non-resonant levels into alignment.

The model is based on a three-dimensional lattice – see figure 9.5. The approach is to calculate the Fermi energy from the total number of electrons and from the distribution of energy levels δn represents a source-drain voltage, V_{sd}, applied in the z-direction

The computation is implemented as a random walk: first calculate a transition frequency, $\Gamma_{a\to b}$, for each possible transition of each electron Randomly select the most probable transition and perform it, maintaining the electron population in the $z = 0$ and $z = max$ layers. Repeat until the current flow in the z-direction reaches a steady-state value, I_{sd}. Repeat for several values of V_{sd} and find the conductance from the slope of the I_{sd} versus V_{sd} curve. Increment $\langle n \rangle$ to produce a conductance versus $\langle n \rangle$ curve.

9.1.3 Why use the NPL grid?

Using one computer, the quantum dot model can handle up to $2 \times 2 \times 3$ quantum dot arrays with 20 energy levels per dot (S, P, D and S* energy levels). The runtime is infeasibly long for larger systems. This system is too small to draw conclusions about 3D electron transport and to analyse fully the effects of lattice disorder. Using the NPL grid it was possible to model a $3 \times 3 \times 4$ system with 34 energy levels per dot (S, P, D, S* and F). The grid

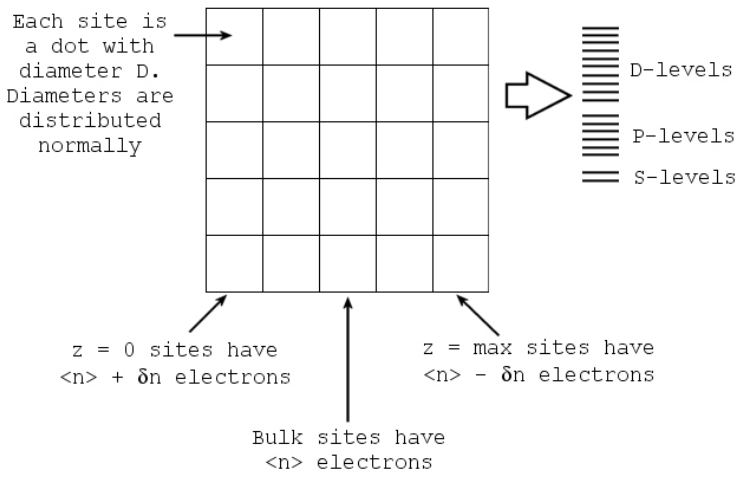

Figure 9.5: Lattice used to model the quantum dot array

made it possible to complete five years' worth of processing in one month. Essentially the approach is as follows.

In the model, the quantum dot assembly is simulated by a 3D cubic lattice of dimensions $x_{size} \times y_{size} \times z_{size}$ where each site corresponds to a dot of diameter D. Each dot has 20 energy levels, where levels 1-2 are s levels, 3-8 are p levels and 9-18 are d levels, and so on. The lattice is randomly populated with $N_{tot} = x_{size}y_{size}z_{size}\langle n\rangle$ electrons. These electrons are distributed such that on average the lattice sites at the $z = 0$ boundary have $\langle n\rangle + \delta n$ electrons, the sites at the $z = z_{size} - 1$ boundary have $\langle n\rangle - \delta n$ electrons and the sites in the bulk lattice have $\langle n\rangle$ electrons. δn is determined from the source-drain voltage, V_{sd}, applied across the quantum dot array.

Once the lattice is set up, the electrons perform a random walk within it. First the transition frequencies are calculated for each possible transition of each electron, using an expression derived using Einstein's quantum fluctuation model. From these transition frequencies a wait time is calculated for each electron, i.e. the time before each one will walk. The electron with the shortest wait time performs its most likely transition. This walk procedure is repeated and source-drain current measurements, I_{sd}, are taken at time intervals from the difference between electron flow in the positive and negative z-directions. The simulation stops after a time long enough such that a steady-state current has been reached. The simulation is repeated many times (~ 500) to determine an average steady-state I_{sd} for the chosen value of $\langle n\rangle$. Assuming the $I_{sd} - V_{sd}$ behaviour is ohmic, the steady-state I_{sd} should be proportional to the conductance through the lattice and therefore the goal of the model is to produce a plot of I_{sd} vs. $\langle n\rangle$.

As far as utilising distributed processing techniques is concerned, the program can be separated into discrete tasks in two ways. Firstly, each individual computer can be assigned the task of generating an I_{sd} vs. time plot for a particular value of $\langle n\rangle$. In total, 49 of these plots are required for values of $\langle n\rangle$ ranging from 0.4 to 10 in increments of 0.2. From these plots, the

steady-state current can be extracted and a plot of steady-state I_{sd} vs. $\langle n \rangle$ can be produced.

For small values of $\langle n \rangle$ and small lattice dimensions, these I_{sd} vs. time plots can be produced quickly but for larger values of $\langle n \rangle$ and larger lattice dimensions the runs will take longer. In these cases each run can be separated into sub-runs in the following way. To produce an I_{sd} vs. time plot requires averaging over 500 simulations. For runs that take a long time, these simulations can be distributed among several computers. For example, 100 simulations can be sent to five different computers and then the average of each of these sub-runs can be obtained. The grid allowed the modelling a $3 \times 3 \times 4$ system with 34 energy levels per dot (S, P, D, S* and F), which was a better representation of a 3-D system than could be achieved with the serial version.

For any particular value of $\langle n \rangle$, a value of conductance needs to be obtained, which requires averaging over approximately 1000 random walk simulations, each with a different seed for the random number generator. To find conductance for 50 values of $\langle n \rangle$ requires 50,000 simulations to be carried out, taking up to 800 days to process. The model was executed by sending small batches of simulations to one of the 200 computers on the grid, with individual run times varying between 10 minutes and 2 hours of processing.

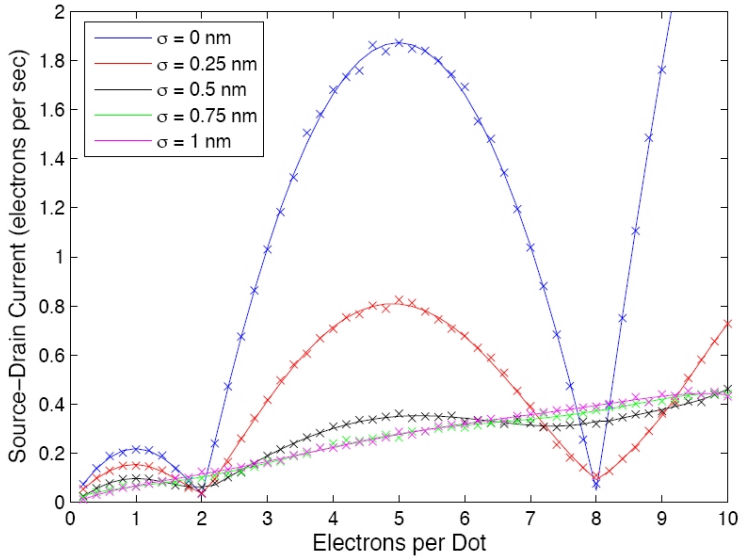

Figure 9.6: I_{sd} versus electrons per dot, for various standard deviations of the dot sizes.

Figure 9.6 presents a plot of the source-drain current against the number of electrons for a particular dot size, with the dot size being allowed to vary by a small amount as represented by the different standard deviations that are plotted. A standard deviation of zero represents identical dots. Note that a conducting-to-insulating transition occurs for identical dots but that this transition disappears as the disorder in dot size increases which is as expected.

If more energy levels are added to the model the result observed in figure 9.7 is obtained.

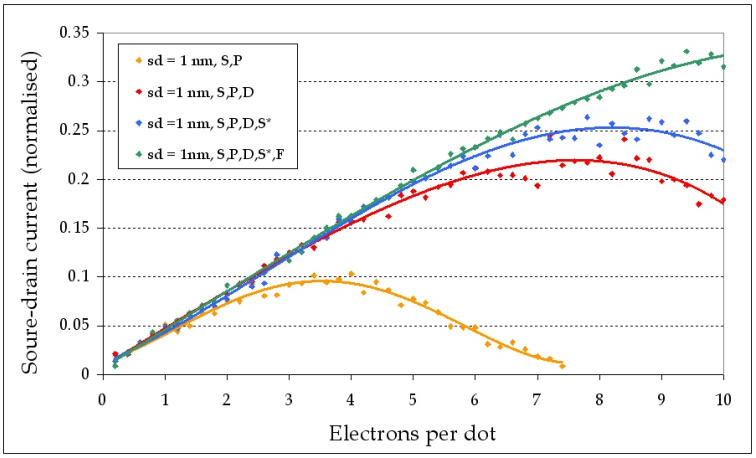

Figure 9.7: I_{sd} versus electrons per dot, for fixed standard deviation and additional energy levels

When the number of levels per dot is too small, the conductance rises and then falls. The conductance rises over the required range only when the S, P, D, S* and F levels are included. Without the use of the NPL grid it would not have been possible to obtain the same qualitative trend as the experimental data.

The work also helped identify future research topics. It was established that quantum dot size disorder washes out the conducting-to-insulating transition:

- Do other forms of disorder (e.g. introducing "voids" into the lattice) have the same effect?
- Can the model be applied to different quantum dot systems (e.g. CdSe, alternative electrolytes)?
- Can different hopping rules be applied (e.g. Miller-Abrahams, variable-range hopping)?
- A study is being carried out into the temperature dependence of the conductance.

9.2 Monte Carlo calculations for ionising radiation applications

9.2.1 Scientific background

Certain cancers can be treated by irradiation – using high-energy particles focused on the cancerous cells to destroy them. At the moment this is usually

done with X-rays, but a relatively new and superior technique uses protons. For a successful treatment it is necessary to measure the dose that is delivered and also how the dose is distributed in the body. Air-filled ionisation chambers are used to measure the dose distribution in a water bath (mimicking the patient) but the introduction of an ionisation chamber itself affects the dose pattern. To ensure that an accurate measurement of the dose can be calculated, we simulate in detail the movement of the proton particles through the geometry of the water bath and the ionisation chamber.

McPTRAN.CAVITY uses the Monte Carlo method, in which the path of a large number of particles is simulated based on the same interaction data but for which random numbers control one or more of the interaction parameters, resulting in a different path for each individual particle. Such calculations are very slow given the large number of interactions protons undergo whilst travelling through matter and also given the very small sensitive mass of air present in the chamber. A typical calculation for one ionisation chamber at a single proton beam energy takes 500 hours, which is reduced to 2.5 to 3 hours on the NPL United Devices Grid. Thus, the grid allows us to generate a database for a large set of ionisation chamber types and proton beam energies, which would previously have been impossible.

9.2.2 Technical details of the implementation on the United Devices grid

When an ionisation chamber is used to measure dose in a clinical proton beam, it perturbs the particle spectrum at the point where we want to know the dose. This perturbation is the result of a complex interplay between the energy losses and scattering of protons in the ionisation chamber materials and the surrounding medium. The calculation of these perturbation factors involves solving a Boltzmann transport equation for which there is no analytical solution, and a Monte Carlo simulation is the obvious method for calculating these numerically. In the Monte Carlo method, the transport of particles is simulated by randomly selecting the location of an interaction, the energy loss, the angular deflection and possibly the generation of secondary particles in an interaction from the probability distributions of all these processes. The quantity of interest (in this case dose or energy deposition per unit mass) can then be scored along the simulated track of the particle.

The Monte Carlo simulation of ionisation chamber response is slow since:

1. due to the statistical nature of the method, a large number of particles needs to be simulated in order to lower the statistical fluctuations on the result below a pre-set limit;

2. the sensitive volume of an ionisation chamber is a small mass of air and the simulation needs a lot of particles interacting in this volume to achieve the desired accuracy;

3. the volume (or projected area) of an ionisation chamber is small compared to the size of the radiation field, which makes the calculation inefficient.

For this application, the NPL grid simply allows the user to run the code many more times than would be possible serially, provided that one uses different seeds for each Monte Carlo process on each agent PC. No other changes to the software are needed. The table below shows typical calculation times for an FWT-IC18 type ionisation chamber, in various mono-energetic proton energies, to calculate the ionisation chamber response with a statistical uncertainty of less than 0.05% using the Monte Carlo code McPTRAN.CAVITY. The figures for the single CPU case are based on a 2 GHz Pentium machines whereas the grid times are the wall clock time taken by the United Device's Grid to complete all work packages.

Proton beam energy	Single CPU	NPL Grid
60 MeV	198 hours	2.25 hours
100 MeV	347 hours	3.84 hours
200 MeV	1107 hours	12.2 hours

9.3 Adiabatic energy transfer in caesium atoms

The operation of an atomic fountain requires that a sample of cold atoms be prepared in a single, magnetically insensitive state (m_F=0). The process of preparation involves the coherent transfer of the atomic population from m_F=3 (initial state) to m_F=0 (final state). Experimentally, this transfer (usually called Adiabatic Passage, AP) is realized by subjecting atoms to two (or more) delayed optical fields. In a series of experiments we have measured the efficiency of transfer (defined as the ratio between population of the final and initial states) and its dependence on various experimental parameters. In order to support this experimental effort we have also performed theoretical studies on AP. The theoretical model has two objectives:

1. to find optimum conditions for efficient transfer (laser intensities, pulse durations, delay between pulses, etc.),
2. and to determine the limit to the transfer efficiency that can be achieved in AP in a multilevel system.

In the numerical calculations it is important to take into account all the relevant energy levels and, at the same time, to keep the complexity of the system at such a level that the computation time does not exceed reasonable limits. The energy structure of the system includes 14 levels. Thus the matrix of the Hamiltonian (H_0) of the system includes energy of 14 atomic states (diagonal elements) and interaction with two laser pulses (off-diagonal elements). The behaviour of the system is fully described by the so-called density matrix ρ (matrix 14*14, where the diagonal elements represent the probability of atom being in a given state and off diagonal element refer to coherences between atomic states). The evolution of the system during the transfer is described by the equation:

$$\rho^{\cdot} = -\frac{i}{\hbar}[H_0, \rho] - \Gamma. \tag{9.2}$$

In order to solve this differential equation Matlab function *ode15s* is used. The code includes three modules:

1. The main module defines the initial state, specifies AP parameters, calls the *ode15s* function that solves the equation and stores the result (final state) in a file.

2. The second module defines the non-linear differential equation that is being solved by the *ode* function.

3. The third module contains the definition of the Hamiltonian, laser pulses (temporal envelope) and atomic/laser field parameters.

Since the optimisation of the AP performance requires solving the same equation for different experiment parameters, we decided to use the NPL Grid to perform the computation. In particular, we focused on the dependence of AP efficiency on the amplitude of the laser pulses. The objective of that exercise was to establish optimal conditions (in terms of the amplitude of two pulses) for the AP. Since the amplitudes of the pulses were varied between 0-6 (Rabi frequency) with step 0.5 the optimisation requires 169 runs of the essentially same code.

To adapt the program to computation using the grid, the main module was modified in such a way that the variable parameters (amplitude of the pulses) were read from the input file, while the other transfer parameters were specified in the main module. Secondly the name of the file with results was defined in a dynamic way. In order to avoid confusion in connecting the results to variable parameters the name of the result file included a part that referred to the relevant input file.

The results of the modelling were compared with experimental data and good agreement was found in terms functional dependence and absolute values. The studies on AP are described in [4].

9.4 Using commercial FE software on the NPL Grid

9.4.1 Input sensitivity analysis of finite element modelling – summary

To use finite element modelling most effectively it is important that the effect of input uncertainty is understood. This makes robust modelling possible and enables the user to make a much more realistic comparison between simulation and test. Additionally, by fixing the limits on the output uncertainty of a model it is possible to determine the required uncertainties for materials property input parameters. This may determine the choice of model depending on the associated cost of getting the data to a requisite quality.

The simplest way thoroughly to investigate the input sensitivity of finite element models is to generate variations spaced around an unperturbed value and to evaluate the model for each of these input parameters. So, for example, a simple linear analysis may be carried out with a varying value of a material property or a model input parameter like element size or shape. For simple analyses with few parameters this can be carried out quite quickly, but more realistic models will require massive computational resources to enable this type of investigation to be carried out in a reasonable time.

Two finite element models have been investigated: the first a simple linear elastic NAFEMS[1] benchmark and the second a full 3D non-linear piezoelectric analysis. At the time of writing, the data from these analyses has had some preliminary processing, but the work is still in progress and there is the potential for much more insight to be gained from the results as the project progresses.

9.4.2 Finite element analysis using the NPL grid

We are using the PAFEC FEA package from Pacsys Ltd. This system has been modified to run exclusively from the command line with no user interaction. We have a special agreement with Pacsys to use 150 licences, which are created on the fly from a master file. The PAFEC System is 33 Mbytes in size, this compresses to 12 Mbytes and is stored on donor systems as persistent data, so it is cached efficiently and network traffic is minimised.

There are various errors that can occur whilst running jobs. It would be possible to run multiple instances of each workunit, to increase the chances of one completing without error, but this would impose a large computational overhead, so a simple strategy of checking errors at the end of a run, and sending out faulty workunits again was implemented.

9.4.3 Input file generation

To investigate input sensitivity we need a range of input parameters e.g. -20%, -10%, -5%, 0%, 5%, 10%, 20% around an unperturbed value. To generate input in an efficient manner for many parameters a series of Perl scripts was used to automate the process. These scripts worked on variable files, which had the types of variability for each parameter in them, and, combined with template files, generated valid PAFEC input files. Each of these files was a workunit for the total job and was also used in post-processing. See section 8.4.4 of this Guide for more information about this process.

The directories containing these files became very large, containing many thousands of files, and were extremely cumbersome to work with. Therefore for very large runs, i.e. containing 500,000+ workunits, jobs were broken into sub-jobs. This enabled the sub-jobs to start more quickly since, whilst the first sub-job was running, workunit file creation and uploading onto the Grid server could take place in parallel. After the first sub-job had been completed it could be downloaded in parallel while the second sub-job was running.

9.4.4 NAFEMS benchmark LE5: Z section cantilever

This is a simple 3D linear elastic analysis with minimal material input parameters and was used to develop the project methodology by investigating different types and aspect ratios of elements. Material input parameters, Young's Modulus and Poisson's ratio, were varied by -10% to +10%. Three variables were used to control the number of elements, thickness of elements

[1] NAFEMS is an international association for the engineering analysis community. As part of its work it publishes benchmark tests for finite element software. To learn more about the organisation see `http://www.nafems.org/`.

and their aspect ratio. Various different element types were examined, including thin shell, thick shell, triangular shell and full 3D brick elements.

The total computer time used for this job was 196 days of processor time, which took 2.3 days real time. This is an average increase in speed of 85x over a single processor and varied from 9x for triangular shell element run to 112x for a full 3D element run. As expected, the Young's modulus had no effect on the benchmark model output. See figure 9.8. (Note that figures are in colour in the on-line version of this Guide.)

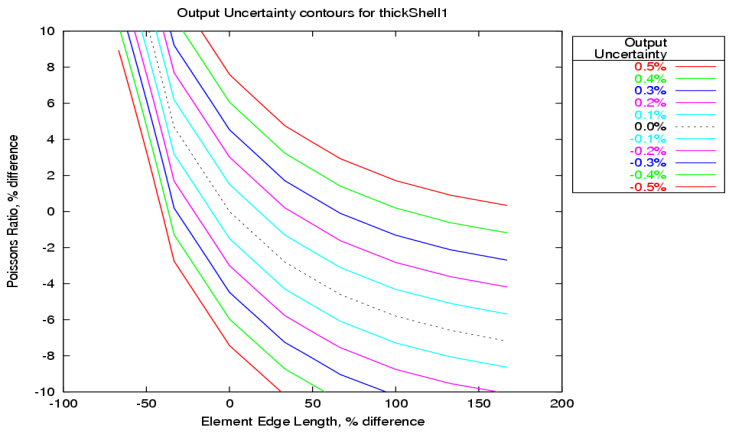

Figure 9.8: A typical output plot showing contours of output uncertainty vs Poisson's ratio and element edge length.

9.4.5 Piezoelectric analysis

A truncated cylinder of the piezoelectric material PC5H with a diameter of 10 mm and an axial length varying from 3 mm to 10 mm was analysed. The specimen was embedded in an epoxy cylinder with the elliptical face at its centre and the axial displacement was measured along the major axis normal to the elliptical face.

There were 23 material input parameters for this problem, however these were reduced to 17 material variables because of symmetry considerations, since the piezoelectric material is transversly isotropic.

Realistic ranges for material input parameters were chosen and it was found that the sensitivity analysis needed 1.7 x 10^{14} runs, which would take 970 million years computer time. Thus the job was broken down into three analyses, looking separately at the specimen mount parameters, the transversely isotropic material parameters and the dielectric/piezoelectric properties, to be combined afterwards. This approach needed only one million runs, which would take 6.75 years of computing time, but which took a few weeks of real time on the Grid. The peak speed for the sub-jobs up was 112x that of a single machine. Each of the three analyses was processed to find the

combination of material input variables that gave the maximum and minimum values of the output graph at each of its 25 points. See figure 9.9.

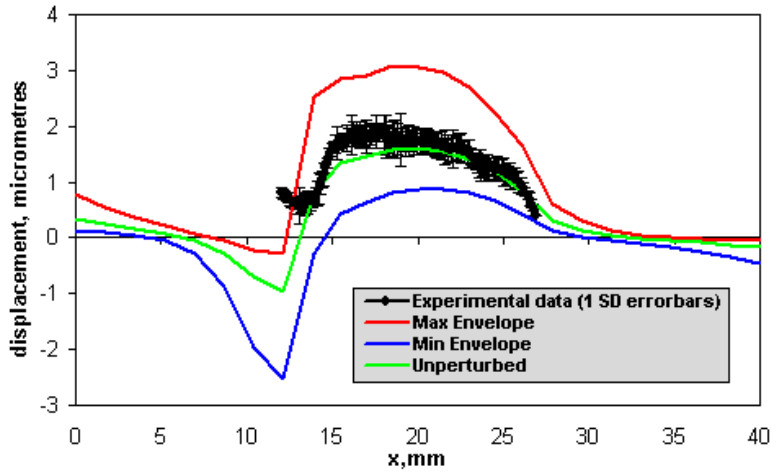

Figure 9.9: A typical output plot of displacement vs element edge length showing a comparison of experimental data with the maximum and minimum envelopes of the results after varying the input parameters.

9.4.6 Conclusions

The NPL grid allows efficient use of spare processor cycles that can be used to carry out many finite element runs to investigate model input sensitivity and the interactions between material input parameters and model parameters.

Further work is needed to understand better the multi-dimensional relationships discovered, perhaps using parallel coordinate mappings, and more detailed comparisons with analytical models and experiment will be made.

9.5 Boundary elements for acoustic applications

9.5.1 Introduction

Manufacturers of underwater acoustic transducers often wish to understand the signal produced a long way from their devices, which is called the “far-field behaviour”. Measuring this far-field behaviour requires a large volume of water. In general large volumes of water are not ideal environments for taking measurements as the water movement and temperature are difficult to control. The difficulty of taking accurate measurements in a large volume of water means that a calculation that uses data measured close to the transducer (the near-field) to predict the behaviour far away from the transducer is very useful.

9.5.2 Problem formulation

The equation that describes the behaviour of static acoustic waves at a fixed frequency in a medium of constant properties is the Helmholtz equation, written:

$$\nabla^2 p + k^2 p = 0, \tag{9.3}$$

where p is the (complex-valued) acoustic pressure, $k = 2\pi f/c$ is the wavenumber, f is the frequency in s^{-1}, c is the speed of sound in the medium, and $\lambda = c/f$ is the wavelength. This equation can be rewritten as an integral over a closed surface, and the surface integral formulation can be solved numerically using boundary element methods if the pressure values on a closed surface are known.

The solution of the associated boundary element problem requires the construction and solution of a system of linear equations $A\mathbf{x} = \mathbf{b}$, where A is a fully populated $n \times n$ complex-valued matrix and $\mathbf{x}$ and $\mathbf{b}$ are vectors of length n, where n is the number of elements used in the representation of the closed surface. In general, the surface representation requires at least three elements per wavelength to produce accurate results. This restriction means that underwater applications, for which $c \approx 1500\ ms^{-1}$, require element dimensions of less than 5 cm for frequencies above about 10 kHz.

Much of the near-field measurement data required for the NPL application is gathered over a cylinder of radius approx 0.3 m and height approx 1 m. These dimensions lead to a total surface area of about 2.5 m^2, which would require around 1000 elements measuring 5 cm by 5 cm. If such a problem were to be solved using boundary elements, a 103 by 103 matrix would be required, which can cause problems with computational storage and memory requirements. In addition, the slowest part of the solution process is the calculation of the components of the $n \times n$ matrix.

The number of elements required can be decreased if the closed surface is axisymmetric, which is the case for measurements taken on a cylinder. Data measured on a cylinder is of the form $p(r, \theta, z)$ where r is the (fixed) cylinder radius, z is the height and θ is the circumferential angle. It can be shown [11] that if the measurement data is written as a Fourier series in terms of θ:

$$p\,(r, \theta, z) = \sum_{m=0}^{\infty} \{a_m(r, z)\sin(m\theta) + b_m(r, z)\cos(m\theta)\} \tag{9.4}$$

and the formulation of the boundary element method is altered slightly, then each of the $a_m(r, z)$ and $b_m(r, z)$ can be treated as providing the input values for a stand-alone two-dimensional problem. The two dimensional problem can require as few as $\sqrt{n}$ elements where n elements were required for the same problem in three dimensions.

If m runs from 0 to M in equation 9.4, then a single run of a model with a large number of elements can be replaced by $2M - 1$ independent models that can be solved more quickly. Since the two-dimensional problems are independent of one another, the problem is ideal for solution using distributed computing.

9.5.3 Results

The technique described above has been used on two data sets that were previously too large to be modelled. The two data sets are signals from the same device at 13.9 kHz and 27.5 kHz. The"input" data sets are shown in figure 9.10. Note that the 27.5 kHz data seems to behave in an unlikely way near to the base of the cylinder, so the data there may be corrupted. The data sets also have extremely localised maxima and amplitudes vary significantly around a circumference, as is shown in figure 9.11. This figure shows the central ($z = 0$) line of the two data sets. This large variation in amplitude means that a comparatively large number of components are required in the summation in equation 9.4: the 13.9 kHz data required 31 terms and the 27.5 kHz data required 51 terms.

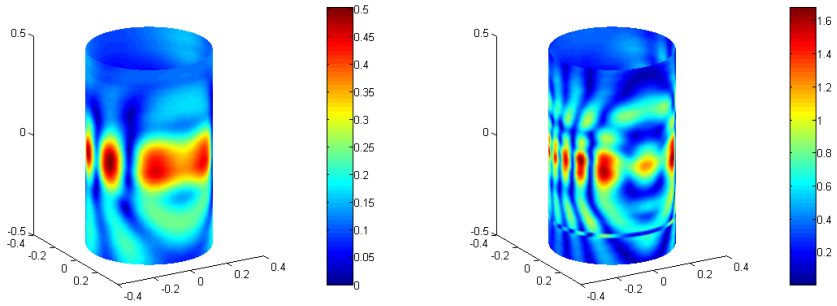

Figure 9.10: Pressure measurements at 13.9 kHz (left) and 27.5 kHz (right): note that colour scales are not directly comparable.

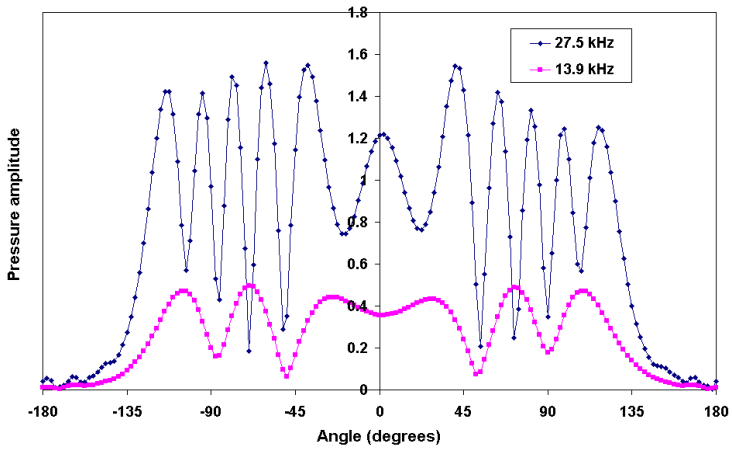

Figure 9.11: Central line ($z = 0$) of the two data sets. Absolute values of the two curves are not comparable with one another.

Each of the input data sets was used to calculate the results at two sets of output points. The pressure values had been measured at these output

points so that the model could be validated against the measurement. The output sets were cylinders of various heights and radii.

Figures 9.12 to 9.15 show comparisons between the measured and calculated pressure amplitudes. Figures 9.12 and 9.13 are the results of the calculation using the 13.9 kHz data, and figures 9.14 and 9.15 are the results of using the 27.5 kHz data. The results shown here indicate that the method predicts the pressure distribution well, and detailed examination of the results shows that the absolute value of the results is close to the measured values for the most important areas.

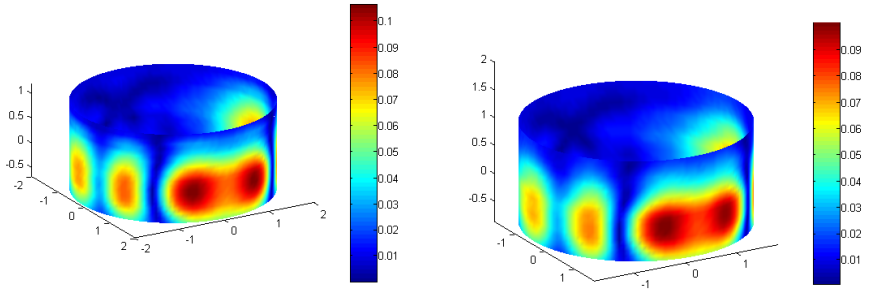

Figure 9.12: Measured (left) and calculated (right) results for a cylinder radius 2 m and height 1.805 m from input data at 13.9 kHz

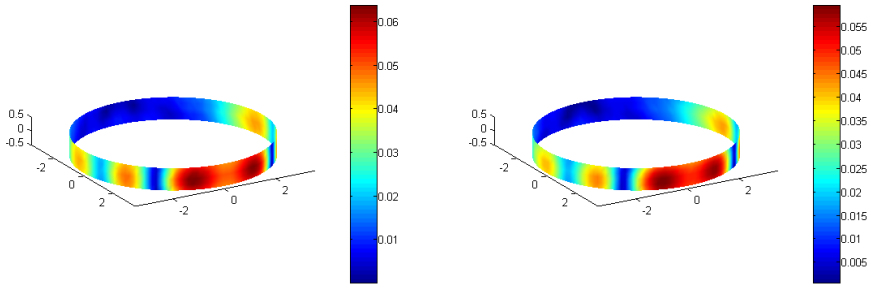

Figure 9.13: Measured (left) and calculated (right) results for a cylinder radius 3.5 m and height 0.8 m from input data at 13.9 kHz

Table 9.1 shows some results of CPU time and elapsed time for calculation of results for several jobs, where the elapsed time is calculated by subtracting the time at which the job was submitted from the time at which the final result was obtained. The jobs have been run using different meshes and different numbers of Fourier components, resulting in different amounts of CPU time. It is useful to remember that the CPU time is likely to be less than the elapsed time for an identical job run on a single PC, particularly if the PC is not dedicated solely to the calculations.

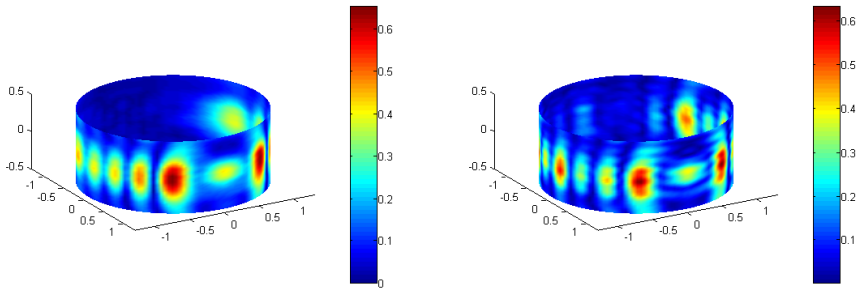

Figure 9.14: Measured (left) and calculated (right) results for a cylinder radius 1.221 m and height 0.96 m from input data at 27.5 kHz

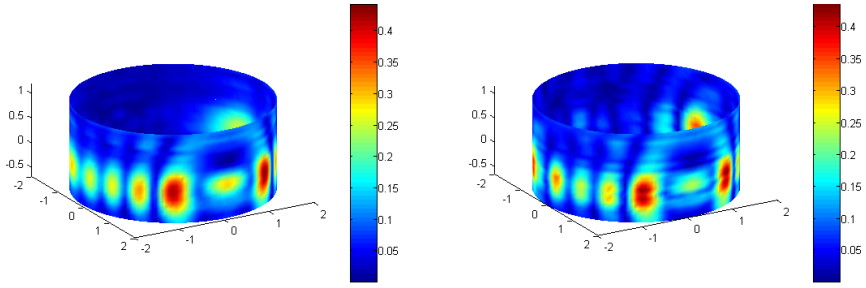

Figure 9.15: Measured (left) and calculated (right) results for a cylinder radius 2 m and height 1.805 m from input data at 27.5 kHz

CPU time (h/m/s)	CPU time (s)	Elapsed time (h/m/s)	Elapsed time (s)	Speed-up ratio CPU/elapsed
10h 34m 32s	38075	38m 55s	2355	16.3
10h 43m 44s	38624	42m 45s	2565	15.1
13h 32m 16s	48736	48m 3s	2883	16.9
13h 37m 9s	49029	45m 36s	2736	17.9
138h 31m 19s	498679	7h 23m 22s	26602	18.7

Table 9.1: CPU times, elapsed times, and speed-up ratios for several calculations; h/m/s is the time expressed in hours, minutes and seconds. Speed-up ratio is the ratio of CPU time to elapsed time.

9.5.4 Extensions and lessons

The software requires several sets of input parameters to run: the definition of the geometry of the boundary element mesh, of the points at which results are to be generated, and of the frequency, sound speed, and density. In addition, each Fourier component needs a set of boundary conditions defining the pressure over the boundary element mesh. These input parameters fall into two categories: those that remain the same for all of the Fourier components (geometric quantities and material properties), and those that vary from component to component (boundary conditions). The input of the parameters is handled using two files, one for those that are the same for each component (*common.txt*) and one for those that vary (*bc.txt*). This separation of parameters means that the file *common.txt* only needs to be sent out once to each PC used in the computation, whereas a different *bc.txt* file will need to be sent for each of the Fourier components. This can save time and bandwidth if a small number of PCs are doing most of the calculation.

The executable software used for the calculations described above was created from Fortran source code without the use of library routines. A potentially useful extension to the software would be the use of library routines for the matrix inversion steps of the calculation. An attempt was made to create a version of the software that used routines from the NAG library. The main challenge in using NAG library routines within the software was identifying which of the files that form the NAG library were necessary for the routines used in the code. Initially these were identified using a trial and error approach, although some guidance was available in the NAG library documentation. It was also difficult to generate an executable linked to the appropriate library files that would run correctly on a machine with a different directory structure.

The final version of the module that would be distributed was a significantly larger file than the version without the library files (7.1 Mb where the version with no library files was 0.54 Mb). This is unsurprising because the combined size of the library files was about 17 Mb, but it may slow the distribution of the module. At the time of writing, the version of the software using the NAG library has not yet been used to generate results.

Bibliography

[1] G.M. Amdahl. Validity of single-processor approach to achieving large-scale computing capability. *Proceedings of AFIPS Conference, Reston, VA*, pages 483–485, 1967.

[2] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computing: numerical methods*. Athena Scientific, Belmont, Mass. USA, 1997.

[3] R.G. Brown. *Engineering a Beowulf-style Compute Cluster*. Website, 2004. `www.phy.duke.edu/~rgb/Beowulf/beowulf_book/beowulf_book/beowulf_book.html`.

[4] W. Chalupczak and K. Szymaniec. Adiabatic passage in an open multilevel system. *Phys Rev A*, 71:053410, 2005.

[5] M.G. Cox and P.M. Harris. Software specifications for uncertainty evaluation. Technical Report CMSC 40/04, National Physical Laboratory, Teddington, UK, 2004.

[6] A. Einstein. On the current state of radiation problems. *Physikalische Zeitschrift*, 6:185–193, 1909.

[7] T.J. Esward, K.M. Lawrence, and P.C. Macey. Distributed computing for sonar transducer analysis. In *Sonar Transducers and Numerical Modelling in Underwater Acoustics*, St Albans, UK, 2005. Institute of Acoustics.

[8] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:94, 1972.

[9] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[10] Raman Khanna, editor. *Integrating Personal Computers in a Distributed Client-Server Enviroment*. Prentice Hall PTR, New Jersey, USA, 1995.

[11] P.C. Macey. Oblique incidence diffraction by axisymmetric structures. *Proceedings of the Institute of Acoustics*, 16(6):67–74, 1994.

[12] Jagdish J. Modi. *Parallel Algorithms and Matrix Computation*. Clarendon Press, Oxford, UK, 1990.

[13] Michael J. Quinn. *Parallel Computing: theory and practice*. McGraw-Hill, New York, USA, 1994.

[14] A.L. Roest, J.J. Kelly, D. Vanmaekelbergh, and E.A.Meulenkamp. Staircase in the electron mobility of a ZnO quantum dot assembly due to shell filling. *Physical Review Letters*, 89(3):036801–1–036801–4, 2002.

[15] B.A. Wichmann and I.D. Hill. Algorithm a183: an efficient and portable pseudo-random number generator. *Applied Statistics*, 31:188–190, 1982.

[16] B.A. Wichmann and I.D. Hill. Correction: algorithm a183: an efficient and portable pseudo-random number generator. *Applied Statistics*, 33:123, 1982.

[17] H. Zeisel. Remark on algorithm a183. *Applied Statistics*, 35:89, 1986.

Appendix A

Glossary of terminology relevant to the NPL grid

A.1 Introduction

This glossary contains brief definitions of some of the most important terms used in describing the detailed operation of the NPL grid. The aim is to help readers understand the main chapters and case studies included in the guide. The definitions chosen often relate specifically to the United Devices MP Grid system that NPL uses, but many of the concepts can be generalised to a wide range of grid and distributed systems.

A.2 Key definitions

Administrator A user who can create, modify, and view the various objects in the distributed computing platform, either through an MP Console or programmatically through the Management Application Program Interface (API), which can be regarded as the software interface to system services or software libraries, so that a programmer can access and control a system. The Administrator is able to dictate the objects that programmers can or cannot modify.

Application An application object is a container for a **program** or a set of programs and the definition of the control and data flow across programs.

Application Service An application service is an executable, script or web page that features a business logic tier which submits pre-processed work to and post-processes work retrieved from the Grid MP platform.

Data The data object describes a partition or segment of a **data set**, and is uniquely identified within a data set.

Data Set A data set object defines a set of **data** files. A data set can be used globally, or it can be associated with a specific **job** or **job step**.

Device A device is a computer running the MP agent software. A device must belong to a **device group** and be assigned a **device profile** in order to process jobs. An alternative term *host* is also used to refer to a device.

Device Group Device groups enable **administrators** to group and manage multiple devices at once. Device administrators can specify which applications and/or programs can run on a device group. Applications and programs may be excluded from a device group either due to administrative policies or due to resource limitations in the device group.

Device Profile Device profiles are configured by **administrators** and applied to **devices** to control the maximum disk space Grid MP platform work units can utilise, the times of day a device is available for work and polling, and the **programs** and **applications** it can run. Administrators can set up multiple profiles per **device group**, but only one profile can apply to each device.

Grid MP Services A collection of programs that run on the main Grid MP Server(s). The Grid MP Services are: Realm Service, Dispatch Service, File Service, Poll Service and RPC Service. The database also runs on the Grid MP Server. See section A.3 for definitions of these services.

GUI Graphical User Interface.

Host See *Device*.

Java Java is a general purpose programming language with a number of features that make the language well suited for use on the World Wide Web. Small Java applications are called Java applets and can be downloaded from a Web server and run on a computer by a Java-compatible Web browser, such as Netscape Navigator or Microsoft Internet Explorer.

Job A job is an instantiation of an **application**. It contains **job steps**, which refer to **program** attributes. Job steps contain all of the metadata the job step needs to run on a **device**, such as resource requirements, and links to the executable file and the data it will process.

Job Step A job step is the instance of the execution of a **program**. A job step belongs to a single **job**.

MGSI The Management Grid Services Interface is a programmatic interface that provides third-party applications and the MP Console access to the **Grid MP Services** and the database. The MGSI also incorporates access control for users of these services.

MP Agent The MP agent is a lightweight program that runs on a **device** and manages job processing. The MP agent is responsible for processing work and for automatically returning result files to the Grid MP platform.

Mpbatch This is a batch submission application, and is a method for submitting both code and data in a single unit of work. It is intended for use with computational requirements that may change frequently. All

the programs for the application and its input data are packaged together and submitted to the Grid MP platform as a job via the "mpsub" command line utility.

MPI Message Passing Interface is a widely used standard for writing message-passing programs. United Devices uses the MPICH standard, which is a freely-available, portable version of the MPI standard.

Perl Perl stands for Practical Extraction and Report Language and is a programming language especially designed for processing text. Because of its strong text processing abilities, Perl has become one of the most popular languages for writing Common Gateway Interface scripts. Perl is an interpretive language, which makes it easy to build and test simple programs.

PHP "PHP: Hypertext Preprocessor" is a widely-used Open Source general-purpose scripting language that is especially suited for Web development and can be embedded into HTML. The main goal of the language is to allow web developers to write dynamically generated web pages quickly.

Program A program is a set of executable code components that perform work in the system. Programs contain the following components: Program Version, **Program Module**, Program Module Version.

Program Module The Grid MP system is designed to contain **devices** that run on different platforms (hardware and operating system configurations). A program module is a collection of one or more executable programs compiled to run on a specific platform.

Sandbox A protective mechanism used in some programming environments that limits the actions that programs can take. A program normally has all the same privileges as the user who runs it. However, a sandbox restricts a program to a set of privileges and commands that make it difficult or impossible for the program to cause any damage to the user's data.

Workunit The Workunit describes the smallest piece of work in the Grid MP platform that it can schedule. Workunits belong to a single **job step**, and generate results upon successful completion.

XML eXtensible Markup Language (XML) is a method of data exchange. It allows the user to create meta-language (tags) to represent data and meta-data (data about data). XML describes the data and its structure. There are many tools available to create XML documents and to load, read, validate and parse those documents. This allows applications running on totally different platforms to exchange data. For more information see the web pages at `www.w3.org/XML/`.

A.3 Administration software definitions

Dispatch Service The dispatch service schedules workunits to devices based on device capabilities and availability, and receives the result status from

those devices. Devices that are idle in the Grid MP platform connect to the dispatch service to receive new workunits. The dispatch service selects and sends a workunit to each connecting device. It includes a workload scheduler that schedules workunits to devices based on job and device preferences and attributes. When a workunit is dispatched, all metadata about it is also sent. The files containing actual data are uploaded or downloaded through the **File Service**.

File service This is a secure (Secure Sockets Layer encrypted) file transmission service for downloading and uploading data to and from MP agents during Job execution. The file service relies on an underlying file system. MP agents can also use the Grid MP platform URL mechanism to download data from a list of specified URLs.

MP Database The MP database is the main storage repository of common information for the Grid MP platform. The MP database stores system state and metadata; the **File Service** server stores job and application data.

Poll Service The MP poll service collects periodic status reports from MP agents and communicates commands to the MP agent from other services.

Program Loader This is a mandatory loader that loads executable code into memory for execution, and provides automatic encryption and compression for executables.

Realm Service The realm service manages all MP agent authentication and MP agent resource access. MP agents communicate with the realm service to register and authenticate themselves. An MP agent cannot perform any operation until it has authenticated itself with the realm service.

Service Manager The primary function of the service manager is to ensure that all required services needed for the proper functioning of the Grid MP platform are always up and running. If any are found to have failed or unexpectedly stopped, the service manager restarts them.

Appendix B

Grid program development, design and creation

Reproduced below is the advice that NPL gives to scientists who wish to develop programs for the NPL grid.

B.1 Can the NPL grid help my application?

A distributed computing system employs the unused capacity on a desktop computer. It monitors the amount of the computer's power that is being used, and uses the spare capacity of the CPU for running calculations at a low priority. If the user suddenly begins to use the computer at full capacity again, the user's applications instantly take over the CPU, and the distributed calculation backs off and temporarily pauses. This means that the user should see no effect from the distributed program – it always uses as much free capacity as the computer has at any given moment, but not more.

On each computer in the system a small program known as an *agent* runs at specified intervals, requesting work from a central computer. If the central computer has a job or part of a job to be run, it packages up all the files required to run the job and sends them out to the agent, which then runs the job in the spare time of the computer and sends the results back to the central computer (which can then, of course, send out another job to the agent). Since the computers in the distributed network only communicate with the central server, and communicate at their own convenience, this allows for a fairly relaxed system. Machines can come and go from the network, connecting to complete work when they are able. If a machine is not on line all the time it can get a job, go off line to process it, then come back on line to send the results to the central computer and request the next job.

Because each individual computer only talks to the central controlling computer, the type of jobs that work best with distributed computing are those in which the work can easily be broken up into discrete packets that do not need to exchange information with each other.

Different types of parallel applications suitable for distributed computing:

- Monte Carlo applications: This type of application is the simplest to convert to distributed computing. Each computer on the DC system runs the same application with almost identical data, but the application relies on a random number for some parameters. Each computer therefore gets a different random number. Will parallelize well on small, medium, or large tasks.
- Non-MC Data Parallel applications: In this type of application, each computer runs the same program, but on a different block of data. For instance, an image analysis application might split a large image into an array of smaller images, each one farmed out to a single computer for analysis. This will only work, obviously, if the individual parts of the image do not rely on other parts of the image already being analysed. Will parallelize well on small, medium, or large tasks.
- Top-Down-Partition applications: This type of application is similar to the previous type, but the individual blocks rely in part on the larger block of information they are part of. Partition sorts are one example, or image analysis where two neighbouring blocks need to know basic (but not detailed) information about the analysed properties of each other. In this case, the controlling software begins by farming out a few general jobs, then takes those results and makes more detailed jobs, then uses those results to create even more detailed jobs, and so on (for instance – do a basic analysis on the entire image). Will only parallelize well on medium to large tasks.
- MPI "Gang": Some distributed computing systems can organize the computers on the system into "gangs" – temporary clusters which can then run clustered software. The United Devices system at NPL can run MPI programs in this mode. However, there are some serious disadvantages – the computers must all be available for the duration of the calculations, so once a gang has been formed none of them can be turned off, and if one begins to run slowly (say the user has started using it heavily), it can slow down the rest of the gang. Where a program cannot be run in any other way, this allows it to get most of the power of the distributed computing system, but if it can easily be rewritten in one of the other forms it will usually see a speed increase.

B.2 The Program Itself

- Grid programs should be Windows console applications (i.e. applications that have no graphical user interface) which can run on Windows 2000 and Windows XP.
- The actual executable name must begin "mp_". For instance, mp_calculation.exe. This allows easier administration of the system, because the grid executables can be easily distinguished at runtime.

Grid programs must **not**:

- listen on a socket for incoming connections;

- make outbound socket connections;
- change system files or the registry;
- access files that include full file paths or `..\` type behaviour that try to get to the parent directory in any form ;
- use C system() or exec() type calls;
- rely on threads and process creation;
- have any graphics;
- allow any type of blocking on events such as user input even if an error occurs, as the task may hang forever or until the agent gets a timeout and kills the task. Therefore, no pop-up error boxes.

B.2.1 Some advice for development

- If source code is available, compile the application with all warning messages enabled and try to get a clean compile without warnings, at least using -Wall on GNU type C/C++ compilers for example.
- If possible, always link libraries statically into the application executable. Do not depend on any libraries or external resources on the agent machine since, very often, you do not/cannot control what is on a user machine. e.g. Matlab and NAG libraries. Consider licence issues.
- Run a bounds checking program where possible to make sure no obvious bugs are present in the code, which might cause a crash on an agent machine. This will also help eliminate memory leaks, which may take up more and more resources on an agent machine as the application runs. If advice is needed on this contact NPL's Mathematics and Scientific Computing Group.
- Running a program such as "lint" on the source code of an application may be a good way to ensure no obvious bugs are located. This is not a required step, but it may be worthwhile to consider for very important applications.

B.2.2 Making the Workunits

Depending on how simple your workunits are, it may be possible to generate them quickly using the NPL Grid utilities. These Perl scripts perform some of the common tasks required for generating simple Grid applications. Two utilities in particular are of note here:

- *GenerateSeeds.pl* – This script will generate a series of seed files for a Monte Carlo application. Given a range of numbers it will create work unit files with seed numbers covering that range.
- *MakeWorkunitsFromTemplate.pl* – This more complicated script will generate a set of workunits that cover a range of variables that you specify, plugging all possibly combinations of those variables into a

template that you provide. If you need to do more complex work during generation of workunits, you will need to create your own script. If workunits can be created as simple files from your script you can use another of the NPL grid utilities, *CreateJob.pl* to send the work units to the agent machines.

B.2.3 Getting and interpreting the Results

The actual results files can be retrieved using two of the NPL grid utilities – *GetJobResults.pl* to retrieve the results, and *ExtractSingleFileFromResults.pl* to parse the results packages returned from the grid and extract the data files. But you will then need to decide on how the results should be grouped. If they need to be joined into a single file, either in a predefined order or randomly, you will need to write a script to do that.

Finally, you will need to think of a strategy for actually interpreting the results when you have retrieved and aggregated them. The large amounts of data that a typical grid run can produce can be difficult to understand without preprocessing, statistical analysis, and/or some form of visualisation tool.

B.3 Using Buildmodule, Buildpkg, and running the Test Agent

B.3.1 Using Buildmodule

To allow your executable to run on the grid, you will need to package it, plus any other files it needs to run (licence files, DLLs, etc.) into what is known as a *module*. This file contains your executable file, support files, and some information about the program's outputs. To make a module, you will need *buildmodule.exe* (*buildmodule* for linux users), which is in the United Devices SDK in the tools/build/ directory, and *loader.exe* (*loader* under linux), in the same directory (*loader.exe* is the wrapper code that is added to your executable to make it work in the grid agents).

Buildmodule at its simplest takes two arguments and one option: the name of the module, the name of your executable, and an option that specifies the results file (the file that will be collected from the agent and returned to you). For instance, if your wanted to add a program to the grid, and your executable was called *mp_calculation.exe* and wrote a results file called *out.dat*, you would use *buildmodule* like this:

buildmodule -oout.dat calculation mp_calculation.exe ... and a file called "calculation" would be created.

A note on naming executables: to help IT support keep track of programs running on the grid, your executables must all be called *mp_[whatever].exe*, which allows system administrators to identify grid programs on a particular computer.

B.3.2 Other options for Buildmodule

Buildmodule allows you to specify other files to be included (for instance,

DLLs required by the main program), and command line arguments for the program call. The options to *buildmodule* must always come before the module name and the arguments to be passed to the program when it is run must always come after the program name:

buildmodule [options for buildmodule, as listed below]
⟨module name⟩ ⟨program name⟩ [command-line arguments for the program]

For a complete list of *buildmodule* options, run *buildmodule* without any arguments. Two common options you may wish to use, however, are the *-f* and *-R* options:

- -f[filename]: include an extra file in the module. This is how DLLs, etc., are included. You may specify this option more than once.
- -R[stream]=[target]: redirect a stream (stdin, stdout, or stderr) to or from the program to the file specified by target. This allows you to capture results that are sent from stdout, or take input from a file as stdin.

B.3.3 Using BuildPkg

When a job is created on the grid, a number of data packets are created, each containing job information for one particular "slice" of the job. Normally you will not be creating these manually (and if you use the simple Perl utilities to create your job, this will do all the work necessary to pack your workunit files into data packages), but you will want to create at least one or two of these during testing in order to run your program against the test agent.

To make a package, you will need buildpkg.exe, which is in the United Devices SDK in the *tools/build/* directory (or *buildpkg* under Linux). *Buildpkg* is fairly simple to use, taking as arguments a package name, then a space-separated list of files to include in the package. One nuance is that you can specify that a given file should have a different name in the package than it does on disk, by specifying *[local name]=[packagename]* instead of just specifying the local name. Supposing you wanted to pack two control files into a package – CONTROL.DAT, and DATA_0005.DAT (which you want your program to see as DATA.DAT). You would call *buildpkg* like this: *buildpkg datamodule_5 CONTROL.DAT DATA_0005.DAT=DATA.DAT ...* and a package called "datapackage_" would be created.

Typically, you will use an automated script to split your work up into slices and package it, but you will need to build one package to use the test agent, and it is good practice to establish what files you will need anyway. *Buildpkg* has a few other options, which you can see by running it with no arguments.

B.3.4 Using the Test Agent

When you create a new version of your executable, you should first test it running locally on your machine, obviously. But after you have assured yourself that it works on the command line, you should also test it locally using the test agent. This program simulates the environment a program running on a remote agent will find itself in as closely as is possible, in order

to ensure that the program is not relying on local disk access outside the agent's sandbox, network access, etc.

Obviously, it is important that when you run the test agent on your machine you ensure that no local DLLs or configuration files are being picked up. If your program stores configuration in the registry it will not work on the agent machines, and if you have not included all the required DLLs in the program module it will not be able to run when distributed – but it might pick up DLLs in the local directory when running and fool you into thinking that it will run correctly. For this reason, we suggest that you create a completely new directory to run the test agent in, and copy just the test agent and your program module into it.

To use the test agent, you will need a module (built with buildmodule), and a valid package of data (built with *buildpkg*), and the test agent program itself, which consists of two files: *testagent.exe* and *udtapi.dll*, which can both be found in the United Devices SDK in the *tools/testagent/* directory. Double-click on *testagent.exe* to run it, and complete the GUI dialog that is presented to you.

B.4 Requesting grid access

Before an application can be put onto the NPL grid, you need to be able to demonstrate that it is unlikely to affect the normal operation of the donor machines, and you need to supply some information about the use of the application.

Download the access request form (see appendix D for details of the content of the form), and fill it in using Word. Save the file, and email it to the System Administrator, along with your program module and the data package you used to test it on the test agent, so that we can double-check that the program isn't relying on any DLLs installed on your computer.

You must also have an Intranet web page in place before the program can be made live. This web page should explain what your program does, what applications the programs results have, etc, so that the people whose PCs you will be using can get an insight into the work their computers are being put to. Links to more detailed pages for people who are interested in finding out more about your work are also encouraged.

When a grid program is running on a "donor" PC, an icon appears in the system notification area (the little icon area near the clock on a Windows machine). Clicking this icon will allow the user to see what job is currently running on their PC and jump to this web page.

Appendix C

Developing software for the NPL distributed computing grid

This appendix lists in detail the stages one needs to go through to be able to set up a new application on the NPL Grid. It was constructed while implementing the Wichmann-Hill pseudo-random number generator on the Grid. It is included here to provide an example of an approach that can be adopted to developing applications.

C.1 Prepare the program

1. Write a program that will run on all processors. In the data-parallel model this will take a different input file on each processor, and write a different output file or files; however, the names of the files will be the same for all processors. Each processor will then perform exactly the same operations on different data.
2. Prepare the .exe program file, and separate input files for all processors. If the program expects a file *filename.txt*, then call the files *filename_<number>.txt*, for example, or name them in some other consistent way that distinguishes them.

C.2 Installing the United Devices Software Developers' Kit (SDK)

This has to be done so that the program can be tested locally. Local testing must be done before trying it on the grid.

1. Install the United Devices SDK, which can be found on *cdserver**volumes*, on your PC.
2. Instructions for using the kit are available at `www.intranet.npl.co.uk/united_devices/buildmodule.html`

3. Command line programs are available for building the module and a data package, and for testing them locally.

C.3 Test the program locally

C.3.1 Build the Program Module

1. In Windows, navigate to the folder containing the .exe file of the program to be run on the grid. Prefix the file with "mp_". It is an ITSU requirement that executables on the Grid are called *mp_something.exe*.
2. Copy into this folder the file *loader.exe*, which is provided in the folder *United Devices SDK\UDsdk_v4.0\tools\build*.
3. Open up a DOS window, and navigate to the same folder
4. In DOS, type *buildmodule [-e] –ofile1.txt –ofile2.txt (&c) progname mp_progname.exe*, where *file1.txt*, *file2.txt* (&c) are the names of the files output by the program. This creates a module directory called *progname*. (The "-e" is optional and asks for no encryption. Without it the output generated by the program may be incorrectly preceded with "^0".)

C.3.2 Build the Data Modules

1. Prepare the required input files for the processors (only one is needed for testing). Copy them/it into the same folder.
2. Type *buildpkg datamodule_n infile_n.txt=infile.txt*, where *datamodule_n* is the name of the data module being made, *infile_n.txt* is the input file for the nth work package, and *infile.txt* is the name of the file the program will open for input. This generates a file called *datamodule_n*.

C.3.3 Test the program using the Test Agent

1. In Windows, double click on *testagent.exe* which is provided in folder *United Devices SDK\UDsdk_v4.0\tools\testagent*.
2. Use the buttons provided to select the program module and a data module (testing only works with a single data module).
3. Type a name for the results – e.g. *result.tar*.
4. Click on *RUN*.
5. The results file will be put in the same folder as *testagent.exe*, unless the button was used to navigate to the program folder.

C.4 Set up an application on the test grid

Ask the system administrator to do this, providing a name for the application, and a name for the program (or programs). The administrator will set up administrator and personal accounts, each with a username and password, on `https://udserver.npl.co.uk`. The administrator account allows programs to be added and removed. The personal account should be used for making grid runs, as it is risky using the administrator one for this.

C.5 Install Windows Perl on the PC

Perl must be installed on the PC that will be submitting the distributed application to the grid. It can be downloaded from
`www.activestate.com/store/languages/register.plex?id=ActivePerl`

Perl scripts are used to create and run jobs on the grid. Install the NPL Grid Perl utilities, obtainable from the system Administrator. Install them in folder *NPL Grid Perl Utils*

C.6 Run the application on the test grid

This is done using the Perl scripts.

1. Edit *NPL Grid Perl Utils/options.ini* – set up the username, password, program name, and workunit (input file) name.
2. Prepare the required input files for all the workunits, and copy them into folder *NPL Grid Perl Utils/workunits*.
3. Open a DOS command window, and navigate to *NPL Grid Perl Utils*.
4. Type *CreateJob.pl* to start the job running. If there are no errors, make a note of the job number at the bottom of the output.
5. Type *GetJobComplete.pl <job number>* to see how complete the job is.
6. When complete, type *GetJobResults.pl <job number>*. This copies a series of <results n>.tar files into the folder *NPL Grid Perl Utils/results*. These contain the files specified in the *buildmodule* stage.
7. Type *ExtractFileFromResults.pl <file name>* to extract all the copies of *filename*. These files will appear in the results directory with the workunit number appended after a dot.

C.7 Run the application on the live grid

Edit the file *options.ini* in the folder *NPL Grid Perl Utils*, and change the option *trainingOnly=on* to *trainingOnly=off*. Follow the instructions given above for running on the test grid.

Appendix D

Form for requesting grid access

Potential users of the NPL grid are asked to complete a standard form that gives basic information about themselves and their application. This appendix lists the questions that the user is required to complete. Readers may wish to consider whether they could adapt aspects of this approach for their own organisations.

D.1 Request For NPL grid access

Please answer the following questions for your application:

D.1.1 Your details

- User [Your name]
- Division [Your division, team, and group]
- Project Number [The project this application is being set up for.]
- Date[Date of request]

D.1.2 Application details

- Application Title [Basic name of the program]
- Description [A description of what this application does and how it does it. This will be used to generate a web page on the Intranet describing this application]

User information and contacts

Give NPL Division, application creator, user and any contacts or references that are useful.

Time scale for project

Are there any deadlines for application development, or data generation?

Supported platforms

On which platforms does this application run, e.g. Windows 9X, NT, 2000, XP, Linux, Unix, AIX, Solaris, etc?

Application licence

Is this application licensed or proprietary? If licensed, please provide details of licence.

Application use

How does the user use the serial version of the application today? How will this software be used in a grid environment?

Application version

What version of this software will be migrated? Is there need to support multiple versions?

Application type

Is this a model development activity or a production run to generate data?

Scientific importance

What are the main scientific benefits of doing this work?

Language and compiler

What language is this application written in? What compiler (and version) was used to compile it, e.g. Fortran, C, Java, Matlab, Labview etc? Is the source code available?

Scientific validation

How can output of this application running on a grid be validated?

D.1.3 Resource requirements

File sizes

How large are the typical input and output files that are used in a job?

File formats

What are the file formats/types for the input and output files?

Assessment of computation/communication ratio

What are the typical run-times of a "job" using this application with conventional computational means? How big is a typical input and output file?

D.1.4 End-to-End issues

Splitting input

How can input data be split into more manageable pieces?

Merging results

How can results be merged back together?

Appendix E

Matlab Distributed Computing Toolbox 2.0

For those organisations that make substantial use of MATLAB, it may be worth considering whether the the MATLAB Distributed Computing Toolbox is able to meet the need for a system that can tackle large parallel or distributed computations. The account below is based on material that MathWorks, the developers and vendors of MATLAB, provided for the authors of this guide.

E.1 Introduction

The Distributed Computing Toolbox and the MATLAB Distributed Computing Engine enable users to develop distributed MATLAB applications and execute them in a cluster of computers without leaving the development environment. Users can prototype applications in MATLAB and employ the Distributed Computing Toolbox functions to define independent or interdependent tasks. Algorithms that require interdependent tasks use the Message Passing Interface (MPI)-based functions provided. The MATLAB Distributed Computing Engine schedules and evaluates tasks on multiple remote MATLAB sessions, reducing execution time compared to running in a single MATLAB session. Together, these products support high-productivity computing by letting users employ the high-level MATLAB language for algorithm development and then accelerate computation via distributed or parallel execution.

E.2 Key features

The main features of the MATLAB system are:

- Distributed execution of MATLAB applications on remote MATLAB sessions;
- Support for communication among interdependent tasks based on the Message Passing Interface (MPI);

- Application scheduling using the MathWorks job manager or third-party schedulers;
- Dynamic licensing to enable distributed execution of algorithms that use any toolbox for which the user is licensed;
- Access to single or multiple clusters by one or more users.

Figure E.1: The interaction between the client machine, where the Distributed Computing Toolbox is used to define jobs and tasks, and the MATLAB Distributed Computing Engine. The scheduler can be the MathWorks job manager, provided as part of the MATLAB Distributed Computing Engine, or a third-party scheduler.

E.3 Working with the Distributed Computing Toolbox and the MATLAB Distributed Computing Engine

The Distributed Computing Toolbox includes functions for defining distributed applications and for submitting jobs and receiving results via a scheduler. The MATLAB Distributed Computing Engine provides a basic scheduler (the MathWorks job manager) and evaluates MATLAB functions on remote MATLAB sessions called workers.

E.3.1 Creating jobs with the Distributed Computing Toolbox

The Distributed Computing Toolbox allows the user to run MATLAB applications (jobs) on a computer cluster by dividing them into tasks (MATLAB functions). The MPI-based functions enable users to pass messages among workers so that they can communicate, as in an application where one process needs the result from another to continue execution.

E.3.2 Dynamic licensing on the Distributed Computing Engine (DCE)

The workers in the Distributed Computing Engine can execute jobs that use any eligible toolboxes for which the user is licensed. They can also execute independent Simulink models (Simulink is a multidomain simulation and design platform for dynamic systems that is integrated with MATLAB.) Each worker machine in the cluster requires only the MATLAB Distributed Computing Engine licence.

E.3.3 Scheduling jobs

The MathWorks job manager in the MATLAB Distributed Computing Engine coordinates the execution of jobs and asynchronously distributes tasks to the workers. The job manager runs jobs in the order in which they are submitted unless the user sets priorities to change the order of execution. It can run on any machine on the network and can accept jobs from multiple Distributed Computing Toolbox users. Using the MathWorks job manager the user can monitor run-time execution through callbacks. Callbacks enable the user to execute a function that is specified on the MATLAB client when an event occurs (such as a worker finishing a task). The MATLAB Distributed Computing Engine also provides a generic scheduler interface that lets the user integrate the Distributed Computing Toolbox and the MATLAB Distributed Computing Engine with third-party schedulers.

E.4 Executing applications

After the user defines and submits the job, the MATLAB Distributed Computing Engine workers evaluate each of the job's tasks and make the results available for retrieval by the Distributed Computing Toolbox.

E.4.1 Distributed execution

With the Distributed Computing Toolbox and the MATLAB Distributed Computing Engine, users can execute applications that do not require communication among workers in a cluster. These are called *coarse-grained* or *embarrassingly parallel* applications. A typical job might be divided into independent tasks that operate on unrelated data sets or sections of very large data sets. This mode of execution supports homogenous or heterogeneous clusters.

E.4.2 Parallel execution

The Distributed Computing Toolbox and the MATLAB Distributed Computing Engine enable the user to perform parallel execution of applications that require communication between workers during processing. These are called *fine-grained* applications. Parallel jobs have only one task that runs simultaneously on every worker. The function that the task runs can take advantage of a worker's awareness of how many workers are running the

job, which worker this is among them, and the features that allow workers to communicate with each other.

E.4.3 Working in managed interactive or batch mode

With the Distributed Computing Toolbox and the MATLAB Distributed Computing Engine the user can work in managed interactive mode (with the MathWorks job manager) or batch mode (a third-party scheduler).

Managed interactive mode allows use of the Distributed Computing Toolbox interface to develop distributed applications interactively and access distributed computing resources through the scheduler at development time. This mode is used for prototyping and developing algorithms to speed execution, and view results in real time. Batch mode lets the user send applications to a batch queue and receive resources for execution only when the scheduler allocates them. This mode can be used for large data manipulation tasks, to take advantage of available resources and free up the MATLAB client session for other activities while the distributed job runs.

E.5 Web links

More information about the MATLAB Distributed Computing Toolbox can be found at: `www.mathworks.co.uk/products/distribtb/index.html` and `www.mathworks.co.uk/`.

Appendix F

Use of DOE/RSM techniques

The information in this appendix was provided by Bookham Technology Ltd and is based on their experience of tackling engineering design problems with distributed computing methods.

F.1 Introduction

The combination of Design Of Experiments (DOE) and Response Surface Modelling (RSM) techniques provide a powerful means of exploring the effects of varying the inputs to a process (the control factors) on the process outputs (responses). "Design of Experiments" is a technical term that refers to a structured, organized method for determining the relationship between factors affecting a process and the output of that process. The aim of "Design of Experiments" methods is to produce an experimental strategy that explores efficiently the effects of varying inputs to a process. "Response Surface Modelling" is used to examine the relationship between one or more response variables and a set of quantitative experimental variables or factors. These methods are often employed after the user has identified a limited number of controllable factors and wants to find the factor settings that optimize the response.

The combined DOE/RSM technique involves the construction and use of analytical models that approximate the dependencies of the responses on control factors. DOE/RSM techniques are well suited to a distributed computing environment.

Some advantages of DOE/RSM are that it is:

- Inexpensive to evaluate;
- Defined for all allowed values of controls;
- Continuous in derivatives.

Typically RSM models are low order polynomials, usually up to the second order. Once constructed the models can be exported into other

applications (e.g. spreadsheets, visualisation software, etc) which allow the end user to explore the design space and investigate optimisation of the factor settings in order to achieve the desired process responses. Process sensitivity and capability analysis may be investigated by applying the expected factor distributions to the RSM models and analysing the consequent response predictions.

DOE/RSM techniques are only applicable where both the factors and the responses are continuous within the design space. Care must be taken in setting up an experimental design to ensure that the minimum and maximum values used for each of the factors do not result in response values that fall outside of the region that can reasonably be approximated by a polynomial fit. It is important therefore both to estimate the accuracy of the RSM model fits by using standard statistical methods and to perform simulations using the predicted optimal factor settings in order to verify the accuracy of the models. Often it will be necessary to redefine the simulated design space after the evaluation of the initial models.

F.2 Choice of design

When setting up the DOE the choice of the design space needs to be considered carefully. A practical method is to use a Latin Hypercube type of design.(For a description of Latin Hypercube methods see `http://en.wikipedia.org/wiki/Latin_hypercube`.) This approach has a number of advantages.

- The number of experiments can be determined by the user (although clearly thought must be given to the required model accuracy).
- RSM models (albeit of limited accuracy) can be constructed before all the simulations have been completed. This may be advantageous in giving an early indication of whether the chosen experimental range is appropriate.
- The simulations can be run in blocks, with each block based on the same factor ranges. As each block of simulations is completed the results may be combined with those from earlier simulations. This approach allows early access to models whose accuracy improves as more simulations are completed.
- This approach is very tolerant of missing (or anomalous) data points, which can be excluded from the analysis.
- Generally the Latin Hypercube design approach is more efficient than a purely random experimental design.

F.3 Running the experimental design and building response surface models

There are seven clearly defined stages to the DOE/RSM process.

1. Generating the experimental design.
    - The details of the experiment will be determined by the design type, the number of factors, their ranges and the number of experiments (which will either be determined by the design type or by the user).
2. Generating the simulation input files.
    - Once the experiment has been designed the factor settings for each experiment must be translated into input files for the simulator. One approach to this is to set up a template of the input file which contains clearly defined text strings instead of the factor settings. At run time the text strings are substituted by the actual factor settings for that experiment.
3. Running the simulations
    - For efficiency in completing the experiment in the shortest possible time the simulations will be distributed between a number of computers. This requires software to manage the allocation of the simulations to machines based on their load.
4. Extracting results
    - At the end of each simulation the experimental responses are extracted. Typically these are the key parameters which are required to be optimised for a particular process. One approach is to write the extracted parameter names and values to an output file from which the parameter values can be extracted and associated back to the original factor settings.
5. Building the RSM models
    - Once the responses have been collected and associated with the original factor settings it is possible to build RSM models. It is at this stage that the model accuracy can be checked and missing or anomalous data points excluded. It is also at this stage that it may become apparent that the original design may need to be redefined and repeated (possibly because the original factor settings or ranges were inappropriate).
6. RSM model visualisation
    - Once satisfactory models have been built then the models may be transferred to other applications (e.g. spreadsheets) which may improve the visualisation of the design space. This approach also allows the models to be presented to an end user in a user-friendly format.
7. RSM model verification.
    - At the end of the process it is important to verify the RSM model predictions by running discrete simulations using any specific factor settings which have been identified as optimal.