5-2010

Enhancing reliability with Latin Square redundancy on desktop grids.

Nathan Patrick Johnson
University of Louisville

ENHANCING RELIABILITY WITH LATIN SQUARE REDUNDANCY ON DESKTOP GRIDS

By
Nathan Patrick Johnson
B.A., Western Kentucky University
M.S., University of Louisville

A Dissertation
Submitted to the Faculty of the Graduate School
of the University of Louisville
In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy

Computer Engineering and Computer Science Department
Speed School of Engineering
University of Louisville
Louisville, Kentucky

May 2010

ENHANCING RELIABILITY WITH LATIN SQUARE REDUNDANCY ON DESKTOP GRIDS

By

Nathan Patrick Johnson
M.S., University of Louisville
B.A., Western Kentucky University

A Dissertation Approved on

April 15, 2010

By the following Dissertation Committee:

Dissertation Director
Dr. James H. Graham

Dr. Dar-Jen Chang

Dr. Gail W. Depuy

Dr. Adel S. Elmaghraby

Dr. Rammohan K. Ragade

DEDICATION

This dissertation is dedicated
To my wife Shelley Catharine Johnson and our son Patrick
Both of whom gave me wonderful support

And

To my parents Mr. Donald R. Johnson and Mrs. Geneva Johnson
who gave me confidence

And

To my grandparents

ACKNOWLEDGMENT

I would like to thank Dr. James H. Graham for his guidance and patience and Dr. Rammohan Ragade for his continued encouragement during my university career. Committee members Dr. Dar-Jen Chang, Dr. Gail W. Depuy and Dr. Dr. Adel S. Elmaghraby have given me help and support for many years. Dr. Jeff Hieb has been a good friend and provided much helpful advice. I would like to acknowledge the flexibility Mr. Ed Birchler has provided me while I worked on this dissertation.

I especially would like to thank my wife Shelley for her unfaltering support, kind encouragement and for having more faith in me at times than I had in myself.

ABSTRACT

ENHANCING RELIABILITY WITH LATIN SQUARE REDUNDANCY ON DESKTOP GRIDS

Nathan Patrick Johnson
April 15, 2010

Computational grids are some of the largest computer systems in existence today. Unfortunately they are also, in many cases, the least reliable. This research examines the use of redundancy with permutation as a method of improving reliability in computational grid applications. Three primary avenues are explored – development of a new redundancy model, the Replication and Permutation Paradigm (RPP) for computational grids, development of grid simulation software for testing RPP against other redundancy methods and, finally, running a program on a live grid using RPP. An important part of RPP involves distributing data and tasks across the grid in Latin Square fashion. Two theorems and subsequent proofs regarding Latin Squares are developed. The theorems describe the changing position of symbols between the rows of a standard Latin Square. When a symbol is missing because a column is removed the theorems provide a basis for determining the next row and column where the missing symbol can be found. Interesting in their own right, the theorems have implications for redundancy.

In terms of the redundancy model, the theorems allow one to state the maximum makespan in the face of missing computational hosts when using Latin Square redundancy. The simulator software was developed and used to compare different data and task distribution schemes on a simulated grid. The software clearly showed the advantage of running RPP, which resulted in faster completion times in the face of computational host failures. The Latin Square method also fails gracefully in that jobs complete with massive node failure while increasing makespan. Finally an Inductive Logic Program (ILP) for pharmacophore search was executed, using a Latin Square redundancy methodology, on a Condor grid in the Dahlem Lab at the University of Louisville Speed School of Engineering. All jobs completed, even in the face of large numbers of randomly generated computational host failures.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Malaria, climate forecasts, particle simulation, astronomical star search, DNA and protein analysis, cryptography, the search for alien life; these are just a few of the problems under attack by the largest and most powerful computer in the world. The computer is not an incredibly expensive machine sequestered at some large institution. Part of it in fact might be on the desk in front of you because the largest computer system in the world is a volunteer desktop grid.

In one twenty-four hour period on February 18, 2010, the average throughput for BOINC, the Berkeley Open Infrastructure for Network Computing grid middleware system, was 4,326.99 Teraflops. The fastest traditional supercomputer in the world, according to the Nov. 17, 2009 release of the Top500 list, was the Cray XT5-HE Jaguar at Oak Ridge National Laboratory with a theoretical peak of 2.3 petaflops or 2,300 Teraflops.

As computational power in the form of desktop computers has become ubiquitous and less expensive, the unused cycles of such systems have become an available resource for serious computing efforts. Why then aren't most computationally intensive jobs sent to such computational grids? In particular, the problem of reliability limits the usefulness of desktop and volunteer systems. While recent research has focused on improving grid middleware schedulers and algorithms, and more recently on cloud computing, much remains to be done

This research explores a method of building reliability into grid applications -- generally described as RPP, the Replication and Permutation Paradigm -- by changing the way that data and tasks are arranged and distributed to the various hosts that make up the grid. Specific objectives from this research include:

1. The concepts of reverse mirroring and a Latin Square arrangement of data/tasks is explored in a set-theoretic model.
2. Grid simulation software is constructed and used to evaluate reliability of the model versus other types of replication or over-provisioning in the face of randomly generated host failures.
3. The upper bound for job length in the face of a known number of host failures using the Latin Square data and task distribution is shown by mathematical induction.
4. Finally the practicality of running actual grid jobs with a Latin Square configuration is shown in a case study by reproducing a previous job on an actual Condor grid where host errors are introduced.

The research shows that reliability and even efficiency can be greatly improved using the methods outlined here. Job length may be predicted and a grid job will complete even when all of a job's computational hosts but one have failed

1.1 Overview of Grid Computing

Desktop and volunteer grids are a subset of grid systems and are generally considered to be "computational grids" where the main purpose is to distribute computationally intensive tasks. This research is most concerned with such computational grids. In general, however, grids are a wide-ranging subset of distributed computing and provide "sharing, selection, and aggregation" of a variety of resources in a "seamless, integrated computational and collaborative environment ... that performs resource discovery, scheduling, and the processing of application jobs." [1] This is shown in Figure 1.1.

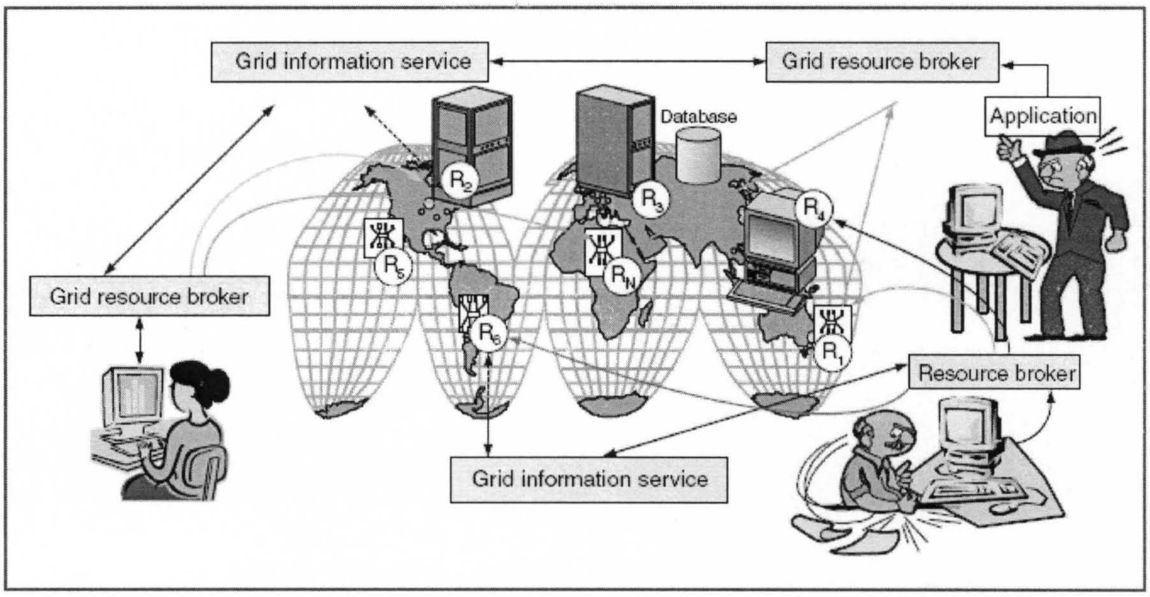


Figure 1.1: A high-level view of the Grid showing users interacting with the Grid resource broker which then discovers resources, handles scheduling and processes jobs (adapted from [1]).

Ian Foster and others write in [2] that:

> "The real and specific problem that underlies the Grid concept is *coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations*. The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource brokering strategies emerging in industry, science, and engineering."

His definition bears similarities to a new computing paradigm, which is also based on the idea that computation should be provided as a utility – cloud computing.

Grids differ from computational clusters, such as Beowulf clusters [3], in that they are not tightly coupled with dedicated internal networks, generally have heterogeneous hardware and are not centrally managed. Grids differ also from clouds as in "cloud computing," although the relationship is less clear. In general a cloud computing model involves a set of services offered on a network for a fee, which frees local enterprises from the cost of maintaining hardware and other infrastructure. The idea of "transparent access to resources on a pay-per-use basis" [4] is one that has been proposed for grid systems as well. [5] Generally, however, one thinks of a cloud as providing services on a virtualized machine where hardware can by dynamically configured to variable loads. Another central idea of cloud computing is integration into the user's computer and routine so that cloud services are innocuous and easily accessed.

Although these may be goals for grid computing as well, these ideas are not central to the paradigm of a "computational grid" where aggregation of computational resources for large jobs is of central interest. There are many types of grids, and taxonomy is discussed in the next chapter.

As mentioned previously the largest and least expensive of grid systems are also the least reliable in terms of hardware and resources. Such desktop and volunteer grids offer great potential for helping to solve some of the most intractable computational problems. The difficulty presented by lack of reliability was borne out in previous research at the University of Louisville where machine failures extended the time and effort needed to retrieve results from an Apple Xgrid of machines spread across the Commonwealth of Kentucky.[6]

1.2 Organization of Dissertation

Discussion of the research continues in Chapter 2 with an investigation of related work in grid computing. Chapter 3 presents a basic model for data and task replication across desktop grids. Chapter 4 extends the model with a proof by mathematical induction of the maximum makespan (time required to finish all tasks in an overall job) given the number of host failures. Chapter 5 discusses comparison of various methods of using redundancy or over-provisioning for reliability in a software simulation of a grid system as well as discussion of development of the grid software. Chapter 6 describes a case study using a Latin Square data and task distribution methodology to conduct pharmacophore search on a Condor [7] grid at the University of Louisville Speed School of Engineering. Finally, Chapter 7 presents conclusions and directions for future research.

CHAPTER 2

SURVEY OF RELATED LITERATURE

Because the motivation of this research involves producing reliable application software for desktop grids, previous and current work involving desktop and volunteer systems will be examined, followed by a survey of the literature with regard to redundancy and checkpointing as a method of aiding reliability. More specifically what follows here is: a discussion of the definition of grid computing in Section 2.1, a discussion of a very broad taxonomy of grid systems in Section 2.2, a brief history of the development of grid systems in Section 2.3, a discussion of desktop and volunteer grid systems in Section 2.4, a discussion of unreliability in desktop and volunteer grids in Section 2.5 and some ways researchers have moved toward greater desktop grid reliability in Section 2.6. The final Section, 2.7, describes research involving the central ideas of replication and permutation.

2.1 What is a Grid?

The term "Grid Computing" was first used in a seminal paper "The Grid: Blueprint for a new computing infrastructure." [8] The idea was that a computational grid would make computing power as available on the computational grid as electric power is on the power grid. Ian Foster, who's becoming recognized as the "father" [9] of grid computing, Carl Kesselman and Steven Tuecke , all of Argonne National Labs at the

University of Chicago, have done much of the seminal work in grid computing as well as producing the popular Globus Toolkit middleware. [10] Foster and others also have attempted to define grid computing, to characterize the need for it and to provide a framework to think about the concept of grid computing.

In 2002 Foster pointed out the need for a clear definition [11]:

> "Grids have moved from the obscurely academic to the highly popular. We read about Compute Grids, Data Grids, Science Grids, Access Grids, Knowledge Grids, Bio Grids, Sensor Grids, Cluster Grids, Campus Grids, Tera Grids, and Commodity Grids. The skeptic can be forgiven for wondering if there is more to the Grid than, as one wag put it, a "funding concept"—and, as industry becomes involved, a marketing slogan. If by deploying a scheduler on my local area network I create a "Cluster Grid," then doesn't my Network File System deployment over that same network provide me with a "Storage Grid?" Indeed, isn't my workstation, coupling as it does processor, memory, disk, and network card, a "PC Grid?" Is there any computer system that isn't a Grid?

Foster defines a Grid in [11] as a system that:

- Coordinates resources that are not subject to centralized control... For example a grid user might use two computers that have different system administrators and that are owned by different entities.
- Uses standard, open, general-purpose protocols and interfaces... Such a standard would provide solutions to developers for authentication, authorization, resource discovery and access.
- Delivers nontrivial qualities of service including throughput, availability, security and resource allocation so that the system is of greater value than simply the use of its parts.

He points out some systems that do not qualify as grids include Sun's "Sun Grid Engine" and Veridian's "Portable Batch System." Indeed it has become fashionable to refer to perfectly good computational cluster computers, particularly if they are not stowed in a single rack, as "Grids." Of course there are other definitions of grid computing; it should be pointed out that they tend to share the concept that the systems are operated by different administrative organizations.

In "Grid Characteristics and Uses: A Grid Definition," Bote-Lorenzo and colleagues gather numerous academic sources in their investigation and support for their definition of a grid. In [12] they define a grid as "a large-scale graphically distributed hardware and software infra-structure composed of heterogeneous networked resources owned and shared by multiple administrative organizations which are coordinated to provide transparent, dependable, pervasive and consistent computing support to a wide range of applications. These applications can perform either distributed computing, high throughput computing, on-demand computing, data-intensive computing, collaborative computing or multimedia computing."

In "What is a Grid?" [13] Grimshaw says:

> "From a hardware perspective a Grid is a collection of distributed resources connected by a network, possibly at different sites and in different organizations. Those resources may include terascale supercomputers, instruments such as telescopes and microscopes, computer-controlled factory floor tools, mid-level servers, desktop machines, laptops, PDAs, and even someday devices such as video cameras, cell phones, and kitchen appliances.
>
> "What distinguishes these resources is that they have a network interface and some software that grid-enables the device. Thus, one could say that from a hardware perspective potential Grid resources range from toasters to teraflops. One could argue that the above definition of Grid is what we used to call a distributed system. I do not dispute that it is what we used to call a distributed system. To me Grids are the evolution of distributed systems to a wide area, multi-organizational context."

He goes on to say that the objective of Grid middleware is to virtualize resources, provide access and, in general, deal with the physical characteristics of the Grid. Grid middleware should allow users and applications to access Grid resources in a transparent manner. "The first and most important aspect of the problem is how do you name and access these resources? This has been a problem in distributed systems for over two decades. The solution is to develop an integrated, global naming scheme where all resources, applications, hosts (CPU's), storage, files, people, security policies, etc., are all named in a consistent manner." Naming is one of the cornerstones of OGSI [3] the Grid standard being developed in the Global Grid Forum.

In the 2007 paper, "Defining the grid: a snapshot on the current view," Stockinger, discusses the results of a survey of more than 40 grid researchers around the world [14]:

> "We can consider the grid as the combination of *distributed, high-throughput* and *collaborative systems* for the effective *sharing* and *distributed coordination* of *resources* which belong to *different control domains*" [Maria S. Perez, Technical University of Madrid]. "Generally, a Grid provides a "distributed computing power infrastructure. It is supposed to provide researchers (users) with a *single entry point* to launch jobs" [Laurent Falquet, Swiss Institute of Bioinformatics]. "Simply put, Grid means "distributed computing across multiple administrative domains" [Dave Snelling, Fujitsu UK]. "Sometimes the Grid is also called to be the software environment [Geoffrey Fox, Indiana University] that *integrates*, *virtualizes*, and *manages* distributed *resources* (software and hardware)." Another view is that a Grid is "a *very large scale resource management system*" [Andrea Domenici, University of Pisa].

According to the Global Grid Forum's Open Grid Services Architecture glossary, a grid is "A system that is concerned with the integration, virtualization, and management of services and resources in a distributed, heterogeneous environment that supports collections of users and resources (virtual organizations) across traditional administrative and organizational domains (real organizations)."

CoreGRID [15] is The European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies. It is operated as a European Research Laboratory (known as the CoreGRID Research Laboratory) and defines a grid as: "A fully distributed, dynamically reconfigurable, scalable and autonomous infrastructure to provide location independent, pervasive, reliable, secure and efficient access to a coordinated set of services encapsulating and virtualizing resources (computing power, storage, instruments, data, etc.) in order to generate knowledge."

In his seminal 2002 paper "The Grid: A new infrastructure for 21st century science" [16], Foster points to some of the services this new sort of computational infrastructure makes available including:

- Science portals where web based clients or other methods provide simple ways of running remote software packages
- Distributed computing where numerous computers are "harnessed" together to provide computational power for large problems
- Large-scale data analysis
- Analysis of the output of various instruments where large numbers of computers are needed to sift though the output of telescopes and other scientific apparatus
- Collaborative work as in the Access Grid project, an open source conferencing system developed at Argonne National Labs as well as other places, that allows scientists to discuss and visualize their work

Another way to look at the grid is in terms of protocols. In [2] the grid is described as a layered set of protocols similar to the manner in which the more familiar Internet Protocol Architecture is often described.

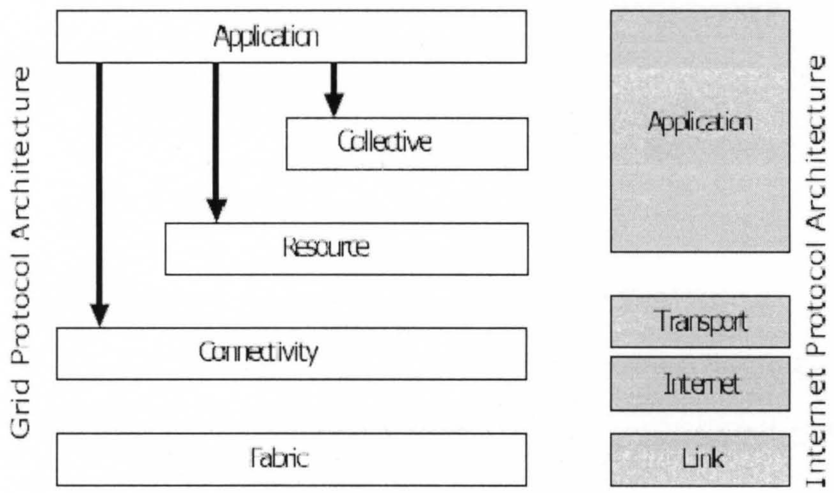


Figure 2.1: The grid architecture and the relationship to the Internet protocol architecture (adapted from [2]).

The grid architecture is described as follows:

- The Fabric layer defines a range of local resource types such as "computational resources, storage systems, catalogs, network resources, and sensors." A "resource" may be a logical entity, such as a distributed file system, computer cluster, or distributed computer pool … "Local resources should provide mechanisms that allow discovery of their state and capabilities and resource management mechanisms."
- The Connectivity layer defines communication protocols generally drawn from the TCP/IP stack. In terms of authentication this layer should provide single sign on, delegation of user rights to programs, integration with local resource security and user-based trust relationships so that users can move from one resource provider to another without security interaction between the individual systems.
- The Resource layer provides a small number of protocols can be used to attain access to the underlying local resources. Information protocols can be implemented to get information about resource configuration, load and cost etc.

Management protocols negotiate access to a resource including requirements such as advanced reservation, operations like process creation, operation status and termination.

- The Collective protocol layer contains protocols and services, application programming interfaces and development kits that call protocols in the resource and connectivity layers. Examples cited by Foster include directory services of resources by name or, for example, by load; scheduling and brokering services for placing tasks on resources; monitoring and diagnostic services; data replication services to place data for best performance and reliability; grid enabled programming systems that allow access to various grid services; workload management and collaboration systems; software discovery services; accounting and payment services; and collaboration services such as the access grid, a collaborative audio and video enabled meeting environment.
- Finally the application layer may use many other languages and frameworks in addition to calls to the various grid services and resources.

In [17] Asadzadeh, Buyya and others examine four global grid systems and software toolkits. They organize the grid middleware into a four layered stack.

The authors define the layers in [17;18] as follows:

- The Grid Fabric layer includes distributed resources such as supercomputers or PCs running various operating systems, networks, storage devices and scientific instruments such as telescopes or sensor networks.
- Core Grid middleware provides a consistent method of accessing distributed resources in the fabric layer by providing services including remote process management, co-allocation of resources, storage access, information registration and discovery, security, and aspects of Quality of Service like resource reservation and trading.
- User-level Grid middleware utilizes the services provided by the lower-level middleware to provide higher level services including application development environments, programming tools and resource brokers for managing resources and scheduling application tasks for execution on global resources.

- Grid applications and portals are typically developed using various languages and utilities. A bioinformatics problem for example would require computational power and access to remote data sets. Other types of programs may need to interact with scientific instruments. Grid portals on the web offer interfaces to job submission services and methods to collect the results.

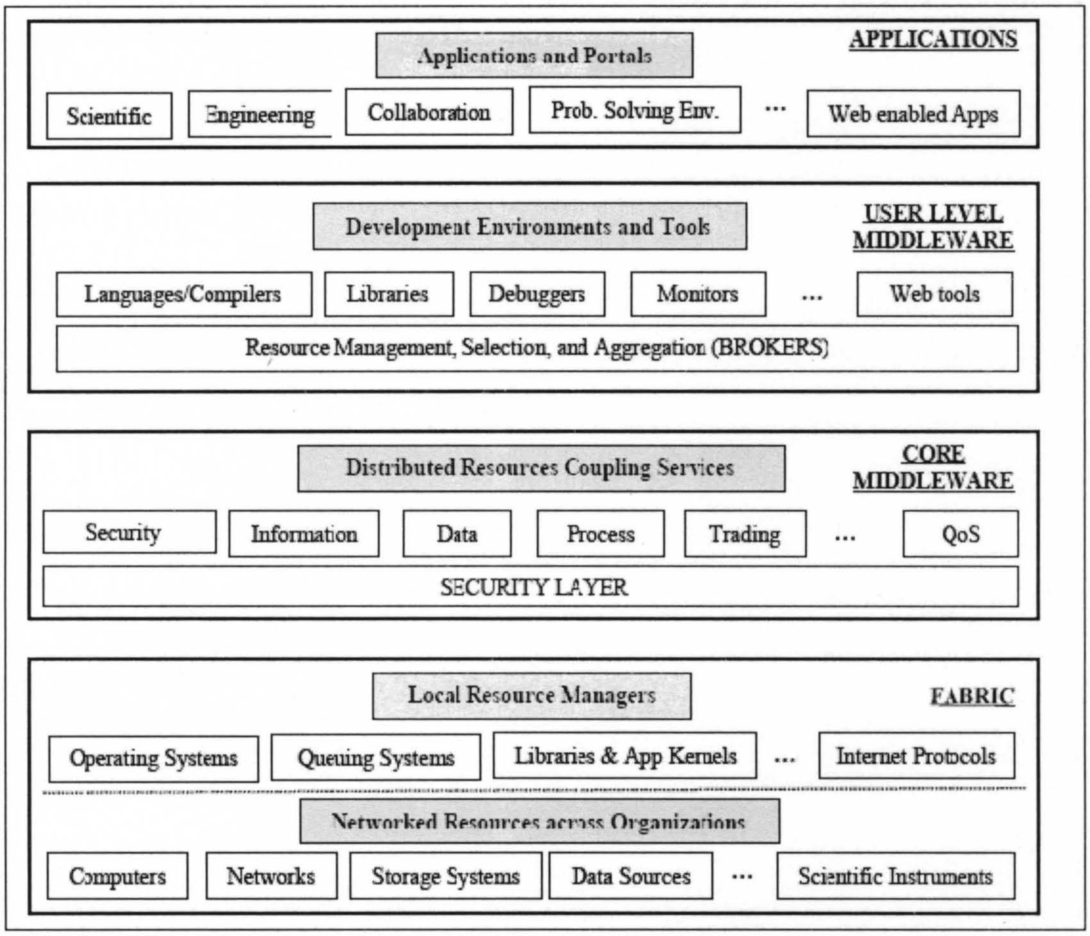


Figure 2.2: A layered grid architecture and components (adapted from [17]) .

In addition the authors provide a hierarchical list of grid projects according to the layer of services provided by the project. A few selected examples from their listed projects – along with some current updates -- include:

Integrated Grid systems:

- BOINC, Berkley – Provides tools for creating and managing volunteer grid projects.
- Javelin, UCSB -- A Java-based system.
- XtremWeb, Paris-Sud University – A global computing (cycle-stealing) environment.
- Unicore, Germany – A java environment for accessing HPC resources.
- World Community Grid – Currently migrating projects to BOINC platform.

Core Middleware:

- Cosm, Mithral -- A P2P toolkit.
- Globus, Globus Alliance of Argonne National Labs and others – A secure set of tools for accessing distributed resources.
- Gridbus, University of Melbourne – A project of the active GRIDS lab intended to merge grid technology with business needs; the lab also offers a grid simulator, Gridsim.
- Legion, University of Virginia – An object oriented system.

User-level Middleware:

- Condor-G, University of Wisconsin
- Nimrod-G, Monash University

Major grid application efforts include TeraGrid, European DataGrid, CERN and many national grid projects such as D-Grid in Germany, GARUDA in India, National Grid Service in the UK, the China Grid Project and many others. Communities of grid researchers and developers also have evolved an effort to produce standards. The Open Grid Forum (OGF) [19] was formed from the merger of the Global Grid Forum (GGF) and the Enterprise Grid Alliance (EGA). OGF is responsible for the OGSA, OGSI, and JSDL standards among others. The GGF had a rich history and established international presence within the academic and research communities along with a growing participation from industry. EGA was a consortium focused on developing and promoting enterprise grid solutions. The GGF grew out of SC98, the annual supercomputing conference. The first such group, the Grid Forum, merged in 2000 the European Grid Forum (eGrid) and the Asia-Pacific Grid Forum to form the Global Grid Forum.

Enabling Grids for E-scienceE (EGEE) [20] connects some 70 institutions in 27 countries across Europe to create a reliable, robust grid infrastructure, middleware and "to attract, engage and support a wide range of users from science and industry, and provide them with extensive technical and training support." Grid 5000 is a French grid project which provides a base system for experiments into grid scheduling and reliability among other issues. [21]

2.2 A Partial Taxonomy of Grid Systems

Any taxonomy presented here is not an attempt to place a firm template across the rapidly changing field of grid computing where even the definition of the term "grid" differs according to purpose and viewpoint of the writer, but rather an attempt to find some frame of reference for interpreting the literature and narrowing the field of interest to something manageable. Grids might be broadly categorized according to two modes of analysis, either some metrics that define how the grid is constructed such as size, geographic separation and "connectedness of nodes," or by some qualitative analysis of functionality such as computation or data service.

Taxonomies of grid systems exist in terms of particular properties -- including taxonomies of workflow management systems [22] and taxonomies of resource management systems for grids, and some of these propose general taxonomies for grid systems in more general terms as well. Krauter, Buyya and Maheswaran classify grid systems [23] according to functionality because design of resource management systems is to some degree a function of the use to which the system will be placed. Their taxonomy includes:

- Computational Grids – Provide more computational power in aggregate than is available on single systems.
- Distributed Supercomputing – Computational tasks are executed in parallel on multiple machines.
- High Throughput – Streams of jobs are sent to nodes on the grid to complete the pool of jobs as quickly as possible.
- Data Grid – Provides services relating to storage management and data access over a wide area.
- Service Grid – Groups and provides services from a number of machines.
- On Demand – Pulls together a variety of resources to provide new services.
- Collaborative – Users and applications are connected in workgroups.
- Multimedia – Real-time media services are provided across machines in the grid.

While this is a solid taxonomy and summary of the current situation in grid development so far as functionality is concerned, it doesn't address the issue of the capabilities of various types of grids and the challenges facing grid researchers in these areas.

Another way to approach taxonomy is to seize upon Foster's definition of a grid and consider the number and qualities of the administrative domains that comprise the particular grid in question. The administrative aspect is that which differentiates a grid from a cluster and to some degree determines other factors including the grid middleware. The term grid middleware is defined in [17] as the software layer that resides on top of the heterogeneous set of operating system resources in the grid providing uniform functionality and services to grid applications and distributed systems. It is arguable that the set of services provided is to some extent dependent on the number of administrative domains in the grid, which in turn affects the reliability of the computational resources upon which the grid middleware and other software is based. Indeed this is a logical way to examine the issue and tends to crop up in the literature. In their 2002 paper, Baker, Buyya and Laforenza [1] categorize existing grid projects in a hierarchical manner composed of "... integrated Grid systems, core middleware, user-level middleware, and applications/application driven efforts. Selected ones are further grouped into country/continents wise..." They make no claim of taxonomy although their classification system provides a nice framework for discussion of the various capabilities and types of grids and is indicative of the way researchers and others classify grid systems in their dialogue.

In [24] the writers suggest that "Grids can be classified in two ways, according to their architecture and coverage. Considering their coverage we can define two main categories: global grids and enterprise grids." In terms of architecture they point out that global grids require more security, have more heterogeneous resources, among other things, and because enterprise grids, although they might comprise different administrators in a worldwide setting consisting of different departments, they generally are owned by a single overall organization. Once more their use of this sort of thinking in their paper is indicative of the way many think about grid computing. It might be possible to classify grid systems by administrative domains. In fact, doing so provides some insight into the type of grid and its capabilities.

Essentially an administrator, for the purpose of this list, is defined as someone with the power to start or stop computational resources and remove them from the grid. A taxonomy then might be organized from the standpoint of computer administration. Consider the following classification system proposed here:

- Category 1 – Includes government operated grids composed of the professionally administered high performance computing systems in their countries. Typically this would be a grid of supercomputers and other clusters connected by specialized high speed networks for academic or government research.
- Category 2 – Includes inter-organizational systems composed of high performance computing systems and networks in various countries or organizations that cross national and other organizational boundaries. These are arguably stable based on formal agreements.
- Category 3 – Includes intra-organizational systems including enterprise, academic and other organizations or virtual organizations where high performance computational resources, though diverse and heterogeneous, are professionally managed by a set of administrators who work for the organization.
- Category 4 – Includes intra-organizational desktop grid (cycle-stealing) systems where the individual user has the power to start and stop a computational resource. The individual user who works on the desktop has power to remove it from the grid system if by no other means than by turning it off. Additionally most of these systems suspend grid functionality when the system is in use locally. Please note however that in these systems users generally are employed by the organization or organizations that created the grid. They aren't likely to be intentionally malicious.
- Category 5 – Includes global or volunteer grid systems and peer to peer systems where individual desktop computer users volunteer their systems' unused computational cycles to a global grid system. In these systems the implication is that computational resources will come and go from the grid very frequently and some might even be considered malicious.

Some might argue, on a very sound basis, that this list is in fact inverted because, somewhat counter-intuitively, the most powerful systems computationally are those in Category 5. In any case, various pieces of this informal taxonomy of grid systems are often found in discussion of the various types of grid systems, desktop grids, global grids, enterprise grids, national grid projects etc. in the literature.

In point of fact, the interest here is in the computational grids. Or in the case of the above taxonomy based on administrative domains, the interest in our research is in Category 4 and Category 5, desktop grids and volunteer systems. Because the major interest in this research is in these two basic types of grid systems there is further discussion of desktop and global grids in a section specifically related to them.

Today there has been an evolution of the grid concept to include a global computing infrastructure often composed of large research centers connected by very fast networks such as the TeraGrid [25] and others. Several countries and research units have projects such as these. Some of these are computational grids and generally speaking are the most mature of the grid technologies. The very largest grids, however, in both computing power and numbers of nodes connected to the grids, are desktop and volunteer grids. Sometimes referred to as cycle scavenging grids, desktop grids offer a largely untapped resource for computational power. They also offer numerous challenges.

2.3 The Origin of the Grid

The term "grid" came into use in the mid-90's amid a computational world composed of High Performance Computing (HPC) and various types of computational clusters from IBM's RS6000 series of machines to off-the-shelf Beowulf clusters. [8;26]

In the 1980's Parallel Virtual Machine ran on distributed systems and was able to dynamically spawn processes to be executed. Later the Message Passing Interface (MPI), standard became the more widely used method of executing parallel computations on tightly coupled distributed systems. The term "grid" is often erroneously applied to these sorts of systems – loosely coupled clusters of computers, and sometimes misused to refer to tightly-coupled systems running "grid" software. Classic "Beowulf" type clusters typically spawn the same program from a master node to several worker nodes where each worker computes a different dataset and where the nodes communicate with one another when necessary to complete their own computations.

Because of the need for continuous reliable communication these systems often have internal proprietary networks and are made as reliable as possible in terms of node availability. In general the loss of one node causes the entire parallel program to block.

Because program speedup is generally bounded on such systems by communication [27] and network constraints as well as by Amdahl's Law [28], only so many nodes can be applied to a problem before no more speedup occurs. It should be noted that Amdahl's work was reexamined in 1988 by Gustafson who pointed out that increasing the amount of work with faster and faster processors actually reduces the impact of the serial portion of the code and increases efficiency. "As a first approximation, we have found that it is the parallel or vector part of a program that scales with the problem size. Times for vector start-up, program loading, serial bottlenecks, and I/O that make up the serial component of the run do not grow with problem size." [29] Still the point remains the same.

There is a subclass of problems that avoids the communication problem, however, because there is no communication between subprocesses. This class of problems has been referred to as "embarrassingly parallel." During the 90's and into the early part of the 21st century it became necessary for scientists and engineers to have access to a variety of systems, some of them tightly coupled clusters, some shared memory machines, some providing large amounts of storage from widely distributed geographical locations. Many of the problems under consideration were multidisciplinary in nature and involved people in different locations. And many were simply so computationally complex and large that the cost of the computation in a traditional computing center was simply too great for most researchers. Desktop and volunteer systems began to make an appearance.

In Evolution of the Grid [30], De Roure and others discuss three generations of grid computing which will be summarized here. The first generation of the grid began as an attempt join together supercomputer sites including the CASA project [31], FAFNER and I-WAY[32]. FAFNER, Factoring via Network-Enabled Recursion, was an attempt to factor large numbers by splitting and distributing tasks. I-WAY, Information Wide Area Year, was an attempt to link supercomputers using a resource broker. Work began on both around 1995. The term generally in use at the time was "metacomputing, " [33] popularized by Larry Smarr, former director of the National Center for Supercomputing Applications, around 1990. Following a 1997 workshop, "Building a Computational Grid," at Argonne National Laboratory in September 1997 in 1998, Ian Foster of Argonne National Laboratory and Carl Kesselman of the University of Southern California published "The Grid: Blueprint for a New Computing Infrastructure," [8] the seminal work on grid computing.

De Roure points to the second generation of grid development as outline in the 1st Edition of Foster's 1998 book -- and which to a large degree has been realized. He points to three issues that had to be confronted: heterogeneity, scalability and adaptability. Grid middleware solved many of these problems by hiding the underlying operating systems and machine types and providing a standard environment for users. A couple of projects have been most important in terms of providing middleware solutions to many of the problems inherent in the 2nd generation grid. Some of the middleware and systems developed during this period include Globus [10;34], Legion [35] and Condor [7]. Globus is a "low-level toolkit (that) provides basic mechanisms such as communication, authentication, network information, and data access. These mechanisms are used to construct various higher level metacomputing services."

Legion is an object oriented approach from the University of Virginia. Work began on the project in 1993. It was first released in 1997 and exists today as Avaki Corporation. At its inception, Grimshaw and others described Legion in this way [35]:

"When complete, Legion will provide a single, coherent virtual machine that addresses such issues as scalability, programming ease, fault tolerance, security, and site autonomy. Legion is a conceptual base for the sort of metasystem we seek. Our vision of Legion is a system consisting of millions of hosts and billions of objects co-existing in a loose confederation united through high-speed links. Users will have the illusion of a very powerful desktop computer through which they can manipulate objects."

UNICORE (Uniform Interface to Computer Resources) [36] in Germany was another second generation system. "The idea behind UNICORE is to support the users by hiding the system and site specific idiosyncrasies and by helping to develop distributed applications. Distributed applications within UNICORE are defined as multi-part applications where the different parts may run on different computer systems asynchronously or sequentially synchronized," according to Romberg.

De Roure points out that the second generation of the grid saw the development of a variety of tools and utilities providing services to users, as well as resource schedulers and other middleware. But De Roure also discusses a "more holistic" view of the grid with automation that, among other things, reconfigures itself dynamically, recovers from malfunction, protects against attack, implements open standards and optimizes resource use.

The 3rd generation grid incorporates Web Services along with some of the emerging standards from the World Wide Web Consortium, including things like SOAP, Simple Object Access Protocol, and Universal Description Discovery and Integration (UDDI) and others. His third generation grid also includes the Open Grid Services Architecture [37] which is gaining popularity as a standard. OGSA "defines a uniform exposed service semantics (the Grid service); defines standard mechanisms for creating, naming, and discovering transient Grid service instances; provides location transparency and multiple protocol bindings for service instances; and supports integration with underlying native platform facilities" among other things.

De Roure's 3rd Generation Grid also includes collaboration within virtual organizations [2] with various interactive services such as those provided by the Access Grid collaborative environment. The Access Grid [38] is an open source conferencing system that includes multiparty meetings with multi-source video and audio and presentation materials.

2.4 Desktop Grids and Volunteer Computing

While the concept of the grid might involve bringing high performance research computers together for the use of scientists and institutions from around the nation or the world, for the time being at least, these are ***not*** the most powerful computer systems in the world. Desktop grids, cycle scavenging systems, volunteer computing systems, peer-to-peer grids and global computing initiatives all are terms that refer to some of the largest systems in existence. In general these systems involve the use of software to harness the resources inherent in the unused cycles of various desktop computers in an organization, virtual organization or individual, whether on the internet or a local LAN. As such they meet our definition of a grid because each computer is under the control of the primary desktop user and hence are not centrally administered. A user might simply turn the system off while a computation is in progress for example.

These systems differ from clusters of desktop computers specifically "racked" for use as a "grid" under the control of a single system administrator. This indeed would not meet our definition of a grid because a central administrator would control the availability of individual machines in the cluster. It also would not provide the most important benefit – stealing otherwise unused or "free" cycles in a machine that otherwise would not be available to the grid users. In any case, attempts have been made to bring heterogeneous computing resources together for some time. Such "cycle stealing" systems have been used as early as the PARC (Xerox Palo Alto Research Center) Worm. [39]

Most traditional desktop grid systems, particularly enterprise systems which are owned by a single entity such as a corporation or university, operate by assigning tasks to daemons on worker hosts in the grid from a central server. Spawning of tasks generally depends on workload of the host to determine whether the host is available. Most systems allow tasks to be suspended when the keyboard on the computational host is used in order to avoid an unfavorable impact on the desktop user. Although task distribution operates somewhat differently on certain volunteer systems and P2P systems, job suspension and other concerns remain the same.

As discussed previously, the term "grid" has a marketing as well as a technical connotation. More recently the term grid has been used when businesses purchase inexpensive desktop computers explicitly and solely for use in a so-called "grid" and then link them with commercial middleware, forming a system where cycles aren't *harvested* so much as *cultivated* for use by the organization. Such a system might more aptly be termed a loosely-coupled cluster, perhaps more reminiscent of a Beowulf system [3] running grid middleware than the usual definition of a desktop grid. Why? Because in this case the computational resources are generally homogeneous, not distantly distributed and often in the same room, and reliable in the sense that the complete administration privilege of all computational resources resides with the administrator of the grid system. For our purposes, at least, these systems lack the more interesting problems associated with desktop grids of the classic definition.

Extremely large and widely distributed desktop grids are very useful for a subset of computational problems where communication between processes is not a significant issue. Although some work has been done in the area of a reliable message passing interface library for grid computing [40], many grid successes to date have involved large embarrassingly parallel computational problems. An embarrassingly parallel problem is one in which there is no communication between parallel tasks. In grid computing this sort of process is sometimes referred to as a Bag of Tasks (BoT) application. In such cases the speedup curve is relatively linear in relation to the number of processors used to solve the computation. Communication, other than some constant amount for setup and retention of results, does not exist. Often these are data parallel applications where the same program is sent to nodes on the grid, and, intentional redundancy notwithstanding, each computer considers a different dataset.

Volunteer computing is a term used for what have become the largest computational systems in the world where individual users on the internet volunteer the unused cycles of their desktop computers to some research effort. Because such efforts harness the unused cycles of desktop computers worldwide they sometimes are referred to as "global computing" [41] systems.

In his paper about BOINC, Berkeley Open Infrastructure for Network Computing [42], which is now the underlying framework powering several volunteer computing projects, David Anderson points out that:

> "No longer is the mass of computing power sitting in supercomputer systems at large institutions. Instead it is distributed in hundreds of millions of personal computers and game consoles belonging to the general public. Public-resource computing (also known as "Global Computing" or "Peer-to-peer computing") uses these resources to do scientific supercomputing."

In the mid-1990's two distributed systems used volunteered cycles to solve computational problems, GIMPS, the Great Internet Mersenne Prime Search, looked for Mersenne Primes [43] and distributed.net's software cracked encryption standards [44], announcing on 14 July 2002 that the RC5-64 key had been found after some 1,757 days. The system used the equivalent of 45,998 2GHz AMD Athlon XP machines at peak processing power and involved 331,252 people and their computers.

The first volunteer computing system that garnered a large amount of public attention was the SETI@Home project, in which volunteer computing is used to analyze radio signals in the search for extraterrestrial life. Plans for SETI@home were announced in 1998 with 3.91 million users of the client software in 226 countries by Aug. 2002. [45] SETI@home had performed $1.87 * 10^{21}$ floating point operations, the largest computation on record by 2002. SETI@home is being rewritten using BOINC, which provides middleware for volunteer computing projects. [42;46] Volunteers participate by running a

BOINC client program on their computer. The BOINC framework is being used by a number of other projects including Climateprediction.net [47], the Large Hadron Collider project CERN (LHC@home) [48] , Predictor@home [49], an attempt to predict protein structure from protein sequence, and many, many others. A well-kept list of global computing projects is maintained by Kirk Pearson [50].

BOINC, the relatively new volunteer computing software, has proved helpful to researchers and numerous BOINC projects have come into existence. Rather than researchers writing software for each of their projects, they can use BOINC. “In a single stroke,” David Anderson of UC Berkley told Science Magazine, "this has slashed the cost of creating a public-resource computing project from several hundreds of thousands of dollars to a few tens of thousands." [51] An interesting feature of BOINC is that clients register for multiple projects and can determine the percentage of time they want their machine to devote to a particular BOINC project.

XtremWeb is an older but somewhat similar middleware system [52] that was motivated by the needs of physicists at the Pierre Auger Observatory to run the same simulation program on 6.10^5 different inputs. The equivalent computing power was 6.10^6 hours on a 300Mhz PC each year. The XtremWeb was a platform for experimenting with global computing capabilities.

Commercial companies also have offered enterprise desktop grids including Entropia [53] and United Devices, which began in Austin, Texas in 1999 and now operates as Univa UD merging with Univa on September 17, 2007. Univa Corporation was founded in 2004 by Carl Kesselman, Ian Foster, and Steve Tuecke, who have been heavily involved with the Globus project and who have researched and written about grid computing since its inception. In 2004, IBM and United Devices started the World Community Grid [54] project which operates a number of volunteer computing projects including FightAIDS@Home.

Anderson points out that commercial systems for volunteer and "desktop grid" computing, such as United Devices and Entropia, "have roughly the same server functions as BOINC, and use relational databases to store task and participant data…." However, "these schedulers have functions that differ from BOINC's; they deal with complex workflows rather than single tasks, and they do not deal with redundancy and credit." [55]

More traditional desktop grid systems (if one can refer to anything so new to human society and culture as "traditional") differ in some respects from the major global and volunteer systems. The middleware used for more general desktop grids likely isn't suitable for global volunteer computing efforts. [42] Although BOINC might be useful for desktop grids. The main difference between the two is one of trust and to some extent homogeneity and volatility. Most traditional desktop grids or cycle stealing systems occur within the boundaries of some organization, even if it is a large one such as a university or large corporate enterprise. Groups of administrators likely install the grid software and control its removal. Although the grid software automatically starts and stops grid tasks on individual machines as users touch their keyboards, the use of the machine by the grid software is hidden and transparent to the user of the desktop machine. The machines can be assumed not to be malicious because they are owned by the organization. This in no way means they can be considered reliable however, and reliability is an issue that will be examined later. Networks tend to be faster and more reliable than in the case of volunteer computing.

In [56] the authors attempt a taxonomy of desktop grid systems in which they refer to both volunteer computing and desktop grids under the heading of "centralized" desktop grids. Newer systems utilizing peer to peer job dissemination are broadly categorized as "distributed." They list several such P2P systems including CCOF (Cluster Computing On The Fly) [57], Organic Grid [58] , Messor [59] and Paradropper [60].

Pointing out there has been no taxonomy of desktop grids as of 2007, the authors make a distinction between what this paper has termed volunteer computing and desktop computing "according to organization, platform, scale, and resource provider properties." Global volunteer systems would be distinguished from classic desktop grids by scale, Internet vs. internal LAN, and by resource provider -- workers in the enterprise in one case and volunteer computer owners in the other.

There are a plethora of grid technologies and middleware. A detailed description and comparison of four of them -- Gridbus, Globus, Legion, and Unicore – some of which have been discussed previously, is available from Asadzadeh, Rajkumar, Buyya and others in [17]. Globus is a special case in some respects because it supplies a set of low level tools to developers of other middleware. In addition to discussing each of these, the authors describe a typical hardware and software stack in a grid middleware system. Condor [7], another grid system that is still in extensive use and which uses process migration as a fault tolerance method on Linux systems (but not in the Windows version), was developed in 1991. Gridbus [24] is an open source grid software toolkit that was developed by the University of Melbourne GRIDS Lab and others. Major desktop operating system vendors also have offered Grid software including Xgrid from Apple and Alchemi [61], written for the Microsoft Windows operating system.

2.5 Reliability of Desktop and Volunteer Systems

In a desktop grid the various computational resources likely are heterogeneous, are spread across a wide geographic area and are connected by highly disparate networks with differing capabilities. Individual users might, or might not, have any interest or even awareness of the desktop's role as a computational resource in a grid. Machines might be turned off or rebooted at a whim. In addition the work of the grid usually suspends at any time when a user sits at the keyboard and begins to use the system. Some might have more memory than others or contain faster processors.

Network speed might be a factor and the amount of disk space might be different between machines. Some computers on the grid might not be able to contain the input dataset or might be so incredibly slow that it is virtually useless for the particular problem at hand. All of these considerations and many others combine to make this sort of grid computing unreliable. The most important and difficult problems may be categorized generally as the problem of volatility. [62]

In addition to failure of individual computational resources in the grid (often termed computational hosts or nodes) because of hardware issues, network failure or for other unforeseen reasons, such computers often are either owned or managed by individual users. Users start working with their machine, causing the grid task to suspend automatically, might remove the node from the grid or simply shut it off. Any of these activities would result in the failure or suspension of the particular grid application process.

Reliability is the central concern in this work. Generally in an enterprise desktop grid, the project is broken into tasks which are then sent from a server process to the computational resources that will do the actual work. How those resources are discovered and managed is handled differently in different systems and is an active area of research. In a computationally intensive job of any length there is a high probability that a task will fail due to the failure of a particular network or computational resources. A job running on a heavily used desktop grid almost certainly will not complete or will return only partial results depending on the size of the grid and the length of the job. It would be nice to have a more specific model of just how unreliable we can expect desktop grids to be, and some substantial work that has been done in this area will be considered here.

Overall, the area of reliability analysis is more complex than might appear upon first blush. So what is reliability? To have some basis for discussion consider that we discuss reliability in terms of probability of failure. In a series system where the reliability of each serially connected component is independent and the same, for example, reliability is:

P[failure] = $1-(1-P)^n$ where P is the probability of system survival and n is the number of components in a system where the probability of failure of each of the components is presumed to be the same.

A parallel system a system is considered to fail only if all of its components fail and so the general probability of success is: P^n where P is the probability of success.

Most real-world systems fit neither category completely and discussions of reliability generally revolve around "m out of n" systems where a system fails if m or more components of n components fail.

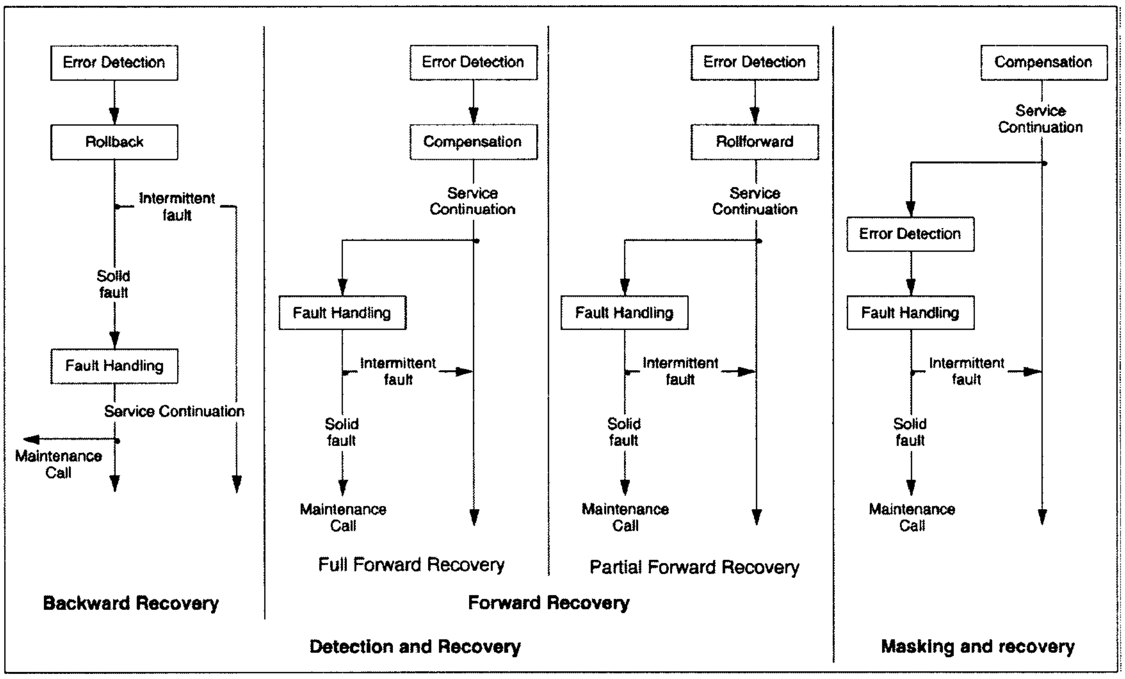


Figure 2.3: Examples of basic strategies for implementing fault tolerance (adapted from [63]).

Assuming the probability of failure of each component is the same and is independent, such a system fails with a probability:

$$P[\text{failure}] = P[\, M \geq m \,]$$
$$= 1 - F_{M;n}(m-1)$$

Where $F_{M;n}(m) = P[M \leq m]$ is the cumulative distribution function of M.

Rueda and Pawlak, University of Mantioba, have produced a brief survey of the pioneering work in general reliability theory during the past 50 years in [64]. "In Basic Concepts and Taxonomy of Dependable and Secure Computing" [63] the authors attempt to give precise definitions and a taxonomy of fault tolerant computing. They discuss system function and structure, threats to systems and a taxonomy of faults including natural and human, a discussion of faults, errors and failures, dependability and trust before moving into the area of fault prevention.

But what is reliability in terms of grid computing? Dai and others provide a definition in "Reliability Analysis of Grid Computing Systems" [65] and discuss two types of reliability -- system reliability and application reliability. "From the viewpoint of grid computing program, the program reliability can be defined as the probability of successful execution of the given program running on multiple nodes and exchanging information with the remote resources of other nodes. From the system point of view, the reliability of the grid system can be defined as the probability of all of the grid computing programs to be executed successfully in the grid computing environment."

Dai points out that the "grid program/system reliability is a special case of distributed program/system reliability" and provides a set of algorithms for evaluating the reliability of grid systems with emphasis not only on the computational resources but on the communication channels. A Minimal Resource Spanning Tree (MRST) connects all of the resources in the grid with the MRST reliability defined by:

- Reliability of all the links contained in the MRST during the communication.
- Reliability of all the nodes contained in the MRST during the Communication.
- Reliability of root node that executes the program during the processing time of the program.
- Total program reliability is defined by the probability of having at least one reliable MRST.

He defines grid system reliability as "the probability that all the computing programs are executed successfully. Thus, the grid system reliability equation can be written as the probability of the intersection of the set of MRST's of each program." The paper provides a formalism and algorithm for determining reliability based on a body of previous theoretical work involving the reliability of distributed systems. Although the paper provides a nice theoretical basis for discussing the reliability of grid systems it doesn't provide one a feel for exactly how unreliable actual grids really are.

Early work on Entropia [53] describes some experimental results relating to performance. In the case of a molecular docking program, for example, 50,000 molecules were partitioned into 10,000 slices of five molecules each. Ideally this job would have required only 10,000 subjobs, to complete, but in this case, 10,434 were required. The authors point out that most of the additional subjobs were caused by reboots. Some, however, were the result of variation in execution time. In order to ensure that jobs were completed the system initiated redundant subjobs when a subjob has failed to return within the anticipated period (determined by the user). The writers point out that relatively inexpensive grid resources are traded to improve job completion time. For the molecular docking program, the average subjob ran for 20 minutes but the range varied from 8 seconds to 118 minutes: Of the 10,000 subjobs, 204 of them ran more than the expected limit of 60 minutes. The average subjob execution time for the mixed grid was 20 minutes with a standard deviation of 13:4 minutes and a variance 181 minutes.

Their experience points out a central problem with desktop grid systems and fits with our own in an experiment involving pharmacophore search on an Apple XGrid of machines distributed at high schools in Kentucky.

Other work has involved case studies of operating grids. In [62] Kondo and others discussed the desktop grid in terms of how often cycles could be exploited and the distribution of time intervals where host is available for a grid application. They used Entropia DCGrid software at the San Diego Supercomptuer Center with 275 hosts. Of the 275 some 220 were running the Entropia client. From their analysis they determined the expected task failure based on the probability that a host would become unavailable before task completion, which is also understandably contingent on task length. They defined the concept of the cluster equivalence ratio: "Given an N-host desktop grid, how many nodes of dedicated cluster, M, with comparable CPU clock rates, are required such that the two platforms have equal utility?" Assuming a computational cluster based on the same processors as those in the desktop grid, researchers determined that for the desktop grid in question the 220 nodes completed equivalent work of a 209 host cluster on weekends and a 160-host cluster on weekdays when the desktops were more heavily used. In addition the tasks considered generally were a few minutes long. In terms of serious computational problems, a few minutes is not usually the time range in question. Some jobs might continue for hours in which case the failure rate (the rate of incomplete tasks) might increase to the point that the job is effectively stopped. In fact, in light of research done by Kondo and others, it might be that grids are ineffective for lengthy jobs and tasks must be curtailed in length to help ensure reliability.

Kondo and others at the University of California, San Diego, have produced a large amount of work related to desktop grid systems during the past few years. Their work with respect to availability of enterprise desktop grids is summarized in a 2006 paper [66] and later in 2007. [67] They have used application-level traces of four enterprise desktop grids and determined overall and per-host statistics. They point out that despite the popularity of desktop grids the volatility of hosts inside various grids hasn't been well understood.

In terms of methodology, the researchers used a "trace method" where they submitted tasks to a desktop grid that wasn't running other grid jobs. Each task wrote its computation rate at intervals to a file. The computers were kept loaded with tasks of about 10-minute length in a loop that performed a mix of integer and floating point operations. System availability was stored at 10-second intervals. Desktop users were unaware of the testing and tasks were suspended and terminated as necessary by keyboard usage, hardware failure etc. Data was collected from three desktop grids including an Entropia grid at the San Diego Super Computer Center at different times but for a cumulative period of about 28 days over 275 hosts. The second and third data sets were collected using the XtremWeb desktop grid software continuously over about one month on 100 hosts at the University of Paris-Sud including computers users in a classroom and others used by a research group. Other hosts were used by graduate students in the electrical engineering/computer science department at UC Berkeley. In

[68] Kondo and others use their previous results to generate mathematical models of grid availability and task success rate, among other things. They find that tasks can fail to meet their deadlines for two reasons; failure can occur if the aggregate compute power in the system dips below the incoming work rate and failure can occur if a task encounters repeated host failures. Kondo points out that even if the aggregate compute power in the system is always greater than the incoming work rate "host unavailability may still cause some tasks to fail in meeting their deadlines. This is particularly relevant ... where the intervals of availability tend to be quite small."

For a job to execute on a worker host, various conditions must be met. Kondo and colleagues define three types of availability that determine whether a job can run on a particular host and which help push forward our idea of reliability:

- Host availability includes the idea that the host is reachable for general communication. They list reasons for host unavailability as those sorts of things that would make the computer generally unavailable for use -- such as power failure, shutoff, reboot, crash.
- Task execution availability is determined by the grid software. The host might be too busy or the keyboard might be in use and therefore the system might be unavailable for use in the grid.
- CPU availability is the third consideration. If the host CPU is busy then most grid software will refuse to place a job on that host.

In addition to providing some sort of basis for a discussion of types of failure of hosts on the grid, the more immediate concern for their research is in determining what to trace in their attempt to measure reliability. According to Kondo, the completion of a task is related to the lengths of the intervals of time that a host is available to execute a job. Based on their description of the "temporal structure of resource availability" they derive the "expected task failure rate," which is the probability a host will become unavailable before a job is finished. The calculation was done by choosing several hundred thousand random points and checking task status at that time.

Kondo and the other researchers draw several important conclusions related to the expected task failure rate [69]:

- Even on the most volatile platform intervals of machine availability were 10 minutes in length or greater, while the mean length for all platforms with interactive users was about 2.6 hours. They report that an application developer could ensure that tasks are about 10 minutes long to best utilize most of the time intervals the machines were available.
- Task failure rates on each system were correlated with the task size in an approximately linear fashion.
- On platforms with interactive users, execution availability tends to be independent across hosts. However, independence is affected by the configuration issues including wake-on-LAN enabled Ethernet adapters etc.
- The availability interval lengths are not related to clock rate; nor is the percentage of time a host is unavailable. However, interval lengths in terms of number of operations and task failure rates are correlated with clock rates. So selecting computational resources with higher clock rates may be beneficial.
- There is wide variation of availability from host to host, especially in the platforms with interactive users, even in platforms with hosts of identical clock rates. So computational nodes with the same hardware showed heterogeneous efficiency in terms of the grid application.

So the most efficiency came with tasks that were 10 minutes long. The average task failed in 2.6 hours. The task failure rate is a linear function of task size (length). In [69] they noted:

> "We also find that the expected task failure rate is strongly dependent on the task lengths. (The weekends show similar linear trends, albeit the failure rates are lower.) It appears that in all

> platforms the task failure rate increases with task size and that the increase is roughly linear; the lowest correlation coefficient is 0.98, indicating that there exists a strong linear relationship between task size and failure rate. (Clearly, as the task size approaches infinity, the task failure rate will eventually plateau as it approaches one. Nevertheless, the relationship is approximately linear for a reasonable range of task sizes.)".

On systems with interactive users, where a user might type on the keyboard and stop a grid task, the availability of hosts in the system tend to be independent of one another and availability can be increased by using hardware that allows the network card to wake the system. Faster CPUs do not correlate with the system availability but if more work is done in a shorter interval of time then a faster CPU might be helpful.

More recent research agrees with the results from Kondo. In 2007, Iosup and others examined resource availability on a large scale, multi-cluster experimental grid platform in France, Grid 5000. [21] They found that the mean time before failure was short – about 12 minutes for the grid as a hole, about 5 hours at the individual cluster level and at about two days per compute node.

Khalili and others looked at TeraGrid and the earth sciences grid, Geon [70], showing 55 and 80 percent success rates.

Others have attempted to quantify the availability of similar grid systems. In [71] Brevik and others describe a methodology for predicting machine availability based on monitoring data in distributed computing environments. They estimated a specified quantile for the distribution of availability, and associated a confidence level with each estimate. They state the problem the following way:

> "From a set of availability measurements taken from a resource … and given a desired percentile p and confidence level c, what is the largest availability duration t for which we can say with confidence c that p percent of the availability time measurements are greater than or equal to? The answer to this question for a given data set, percentile of interest (and take $q = 1 - p$), and desired confidence level, is a lower bound estimate of the qth quantile from the data set. While not a prediction of the exact availability duration, using an estimate of a quantile provides a lower bound on how long a machine (or collections of machines) is likely to be available, and the confidence measure provides a quantitative (but probabilistic) "guarantee" of the estimate's accuracy."

Volunteer grids are even more volatile than conventional cycle stealing systems in institutional and enterprise settings. Anderson notes in [42] that volunteer computing, what he terms "public resource computing," " involves an asymmetric relationship between projects and participants."

Projects are typically small academic research groups with limited computer expertise and manpower. Most participants are individuals who own Windows, Macintosh and Linux PCs, connected to the Internet by telephone or cable modems or DSL, and often behind network-address translators (NATs) or firewalls. The computers are frequently turned off or disconnected from the Internet. Participants are not computer experts, and participate in a project only if they are interested in it and receive incentives such as credit and screensaver graphics. Projects have no control over participants, and cannot prevent malicious behavior." [42]

In "Volunteer Availability Based Fault Tolerant Scheduling Mechanism in Desktop Grid Computing Environment" [72], Choi and others discuss volatility in volunteer desktop grids along with proposing a scheduling mechanism.

They define types of execution and formalize failure modes. A public execution is the execution of a task as a volunteer and might be started or stopped arbitrarily. Private execution is the execution of a private job by a personal user, often the owner of the computer. They refer to failures caused by the execution of a private job as a volunteer autonomy failure.

Volunteer autonomy failures can result in livelock if traditional job scheduling methods are used because the consistent interruption of the job can cause it never to complete. Their paper formally defines several failure modes. Their definitions are summarized here and the mathematical formalism has been excluded for the sake of brevity and is available in [72]:

> Definition 1: Volunteer volatility failure is abortion of public execution which is caused by freely leaving of the public execution of a task.
>
> Definition 2: Volunteer interference failure is temporary suspension of public execution which is caused by private execution of a individual job.

They point out a livelock problem occurs when all systems executing a task have a volunteer volatility failure.

So desktop grids are somewhat unreliable at the very least -- and sometimes unreliable in the extreme. However, system reliability appears to be quantifiable and possibly predictable. It may also be possible to mitigate system unreliability with a variety of methods with varying efficacy. In [73] two methods are used to predict reliability, a parametric model fitting method using past data to find the underlying probability distribution and two a non-parametric techniques ("resampling" and "the bionomial method"). There is a fairly large body of older work in reliability analysis on distributed systems as well as in the area of software reliability.

2.6 Strategies for Reliability

The primary method to be considered in this research for improving reliability is redundancy, replication or over-provisioning as it is sometimes termed. Very often the same tasks and data are replicated across different computational hosts in a grid in an effort to overcome high failure rates. The interest in this research is in improved methods for adding redundancy to grid jobs. In general where and how to add redundancy to a series-parallel system is in fact NP-Hard [74]. (In brief explanation, a nondeterministic polynomial-time hard problem is at least as difficult as the hardest problems in NP, such as an NP-complete problem. A common example for an NP-complete problem is the subset sum problem. Does the sum of some non-empty subset of a set of integers, other than the empty set, sum to 0? But an NP-Hard problem need not be a decision problem and therefore a member of NP. It's possible that these problems cannot be solved in polynomial time but this has yet to be proved.)

It should come as no surprise, considering the inherent unreliability of desktop grids, that the search for various methods to ensure fault tolerance and reliability is an active area of research. Most of the research is centered around grid middleware development including various proposals and methods for building reliability through the very well known and time tested mechanisms of redundancy/overprovisioning and checkpointing, byzantine results checking, as well as resource scheduling for reliability and other methodologies.

In relatively recent 2008 work, Kandaswamy, Mandal and Reed discuss migration and overprovisioning as strategies for fault tolerance.[75] Most of the work in the area of scheduling has as its underlying paradigm the "task parallel" model of parallel computing. [55;76-81] In the Bag of Tasks (BoT) model the tasks are presumed to be independent and embarrassingly parallel.

In this paradigm, the task to be solved is broken into subtasks. The subtasks are placed in a shared data structure called a bag, "and each process in a pool of identical workers then repeatedly retrieves a subtask description from the bag, solves it, and outputs the solution." Advantages of this programming approach include "transparent scalability, automatic load balancing, and … easy extension to fault-tolerant operation." [82]

In addition, some work has been done in the area of data parallelism specifically with regard to heterogeneous machines in the area of load balancing which is closely related to the area of reliability. Assigning jobs to particular resources in a grid is the job of the scheduler. The job of the scheduler is to arrange tasks in such a way as to assure completion in the minimal time. Traditional scheduling of independent tasks, the kind of job most often associated with grid computing, generally reduces into bin-packing problems, an area which has been the focus of much study for some time. In the particular case of grid computing, however, the task length cannot be known a priori (a requirement of bin-packing schedulers) because relevant information about the computational power of the machine often is not available. Many schedulers are based on attempting to determine how long a particular task might take on a certain machine in the grid, either from historical data or through other means. Other schedulers, however, attempt to solve the problem without a priori knowledge by using replication. An excellent overview of the current literature with regard to replication schedulers is available in the 2007 paper, "On the Efficacy, Efficiency and Emergent Behavior of Task Replication in Large Distributed Systems" [83]. A lot of work has been done in the area of scheduling in general as in [84] for example.

Replication schedulers send copies of tasks to various machines and make use of the first to complete. The desire is to reduce the makespan of the job where the makespan is the time from the beginning of the first task to the completion of the last. The minimum makespan is the result of optimal scheduling so that the time from the beginning of the first subtask to the completion of the last is the minimum possible.

Work flows often are represented on grid systems as directed acyclic graphs or DAGs. Most schedulers fall into general categories including list-based, clustering and duplication based. List-based strategies orders the nodes in the DAG and then assigns each to a resource that minimizes a cost function. A newer strategy is that of the level based scheduler where the DAG is broken into levels. Nodes in each level are scheduled as soon as scheduling is complete for nodes in the previous level. A comparison of the schedulers is available in the 2007 paper, "Relative Performance of Scheduling Algorithms in Grid Environments" [85].

In their 2009 paper, Zhang and others propose combining fault tolerance with over-provisioning and checkpointing with the HEFT (Heterogeneous Earliest Finish Time) and DSH (Duplication Scheduling Heuristic). [86]

Some past replication schedulers include “eager scheduler” from the Charlotte grid [87] and MapReduce [88]. WQR (work queue with replication) [78] schedules all tasks first and then starts replicating them with a limit on the number of replicants so a task with a programming error won’t hang execution by continuously failing. WQR (work queue with replication) [78] schedules all tasks first and then starts replicating them with a limit on the number of replicants so a task with a programming error won’t hang execution by continuously failing In [89] researchers modify WQR to take advantage of knowledge about resources.

Adler and others develop a model of Heterogeneous Networks of Workstations (HNOW) [76] and claim to prove that FIFO (First In First Out) “ worksharing protocols provide asymptotically optimal solutions to a problem related to sharing a bag of identically complex tasks in a heterogeneous network of workstations (HNOW) *N*.” They continue:

> “The main results of the paper establish that, for every HNOW N, over lifespans of sufficient duration, any protocol that orchestrates N’s workstations in a FIFO fashion—i.e. that has workstations finish working, and return their results, in the same order as they receive work—provides an optimal solution for the HNOW-Exploitation Problem. As part of this demonstration, we prove that, no matter how N’s workstations differ in work rate, all protocols that observe a FIFO regimen provide equally productive solutions for the HNOW-Exploitation Problem for N. These results are somewhat surprising, since they demonstrate that over sufficiently long lifespans, there is no advantage to specifying an ordering that favors the faster workstations, for example by sending work to the fastest workstation first and having it return its results last. In fact, in our model, we can completely ignore the relative powers of N’s workstations. [76]”

In [79;90] the authors examine the efficacy of several schedulers and introduce RR (list scheduling with Round-robin order Replication), a task replication scheduler meant for parameter sweep applications on a computational grid. They also point out that makespan is an untenable algorithm for heterogeneous grids because the total computational power varies over time. They suggest that a scheduler should be concerned with consuming all of the computational cycles that were available over time rather than makespan. "RR is akin to WQR with infinite replication, except that it considers tasks with the same length and it does not schedule them at random, but rather from a circular list." [83] Ghare and colleagues look at whether processors should be used for additional tasks or for redundancy in specific scenarios. [91]

In [92] Kondo and others examine and propose four general approaches:

- Resource Prioritization – Hosts are sorted according to criteria such as clock rate, historical performance.
- Resource Exclusion Using a Fixed Threshold – Hosts with poor performance are excluded according to some measure such as clock rate.
- Resource Exclusion via Makespan Prediction – Exclude hosts not expected to complete a certain application within a certain expected time.
- Task Replication – Overcome the problem of task failure by replicating the task on multiple hosts or on faster hosts.

Some point out that Kondo and his colleagues are considering grids where the number of tasks is closely matched with the number of execution hosts and that replication schedulers have performance comparable to bin packing schedulers. [83]

Mapping tasks to processors has been the subject of research for some time. In a 1977 paper [93] for example, the authors consider the finishing time properties of several algorithms for scheduling ***n*** independent tasks on ***m*** non-identical processors in an attempt to find the optimal algorithm from several presented.

Most research focuses on finding the optimal makespan. The authors consider two bin-packing schedulers in [83] including FPLTF, Fastest Processor to Largest Task First, which uses task size, resource load and resource speed. And they consider sufferage. The sufferage value of a task is the difference between the best completion time and the second best, according the capabilities of each available resource. Tasks that would suffer most are assigned first.

The authors also analyze efficiency of replication. "Task replication enables knowledge-free schedulers to attain performance comparable to knowledge-based schedulers. This comes at an increased use of computation, because multiple replicas consume more computational resources than a single one," according to Cirne and others. "However, scheduling BoT applications on grids is still an open problem," Silva says in [78]. "Good scheduling requires good information about the grid resources, which is often difficult to obtain. Known knowledge-free scheduling algorithms usually have worse performance than algorithms that have full knowledge about the environment."

Others have looked at data decomposition rather than strict task scheduling as a method for load balancing on heterogeneous machines. [94] Others propose Natural Block Data Decomposition which maps data to processes according to the relative performance of the process, among other methods. [95] [96]

Scheduling algorithms that work toward the most efficient use of resources have implications with regard to reliability but aren't necessarily designed as mechanisms of fault tolerance. Lee and others examine the current state of grid reliability and fault tolerance in their in their paper "Grid Programming Models: Current Tools, Issues and Directions." [97]

They say that:

> "Reliability and fault tolerance in grid programming models/tools are largely unexplored, beyond simple checkpointing and restart. Certain application domains are more amendable to fault tolerance than other, e.g., parameter sweep or Monte Carlo simulations that are composed of many independent cases where a case can simply be redone if it fails for any reason. The issue here, however, is how to make grid programming models and tools inherently more reliable and fault tolerant. Clearly a distinction exists between reliability and fault tolerance in the application versus in the programming model/tool versus in the grid infrastructure itself. An argument can be made that reliability and fault tolerance have to be available at all lower levels to be possible at the higher levels."

Their remarks accurately summarize the context and thinking that motivates our own research. "A further distinction can be made between fault detection, fault notification and fault recovery," Lee says. "In a distributed grid environment, simply being able to detect when a fault has occurred is crucial. Propagating notification of that fault to relevant sites is also critical. Finally these relevant sites must be able to take action to recover from or limit the effects of the fault." They go on to point to event models as being a necessary element for reliable and fault tolerant programming models and tools.

Sarmenta has produced a solid overall exploration of the topic in terms of volunteer computing systems in his paper "Sabotage-Tolerance Mechanisms for Volunteer Computing Systems." [98] He discusses redundancy, the ratio of the average total number of work objects assigned to workers using redundancy versus the original number of workers. Slowdown is taken from the runtime of the computation with and without redundancy. Sarmenta says that:

> "In general, fault-tolerance mechanisms should aim to (in order of priority) (1) minimize the final error rate as much as possible, or at least reduce it to an acceptable level, (2) minimize redundancy, and (3) minimize slowdown."

Sarmenta also discusses strategies such as spot checking results by duplication to detect problems, blacklisting offending resources, majority voting and other methods. Software that "simulates the behavior of an eager scheduling work pool in the presence of saboteurs and various fault-tolerance mechanisms" was used to check results. He develops the "credibility threshold principle." The key idea in credibility-based fault-tolerance is that: "if we only accept a result for a work entry when the conditional probability of that result being correct is at least some threshold Θ, then the probability of accepting a correct result, averaged over all work entries, would be at least Θ."

In their 2006 and 2007 papers regarding replication and checkpointing in grid systems, [99;100] Chtepen and others point out that many systems still do not implement any form of fault tolerance. They point out that Condor implements checkpointing and Charlotte uses eager scheduling for replication. More recently BOINC implements replication and results checking on an application specific level. Results checking is a continuing field of study. [101] Chtepen's statements notwithstanding, some newer grid systems do implement some checkpointing and redundancy. BOINC will be discussed in more detail later, but it might be noted at this time that some researchers are aware of the need for improved fault tolerance. In "The Challenge of Volunteer Computing With Lengthy Climate Model Simulations," [102] the authors describe the need for checkpointing and their strategy for implementing it:

> "Scientific applications being ported to the volunteer computing paradigm must checkpoint time-consuming tasks (e.g. greater than an hour of run-time). This enables a restart of the task with little loss of previously computed work. Many scientific applications were meant to be run continuously from "start to finish," with job submission by researchers who patiently await the results on a time-shared system, and who do not interrupt the task. Therefore, scientific programs often have no checkpointing capability. For a volunteer computing app this is not desirable, as user intervention, system crashes, and other factors may require a task to be paused, stopped, or removed from memory, and later restarted. Fortunately, checkpointing is available for the climate model used in climateprediction.net. "

The climate model checkpoint is about 20 MB and the checkpoint is written every 144 timesteps or about every 15 minutes on a 2 GHZ Pentium.

In their paper, Chtepen examine two well-known techniques for providing fault-tolerance in grids -- periodic task checkpointing and replication. Checkpointing periodically saves task status. "Task replication is based on the assumption that the probability of a single resource failure is much higher than of a simultaneous failure of multiple resources."

They suggest a task replication algorithm based on replication of arriving tasks. In each scheduler iteration, the longest waiting task, of which less than a certain number of replicas are started, is distributed to the site with free resources and the smallest number of replicas. Load is calculated as a combination of the number of tasks and the speed of the resources in millions of instructions per second. When one task replica finishes, other replicas are deleted. Their Adaptive Task Replication algorithm would stop task replication during peak loads on the grid. Failure Detection suggests that the scheduler reschedule all resources sent to a particular resource as soon as a failure is detected. The Failure Detection and Adaptive Task Replication method combines the two previous algorithms. The algorithms were tested using a software simulator written for the purpose, the Dynamic Scheduling in Distributed Environments simulator developed in C++. They eschewed use of other grid simulators including GridSim, SimGrid and NSGrid "because the possibilities of modeling grid system dynamics are quite limited."

They found that "... heuristics with failure detection guarantee almost lossless task execution at the cost of slower system performance while replication-based algorithms provide good throughput on unreliable grids without giving a guarantee on the number of jobs lost. A compromise between performance and reliability can be achieved by combining failure prevention with rescheduling. To achieve the best result, an appropriate number of replicas should be chosen in function of the properties of the grid system at hand."

The authors also discuss checkpointing, noting that the efficiency of checkpointing is dependent on the length of the checkpointing interval. Their paper presents heuristics that tune the checkpointing interval.

In a 2007 paper about fault tolerance in peer to peer systems [103] the authors discuss two different types of rollback recovery as a method for fault tolerance, including checkpoint-based and log based where interprocess messages are replayed to rebuild the job status. They compare the approaches by looking at the failure free overhead, i.e. the additional time required for the fault-tolerance mechanism without failure and the recovery time required by a failure during execution.

They say that a checkpoint-based technique provides a low failure-free overhead but a long recovery time while a log-based mechanism requires more constant overhead but recovery is faster. Choosing one of those two approaches highly depends on the characteristics of the application and of the underlying hardware. Their paper provides a method to deploy technical information about grid resources allowing the system to make configuration decisions thereby helping ensure reliability.

In their 2006 paper, "Using Checkpointing to Enhance Turnaround Time in Institutional Desktop Grids" [104], Domingues and colleagues discuss the need for placing checkpoints in central storage rather than on the host machine where the job is taking place. "There are two main types of checkpoints: system-level and application-level. Apart from Condor, which relies on system-level checkpoint, all major middleware tools that implement checkpointing, such as BOINC and XtremWeb, make use of application-level checkpointing. An important issue regarding checkpointing lies in the physical location where checkpoints are stored. A limitation of the existing middleware like BOINC is that checkpoints are private, being stored in the same machine where the task is running. If that machine becomes unavailable, the checkpoint file cannot be used and the task has to be restarted from scratch in another machine."

The authors go on to talk about the benefits of storing checkpoints in a central location so tasks can be restarted on another machine should a machine or host fail. As mentioned previously, Condor implements checkpointing and even implements job migration on Linux systems.

In "Fault Tolerance within a Grid Environment," [105] the authors summarize their progress in developing a fault model for grid computing. They also are developing a system that "uses one or more coordination services, constructed under a distributed recovery block scheme, to locate compute resources on the Grid, and to schedule, broadcast, receive and vote upon jobs submitted by a client program. This is in order to not only reduce the likelihood of faulty results being received by the client, but also to protect against malicious Grid resources deliberately altering the results they produce." Their work is somewhat similar to the system implemented in BOINC.

BOINC deserves more discussion because of its growing popularity as a framework for volunteer computing, and because of its specific use of a redundancy/quorum mechanism for fault tolerance. As outlined in [42] BOINC is designed in the following way:

A BOINC project corresponds to an organization or research group that does public-resource computing. The project is identified by a home page URL. Participants register with projects on the web page. The BOINC project server is centered around a relational database that stores descriptions of applications, platforms, versions, workunits, results, accounts, teams, and so on. Server functions are performed by a set of web services and daemon processes. BOINC provides tools for creating, starting, stopping and querying projects; adding new applications, platforms, and application versions, creating workunits and for other functionality. The system also has rich facilities for maintaining redundant sources of file data and upload/download information, whether data should remain resident on execution clients and so on.

Of primary interest here, of course, is BOINC's facilities for redundant computing. "Public-resource computing projects must deal with erroneous computational results. These results arise from malfunctioning computers (typically induced by overclocking) and occasionally from malicious participants," David Anderson, leader of the BOINC project, says. BOINC provides support for redundant computing and "a mechanism for identifying and rejecting erroneous results." The framework uses an M of N quorum type system. A project can specify that N results should be created for each work unit meaning that N work units have been replicated on different machines. After $M \leq N$ have been completed an "application specific" function compares the results and selects a "canonical result." If no result can be found, the process is repeated until a maximum count or timeout is reached.

For cases where result comparison is difficult because of machine architecture differences, specifically because of differences in numerical expression, BOINC provides a homogeneous redundancy feature. When it is enabled the BOINC scheduler selects only hosts with the same operating system and CPU vendor.

2.7 General Fault Tolerance and Replication

There are many, many sources for software reliability research. NASA, however, has produced a nice discussion of software fault tolerance [106] for single version software as well as some multiversion techniques. Fault tolerance techniques discussed include system structuring and closure, atomic actions, inline fault detection, exception handling and others. Multiversion techniques include the idea that software may be built differently so that if one version fails another should continue to work. Recovery blocks, N version programming [107], N self-checking programming, consensus recovery blocks, and t/(n-1) techniques are reviewed.

Of specific interest in our research however is the way in which processors are assigned tasks and data. Some of the research has focused on efficiency in terms of dynamically assigning tasks to processors best able to carry them out in a timely fashion and with concern for balancing the load so that tasks complete within some specific timeframe even though they are running on heterogeneous processors with different capabilities.

Although not particularly related to the area of grid computing, a couple of other sources are of interest because of their relationship to the particular set of replication techniques that will be presented in this paper.

In "A Repetitive Fault Tolerance Model for Parallel Programs" [108] Yen and others propose a replicative model for data parallel programs somewhat similar to the system proposed in our work on grid computing -- although their discussion is related to

VLSI technology on the microprocessor level. In their repetitive fault tolerance model, processors are permuted so that the working processors can execute the tasks that were originally assigned to faulty processors. A permutation function *F* is responsible for computing the processor permutation. After each iteration, the working processors are permuted differently to execute the unfinished tasks due to failures.

The replicative model to be discussed in this dissertation proposal also involves permutation of workload -- not so much as relates to obvious replication of tasks, but to the permutation of data, and not on a single chip, but across a desktop grid. Their work also involves the use of the important concept of the permutation function.

Another interested concept in distribution of replicated entities comes from work involved with maximizing disk throughput by striping, "Maximizing Throughput in Replicated Disk Striping of Variable Bit-Rate Streams." [109] Of particular interest is the discussion of data redundancy policies. They discuss Deterministic Replica Placement where data is placed on disks in a "round-robin" fashion, and a Random Replica Placement, where data is placed on random disks, which is of interest when attempting to replicate tasks and data across execution hosts in a grid.

CHAPTER 3

MODELLING REDUNDANCY FOR FAULT TOLERANCE

One way to improve the reliability of desktop and volunteer grid systems is through the use of redundancy – spawning the same task to different nodes in the grid to improve the probability that the overall job completes. The focus of this research has been to develop and test a paradigm for building reliability into grid applications -- and possibly for future use in grid middleware – by changing the way that tasks and data are arranged and distributed to the hosts that make up the grid. Various methods for static redundancy, for spawning tasks and data to the nodes in the grid, have been modeled. They are then tested and analyzed in a specially developed grid simulator (discussed in Chapter 5) before a job is run on an actual grid (described in Chapter 6). The outcome has been a better understanding of the effects of redundancy in the face of node failure. The overall concept of permuting tasks and data before distributing them in redundant fashion across computational hosts is described generally as RPP, the Replication and Permutation Paradigm. RPP improves on the simple replication of tasks often used to improve the reliability of desktop and volunteer grids. Two methods are of particular interest as seen in initial development of the RPP model in Section 3.1.

The larger the number of hosts used for a grid computation, the less likely the computation will complete, assuming no redundancy. The reason is simply that each node in a desktop or volunteer grid has an associated mean time to failure as do network links and the main server. In reality this means that machines might be turned off unexpectedly in a volunteer grid, or that disk drives might be full or that an inferior network at some location might be overloaded or fail. The greater the number of machines in the grid, the more likely one will fail within a certain time period. As seen from the literature review of the previous chapter, replication has been proposed, and in some cases implemented in grid middleware, as a method of increasing job efficiency and overcoming node failures.

If a parallel job is considered to have failed when one node, with a probability of failure F_i, of the grid of n nodes fails then the failure model is that of a simple serial system. In such a system, the probability of failure F = 1-S where S is the probability of system survival. Such a serial system is made up of individual units, all of which must function for the system to succeed. In the special case where the probability of failure of each component in the series, F_i, is the same for n components, the probability of system failure is given by $F=1-(1-F_i)^n$. It's easy to see that the probability of failure increases very quickly with the size of the grid. Attempts have been made to quantify metrics associated with various grids, and, overall, it has been found that grid tasks must be of relatively short duration in order to avoid being disrupted by node failures – particularly those caused when a desktop user takes control of the system by typing on the keyboard. Note the work by Kondo and colleagues cited earlier.[62;66;68;69;80;92;110-112]

Because the probability that at least one node will fail increases with the number of nodes used in a job, the mean time to failure M decreases as the number of nodes increases. Large jobs use many nodes and sometimes have long execution times E. As the number of nodes and execution times grow to fit large cutting edge problems, M becomes $M <= E$ and the job will often fail.

This unfortunate set of facts has been born out in preliminary research at the University of Louisville. A small to medium sized series of jobs was run on Kentucky's Apple Xgrid of desktops at Kentucky high schools. [6] Although the overall results were successful in that the grid produced shorter run times than ever, the fact remained that there were severe outliers. Worse, because of the ownership and structure of the grid, jobs were submitted by 4 p.m. for an evening run that was forced to terminate early the next morning. Any incomplete subtasks were then resubmitted the following afternoon. So the actual wall-time makespan of a job could cover several days.

Generally the response to these problems in the research community has been to restructure the grid infrastructure, and in particular the scheduler, to resubmit or replicate failed tasks on the grid in order to reduce the makespan of the job or some similar measure of overall job performance. Makespan, sometimes referred to as Cmax, is a term used in scheduling research to refer to define the total execution time for the schedule. In terms of grids, "makespan" is used to refer to the span of time from the beginning of the first subtask of a job to the completion of the last. Often the goal is efficient use of grid resources in an attempt to match the most costly jobs with the most efficient resources. Replication often has been within the paradigm of course-grained task replication where individual tasks are dynamically resubmitted to other grid resources until the overall job completes.

Rather than following a course-grained, approach to scheduling and replication, RPP supports permutation of data and tasks across the grid in a way that assures job completion by using replication to minimize the impact of failures while also minimizing makespan.

Usually, when a job is submitted on a grid, the data and executable are sent from a master or submission node to a number of computational nodes on the system. The computational workers begin to execute the code, consuming the data as they go, and at the end of the job the temporarily stored output is returned to the master, or some other central location, by some means.

It is possible to arrange and disseminate the initial data to execution hosts in such a way that given a known number of failures the worst-case completion time for a grid job can stated exactly.

3.1 A Replication and Permutation Model

A set theoretic analysis and model is developed in this section to discuss various types of redundancy or over-provisioning for reliability on highly unreliable grids. The finished model includes two suggested methods for replication – a Latin Square distribution of tasks and data across computational hosts and "reverse mirroring" where data and tasks are duplicated in reverse order across hosts. The Latin Square method proves most interesting and dominates the research. Also of interest is that the Latin

Square method does not reduce performance unless a host failure occurs, and the job will continue so long as one processor is present. A proof is developed in Chapter 4 showing how Latin Square distribution allows prediction of maximum makespan given a known host failure rate.

The formal grid model, including data d, tasks t, a job J, permutations of data p, hosts h, sets of hosts H and lists of data L among other items, is now presented:

Some n-tuple d must be defined as the minimal or atomic input data object that is required for a single task t to complete and produce a meaningful n-tuple of output.

(3.1) Let $d=\{a_1,\ldots,a_n\}$

A job J is composed of a bag of identical tasks t which consume non-identical lists L of n-tuples d.

(3.2) Let $J=\{t_i|0<i<N+1\}$ Where N is the cardinality of some subset of available processors in the grid.

Each J will be assigned to some set of host processors H composed of individual hosts h.

(3.3) Let $H=\{h_i|0<i<N+1\}$ Where N is cardinality of some subset of the available processors in the grid.

(3.4) Let $L=\{d_1,\ldots,d_k\}$ Where d is a minimal data item and L is an ordered list or permutation of such items.

(3.5) Let $P_0=\{p_1,\ldots,p_N\}$ Where P_0 is a broken k/N-element permutation of L.

We define a broken permutation P_0 of L, a k/N-element permutation, into N permutations $(p_1,\ldots,p_N)$ where the number possible is given $L(k,N) = \binom{k}{N}(k-1)_{k-N}$. Because each element of L is independent with regard to any task t_i any of the numerous possible permutations is acceptable.

Table 3.1: Broken Permutations of Data

{h_1,t}	**{h_2,t_2}**	**{h_3,t_3}**
{p_1}	{p_2}	{p_3}

Table 3.2: Data items d_i contained in permutations p_i

{h_1,t_1}	**{h_2,t_2}**	**{h_3,t_3}**
d_1	d_5	d_9
d_2	d_6	d_{10}
d_3	d_7	d_{11}
d_4	d_8	d_{12}

Table 3.3: Three processors and associated tasks execute three permutations of data

Execution Step	**{h_1,t_1,p_1}**	**{h_2,t_2,p_2}**	**{h_3,t_3,p_3}**
1	d_1	d_5	d_9
2	d_2	d_6	d_{10}
3	d_3	d_7	d_{11}
4	d_4	d_8	d_{12}

Table 3.2 shows the individual data items d_i that are contained in previously discussed permutations p_i. Table 3.3 shows a possible configuration of a grid where three

processors execute the data supplied them as outline in the above formal model. Note that the data permutations are consumed in order, and that the above model thus supplies no redundancy. Also note that there is no implication of concurrency so that each execution step might take more or less time on a particular processor. Execution order, however, is preserved. Obviously, should execution be interrupted some in some processor then some data would not be consumed and would be lost. Some grid systems attempt to correct this deficiency by course-grained task replication or by dynamic scheduling of tasks.

In Table 3.4 execution is interrupted on processor h3 at the beginning of timestep 3. The task is resubmitted and execution begins at timestep 4. The entire makespan of the job is increased to 7 timesteps. If the partial results were maintained, the timesteps would have been increased to 5.

Table 3.4: Host h_3 fails at timestep 3 and a new task is dynamically instantiated

Execution Step	$\{h_1,t_1,p_1\}$	$\{h_2,t_2,p_2\}$	$\{h_3,t_3,p_3\}$	$\{h_4,t_3',p_3'\}$
1	d_1	d_5	d_9	
2	d_2	d_6	d_{10}	
3	d_3	d_7	d_{11}	
4	d_4	d_8	d_{12}	d_9
5				d_{10}
6				d_{11}
7				d_{12}

Another obvious way to overcome the problem is by "mirroring" or multiple replication of tasks:

(3.6) Let $L=(d_1,\ldots,d_k)$ Where d is a minimal data item and L is an ordered list or permutation of such items.

We define a broken permutation p_0 of L, a k/N-element permutation, into N permutations $p_0=(p_1,\ldots,p_N)$

Additionally the model creates redundancy by creating an additional set of identical permutations.

(3.7) $p_0'=(p_1',\ldots,p_N')$. Let $p_0=p_0 \cup p_0'$
(3.8) Let $J=\{t_i|0<i<2N+1\}$
(3.9) Let $H=\{h_i|0<i<2N+1\}$

Apply a bijection of L to J and a bijection of J to H resulting in a mapping of p_i and p_i' from p_0 and to t_i and t_i to h_i from $i=1;i<2N+1$ and so by composition a mapping of p_i to h_i. This results in simple course-grained mirroring of tasks and data to some subset of processors on the grid where the number of processors used is 2N.

Table 3.5: Execution fails on h_3 during timestep 3 while the job continues satisfactorily because of redundancy on processor h_6

Execution Step	$\{h_1,t_1,p_1\}$	$\{h_2,t_2,p_2\}$	$\{h_3,t_3,p_3\}$	$\{h_4,t_4,p_1'\}$	$\{h_5,t_5,p_2'\}$	$\{h_6,t_6,p_3'\}$
1	d_1	d_5	d_9	d_1	d_5	d_9
2	d_2	d_6	d_{10}	d_2	d_6	d_{10}
3	d_3	d_7	d_{11}	d_3	d_7	d_{11}
4	d_4	d_8	d_{12}	d_4	d_8	d_{12}

This example as illustrated by Table 3.5 results in simple mirroring where tasks and data are duplicated once across 2N processors. It is important to note here, however, that execution time of the job, the makespan, requires 4 timesteps in the case that there is a processor failure as well as in the case where there is no processor failure. It also is easy to see that in the case that two processors fail, if those two processors happen to be processing p_i and p_i' then the entire job will fail. Given a large set of several hundred processors, it isn't unlikely that two such mirrored replicants might fail.

A way to improve reliability is to increase the number of duplicated tasks and data, which reduces the failure rate but still does not guarantee completion of the job. In the circumstance where all of the processors happen to be processing replicants of one another when they fail, the job will fail even though most of the processors in the system continue to process tasks. Also, the makespan of a job in which there is no task failure does not decrease. The makespan is the same in the case of failure and in the case of no failure.

A new replication and distribution method, reverse mirroring, reduces makespan in the case of no failures and acts like simple mirroring in the case of failure:

(3.10) $L=(d_1,\dots,d_k)$ Where d is a minimal data item and L is an ordered list or permutation of such items.
(3.11) Let $L'=(d_k,\dots,d_1)$ So that L' is the inverse of the permutation L.

We define a broken permutation p_0 of L, a k/N-element permutation, into N permutations $p_0=(p_1,\dots,p_N)$.

We define a broken permutation p_0' of L', a k/N-element permutation, into N permutations $p_{0'}=(p_{1'},\dots,p_{N'})$

(3.12) Let p0=p0 U p0'
(3.13) Let $J=\{t_i|0<i<2N+1\}$
(3.14) Let $H=\{h_i|0<i<2N+1\}$

Apply a bijection of L to J and a bijection of J to H resulting in a mapping of p_i and p_i' from p_0 to t_i and t_i to h_i from i=1;i<2N+1 and so by composition a mapping of p_i to h_i. The result is a bit different than simple mirroring and has interesting connotations for job execution. See Table 3.6.

Table 3.6: Data permutations p_i composed of d_i are mirrored and inverted in p_i' composed of d_i'

Execution Step	$\{h_1,t_1,p_1\}$	$\{h_2,t_2,p_2\}$	$\{h_3,t_3,p_3\}$	$\{h_4,t_4,p_1'\}$	$\{h_5,t_5,p_2'\}$	$\{h_6,t_6,p_3'\}$
1	d_1	d_5	d_9	d_{12}'	d_8'	d_4'
2	d_2	d_6	d_{10}	d_{11}'	d_7'	d_3'
3	d_3	d_7	d_{11}	d_{10}'	d_6'	d_2'
4	d_4	d_8	d_{12}	d_9'	d_5'	d_1'

As shown in Table 3.6, even with the failure of h_3 in timestep 3, the makespan of the job requires only two timesteps where four was required with simple mirroring. Data items d_{11} and d_{12} are executed by h_4 in timesteps 1 and 2.

Note that in this scenario data is duplicated across two processors as in mirroring except that the order of each of the broken permutations has been reversed prior to mapping to a task and processor. The result of reordering is that no two minimal data items $\{d_i,\ldots,d_k\}$ are duplicated in the same timestep. In this case:

- All of the processor cycles are fully utilized in processing new data during the first two timesteps so that if there are no failures the makespan of this job is two timesteps rather than four, as in the case of simple mirroring of data.
- A failure is less likely because it must occur during the first two timesteps. The probability of failure decreases with decreasing runtimes so that the overall probability of failure

is less with this scheme that with simple mirroring because runtime is less.

- In the case of failure in timestep two, makespan is increased by one timestep. Of course this behavior remains to be generalized to other cases.
- In the case of failure in timestep 1, i.e. complete node failure, makespan is increased to that of simple mirroring, four timesteps.

Reverse mirroring should result in half the computational time as a job with two replicants if the grid has no node failures. In the case of failure of one mirror in traditional mirroring the makespan T(M) is just T or the sum of the total timesteps $T=\sum t_i$ in the job no matter when the failure occurs. However if $t(F)_i$ represents failure after the i^{th} timestep and m represents the number of mirrors then in the case of reverse mirroring one might argue that total timesteps is represented by Equation 3.15:

$$(3.15)\quad T(M)=\left(\frac{T}{m}-t(F)i\right)+T/m$$

Where T(M) is total timesteps, m=number of mirrors and $t(F)_i$ is the timestep where failure occurs.

The problem remains, however, that if two nodes fail and they happen to be processing replicated data, p_i and p_i' then the job will fail. In this case the addition of replicants reduces the probability of failure but does not ensure job completion. It is possible, however, to find a permutation function used with replication that ensures job completion.

Table 3.7: The arrangement guarantees job completion so long as one processor remains functional

Execution Step	$\{h_1,t_1\}$	$\{h_2,t_2\}$	$\{h_3,t_3\}$
1	p_1	p_2	p_3
2	p_2	p_3	p_1
3	p_3	p_1	p_2

The arrangement of data, depicted in Table 3.7, does indeed solve the problem inherent in both our previous replicaton scheme and in simple mirroring—that if two processors while processing the same or a permuted replicant of the data then the entire job will fail. Even if all but one processor fail the entire job will be executed, albeit in a longer time. Because each broken permutation p_i holds four data items d_i, the arrangements shown in Table 3.7 involve a large amount of replication of data; some of the replication may be reduced by delayed transfer of data. Because each processor has all of the data it needs to proceed after the first iteration of data is transferred, there is no need to wait for the remaining data. In fact, transfer of the remaining data should be delayed by some amount of time, which is an issue that should be explored more fully in later research. The data distribution outline in Table 3.7 is a Latin Square of Order 3. Early work with Latin Squares was done by Leonhard Paul Euler, 1707–1783, a Swiss mathematician and physicist.

The Latin Square is an arrangement of items or objects into rows and columns in such a way that no row or column contains the same object more than once.

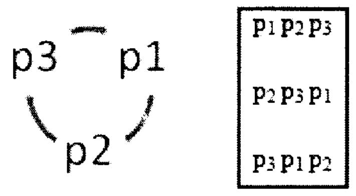


Figure 3.1: Moving clockwise around the circle on the left while advancing the starting point by one generates the linear arrangement on the right.

More specifically a Latin Square is a quasigroup Q defined in terms of a set of distinct symbols and the binary multiplication operation between the elements of the set Q. The quasigroup's multiplication table is a Latin Square.

Because, the model deals with permuations of data, another more natural way of considering the formation of a Latin Square is in terms of a linear arrangment of a circular permutation of data. This concept can be illustrated by examining Figure 3.1, beginning with p_1 and generating a permutation by listing the items in linear fashion. Then offset one place to p_2 and generate a permutation listing the items in linear fashion, and repeat the process with p_3. A linear arrangement of the circular permutation has been generated. It's easy to see that this is the same Latin Square developed previously.

For a circular permutation of n objects there are n linear arrangements of the objects. In Table 3.7 each of the objects refers to a circular permutation of $p_0=\{p_1,\ldots,p_n\}$. Recall that each of the broken permutations $\{p_i \mid 0<i<n\}$ itself is a permutation of minimal data items d and so has an intrinsic order.

Note from Table 3.7, that if one proceeds through the execution steps one step is required if no processor fails, two if one processor fails and three if two processors fail. It must be strongly noted that a single execution step does not refer to the execution of a minimal data item but an entire broken permutation of data p_i. The Latin Square design places an upper limit on makespan depending on the number of host processes that fail. Notice in Figure 3.1 that if one host fails, two timesetps are required for the computation. If two hosts fail, three timesteps are required. The concept of an upper bound on computation time is examined more fully in Chapter 4. Another way of viewing the model is that as the probability of failure increases, the probability that the makespan will increase also increases. In fact the amount of data is increased to $\Theta(n^2)$ where n is the number of data items. The full upper limit on makespan is the number of timesteps required to process all of the data on one node.

After consideration of two distinct mappings of data across processors we find that the first, mirroring with permutation of data items, results in reduction of makespan and improves reliability. However, it leaves unresolved the problem that a job may fail to repeat if all replicants fail. We also found that the second mapping, arrangement of permutations in a Latin Square, results in assurance that a job evenutally will complete even with h-1 processor failures but with greatly increased computational time and data replication.

It is possible to combine the two schemes in an attempt to reduce makespan in most cases while ensuring completion in those extreme cases where large numbers of processors fail and all replicants of a particular set of data items are destroyed. This will be examined in more detail in Chapter 4.

As shown in Table 3.8, if h_4 fails in timestep 1 then the job will complete in timestep 4 on h_3. If both h_3 and h_4 fail then the job will complete in timestep 6 depending on when the earlier failures occurred.

Table 3.8: The table depicts mirroring with inverted broken permutations of data arranged in a Latin Square configuration

Execution Step	$\{h_1,t_1,p_1\}$	$\{h_2,t_2,p_2\}$	$\{h_3,t_3,p_3\}$	$\{h_4,t_4,p_1'\}$	$\{h_5,t_5,p_2'\}$	$\{h_6,t_6,p_3'\}$
1	d_1	d_5	d_9	d_{12}'	d_8'	d_4'
2	d_2	d_6	d_{10}	d_{11}'	d_7'	d_3'
3	d_3	d_7	d_{11}	d_{10}'	d_6'	d_2'
4	d_4	d_8	d_{12}	d_9'	d_5'	d_1'
Execution Step	$\{h_1,t_1,p_2\}$	$\{h_2,t_2,p_3\}$	$\{h_3,t_3,p_1\}$	$\{h_4,t_4,p_2'\}$	$\{h_5,t_5,p_3'\}$	$\{h_6,t_6,p_1'\}$
5	d_5	d_9	d_1	d_8'	d_4'	d_{12}'
6	d_6	d_{10}	d_2	d_7'	d_3'	d_{11}'
7	d_7	d_{11}	d_3	d_6'	d_2'	d_{10}'
8	d_8	d_{12}	d_4	d_5'	d_1'	d_9'
Execution Step	$\{h_1,t_1,p_3\}$	$\{h_2,t_2,p_1\}$	$\{h_2,t_3,p_2\}$	$\{h_4,t_4,p_3'\}$	$\{h_5,t_5,p_1'\}$	$\{h_6,t_6,p_2'\}$
9	d_9	d_1	d_5	d_4'	d_{12}'	d_8'
10	d_{10}	d_2	d_6	d_3'	d_{11}'	d_7'
11	d_{11}	d_3	d_7	d_2'	d_{10}'	d_6'
12	d_{12}	d_4	d_8	d_1'	d_9'	d_5'

Note also from Table 3.8 that the job will complete in two timesteps if no processors fail. So long as both copies of mirrored data do not fail, any other failure will result in completion of the job in four timesteps. If both replicants fail, but no other nodes fail, the job will complete in a maximum of 8 timesteps. If all nodes but one fail, the job will complete in 12 timesteps. Completion is assured so long as one processor continues to function.

Both features of the basic RPP model, reverse mirroring and Latin Square replication, have been created. Analysis of the performance of reverse mirroring and Latin Square replication in specially constructed grid simulation software indicates that both have similar performance in terms of host failure with the exception that jobs fail as node failures increase with reverse mirroring. A complete discussion of various simulation results is presented in Chapter 5. Suffice it to say that the Latin Square replication method appears to be the most interesting of the two.

CHAPTER 4

PROOF OF MAXIMUM MAKESPAN WITH LATIN SQUARE REPLICATION

This chapter includes further examination of Latin Squares which produces some interesting results which allow prediction of maximum makespan in the face of particular node failure rates. Recall that the columns of a Latin Square in the model represent host/task pairs and the rows contain permutations of data. Each row is processed in temporal order so that each row also can be thought of as representing a timestep in the computation.

Table 4.1: The table depicts broken permutations of data in a Latin Square

Execution Step	**$\{h_1,t_1,p_1\}$**	**$\{h_2,t_2,p_2\}$**	**$\{h_3,t_3,p_3\}$**
1	d_1	d_5	d_9
2	d_2	d_6	d_{10}
3	d_3	d_7	d_{11}
4	d_4	d_8	d_{12}
Execution Step	**$\{h_1,t_1,p_2\}$**	**$\{h_2,t_2,p_3\}$**	**$\{h_3,t_3,p_1\}$**
5	d_5	d_9	d_1
6	d_6	d_{10}	d_2
7	d_7	d_{11}	d_3
8	d_8	d_{12}	d_4
Execution Step	**$\{h_1,t_1,p_3\}$**	**$\{h_2,t_2,p_1\}$**	**$\{h_3,t_3,p_2\}$**
9	d_9	d_1	d_5
10	d_{10}	d_2	d_6
11	d_{11}	d_3	d_7
12	d_{12}	d_4	d_8

As shown in Table 4.1, if h_3 fails in timestep 1 then p_3 is not processed by h_3. It will next be processed by h_2 in execution step five, and if that fails then p_3 will be processed by h_1 in execution step nine. This seems intuitive based on our diagram, but it is necessary to show that it is true in all cases of the model.

Previously a Latin Square was constructed rather informally as a linear arrangement of circular permutations. More formally, a Latin Square of order n is defined as an n x n table or square matrix in which n symbols occur once in each row and once in each column. [113] If the first row and first column of the Latin Square are in some natural order such as $\{1,2,\ldots,n\}$ then the square is said to be reduced [114], standard[113] or normalized [115].

An n x n Latin Square in which each row is derived from any other is a cyclic Latin Square. [113] By creating the previously discussed Latin Square in circular fashion in a natural order, the result has been a standard, cyclic Latin Square. In other words it is a Latin Square in which the rows are composed of cyclic permutations of a set $S=\{a_1,a_2,\ldots,a_n\}$ which may be ordered. A permutation is a one to one transformation of a finite set into itself. [116] In terms of combinatorics, a permutation is considered to be a sequence of distinct elements. In terms of group theory, a permutation is a bijection, a bijective function, from a finite set onto itself. If no particular element is mapped to itself (a fixed point) the permutation is a derangement.

Consider the cyclic permutation of order 3 created previously in Figure 3.1, a cyclic permutation that contains one cycle. Other permutations, including those creating disjoint cycles and so forth, will not be considered here. Additionally the permutations are created with offset 1 because each item shifts or rotates by one item each time a new derangement is created. Given these constraints it is possible to define a relatively simple bijection that also is a cyclic permutation or cycle. Consider the following:

Let S be a finite set of n symbols { a_0, a_1, ..., a_{n-1}}. Consider a bijection or transition function ϕ (theta) such that $a_1\phi = a_2$, $a_2\phi=a_3$, ..., $a_n\phi=a_1$. More generally $a_i\phi=a_{i+1}$ where $a_{n+1}=a_1$. If (a_0, a_1, ... a_{n-1}) defines a cycle including 1 cycle of n length one might say that $\sigma(a_i)=a_{i+1}$, where $a_{n+1}=a_1$ (or where all subscripts are taken modulo n.)

The σ (sigma) function defines a mapping of S such that ($a_0 \Rightarrow a_1 \Rightarrow a_2 ... \Rightarrow a_{n-1}$), which is a cycle.

Additionally squaring the function σ^2 carries a_i to a_{i+2} and $\sigma^k=a_{i+k}$, where all subscripts are reduced modulo n. Applying σ, the following set of linear arrangements are created with each application of the function:

Table 4.2: The table depicts a Latin Square of order n

a_0,	a_1,	...	a_{n-1}
a_1,	...	a_{n-1}	a_0
a_{n-1},	a_0,	...	a_{n-2}

In Table 4.2, a cyclic Latin Square has been defined, in which the rows are cyclic permutations of offset one and cycle length n. Although the symbols differ, this Latin Square is equivalent to that used in the RPP model.

The rows are derangements and have no fixed points. Notice however that σ is a mapping from one symbol to another. The resulting mappings are equivalent in terms of permutation. However it is obvious that their linear arrangements differ. The linear arrangements are concerned with position, and it is possible to obtain a positional mapping from the permutation.

Let's begin by looking at the bijective permutation functions. If $a_1\phi = a_2$ then by definition a_2 occupies the position formerly occupied by a_1. In fact a left rotation has occurred. Consider the function σ. If $\sigma(a_i)=a_{i+1 \bmod k}$ then a_{i+1} is shifted to the position formerly occupied by a_i. This can be seen by considering that $\sigma(a_{i-1})=a_{i \bmod k}$.

It is possible to derive a similar function that returns the position of a_i in a linear arrangement following application of σ. Consider that we have both a set of symbols $S=\{a_0,a_1,\ldots,a_{n-1}\}$ and a set P of positions ρ (rho) including $\rho_j \in \{\rho_0,\rho_1,\ldots\rho_{n-1}\}$ in a linear arrangement of S.

Table 4.3: The table depicts a Latin Square of order n labeled with positional information

ρ_0	ρ_1	…	ρ_{n-1}
a_0,	a_1,	…	a_{n-1}
a_1,	…	a_{n-1}	a_0
a_{n-1},	a_0,	…	a_{n-2}

Notice in Table 4.3, for example, that position ρ_0 in row 1 contains a_0 and that ρ_0 in row two contains a_1 etc. Position information may be included in our function in the following way:

Let $\sigma(a_i\rho_j)=a_{i+1 \bmod k}, \rho_j$ Note that the symbol changes from row to row where the position ρ remains the same.

Remember that the symbols relate to data in the programming paradigm. Columns represent computational hosts and rows represent timesteps when data is processed. At this point we see intuitively that if a column is removed the symbol will next be found in the following row in the column immediately to the left, unless it is the first column. In that case the symbol will be shifted into the n-1 column position. We are not so interested in the mapping of a symbol to another symbol in the same position, but in the position of the same symbol in the linear arrangement as it moves from row to row.

Position can be described in the following way. The rows are linear arrangements of cyclic permutations following each application of the bijective function σ. Consider a function $f(a_i,\rho_j)=a_i,\rho_{((j-1)+n) \bmod n}$ where symbol a_i remains constant and position ρ_j changes between the rows subject to $j=((j-1)+n) \bmod n$. This describes a situation where a symbol is shifted one position to the left and where position ρ_{-1} is taken to be position ρ_{n-1}.

Two formal results are now presented and proven. Used together, these two theorems will allow prediction of makespan.

Theorem 1: $\forall$ a_i f defines a mapping of the position ρ_j of a_i following application of σ: $f(\rho_j)=\rho_{((j-1)+n) \bmod n}$ where $\rho_j \in P=\{\rho_0,\rho_1,\ldots\rho_{n-1}\}$ holds between two rows for any ρ_i in an n x n standard Latin Square composed of cyclic derangements of offset 1.

Proof of Theorem 1 will be in two parts.

In part 1 we argue by mathematical induction on j that the theorem applies between two linear arrangements as outlined above.

(1) Base Case:

When j =0, $f(\rho_0)=\rho_{((0-1)+\rho n) \bmod n}$

$= \rho_{(n -1 \bmod n)}$

$=\rho_{(n-1)}$

So that if symbol a_i is in ρ_0, after application of function σ, a_i will be in $\rho_{(n-1)}$.

(2) By Induction:

When j=k, f(ρk)=ρ((k-1)+n) mod n which is our original definition.

When j=k+1, which implies k+1>0 because P={ρ0,ρ1,...ρn}, f(ρk+1)=ρ((k+1-1)+n) mod n.

=ρ(k+n) mod n

=ρ(k mod n)

=ρ(k) which is correct because position has shifted left from k+1 to k, just as position shifts from k to k-1.

In part 2 we show that the position function f is valid for a permutation of any length >2 as defined previously. Because we have shown j to be valid for the position of each a_i, we can fix j and show that it is valid for any value of n > 1.

(1) Base Case:

When n = 2, $f(\rho_j)=\rho_{((j-1)+2) \bmod 2}$

When j=1, $f(\rho_1)=\rho_{((1-1+2) \bmod 2}$

$=\rho_{(2 \bmod 2)}$

$=\rho_{(0)}$ so that an item in position one moves to position 0.

(2) By Induction:

When j=1 and n=k, $f(\rho_1)=\rho_{((1-1)+k) \bmod k}$

$=\rho_{(k \bmod k)}$

$=\rho_{(0)}$ so that an item in position 1 moves to position 0.

When j=1 and n=k+1, $f(\rho_1)=\rho_{((1-1)+k-1) \bmod k-1}$

$=\rho_{(k-1 \bmod k-1)}$

$=\rho_{(0)}$

Thus Theorem 1 shows that the position of any a_i shifts left by one place with application of the cyclic permutation function σ and by induction that the function is applicable for one cycle permutations of length > 2 as defined above.

We have shown that the position of any a_i shifts by 1 in a linear arrangement based on the previously defined permutation. It remains to show that the function is applicable to any number of rows and through any number of applications of the permutation.

<u>Theorem 2</u>: $\forall$ a_i: $f^t(\rho_j)=\rho_{((j-t)+n) \bmod n}$ where $\rho_j \in \{\rho_0,\rho_1,\ldots\rho_{n-1}\}$ holds between rows for any p_i in an n x n standard Latin Square composed of cyclic derangements of offset 1.

Proof of Theorem 2:
In this proof we show by mathematical induction on t that the position function f is valid for the composition of bijective function f and show that the position of a_i shifts to the left by offset one with each application of the function.

If f is a function on P then f^1 is the identity function, f^2 is the composition of f with itself ($f \circ f$) such that $f^2(\rho_j)=\rho_{((j-2)+n) \bmod n}$ and f^t is defined by: $f^t(\rho_j)=\rho_{((j-t)+n) \bmod n}$

(1) Base Case:
When t=1: $f^1(\rho_j)= \rho_{((j-1)+n) \bmod n}$ which is Theorem 1.

(2) By Induction:
When t=k: $f^k(\rho_j)= \rho_{((j-k)+n) \bmod n}$ which is simply by definition.
When t=k+1: $f^{k+1}= f^k(\rho_j) \circ f^1(\rho_j)$
$= \rho_{((j-k+1)+n) \bmod n}$
$=f^{k+1}(\rho_j)$

In addition to formalizing the RPP model, which previously has been show to enhance reliability in a grid simulator, the two theorems together provide a method of predicting makespan based either on past performance of a grid or current failure rate of hosts in a grid.

Consider the Latin Square of order 4 in Table 4.4. A particular data item is available for processing in the column (host) immediately to the left and in the following timestep when a host is lost.

Table 4.4: The table depicts a Latin Square of order 4 labeled with positional information

p_0	p_1	p_2	p_3
$\{h_1,t_1,p_1\}$	$\{h_2,t_2,p_2\}$	$\{h_3,t_3,p_3\}$	$\{h_4,t_4,p_4\}$
d_0	d_1	d_2	d_3
d_1	d_2	d_3	d_0
d_2	d_3	d_0	d_1
d_3	d_0	d_1	d_2

If host h_3 fails on startup then data item d_2 will next be considered in timestep 2 by host h_2. If h_2 has failed then d_2 will next be considered by host h_1 in timestep 3. If host h_1 fails then d_2 will be processed by host h_4 in timestep 4.

In the RPP using a Latin Square data distribution, Make(max)=F+1 where Make(max) reflects the makespan in terms of timesteps, and F is the total number of failed hosts. Failures are assumed to occur before the first data item is processed for the worst case.

Assuming each data item consumes one timestep, the formula for makespan requires adjustment to include the number of data items in a permutation p and, hence, the number of timesteps required to process each permutation. The new formula, Make(max)=(F+1)*(length(p_i)), allows prediction of computational time on unreliable grids when using RPP.

In addition to providing a method for obtaining reliability on highly unreliable grid systems, Latin Square replication also allows prediction of makespan in the face of failures among the hosts on a grid system. Further formalization of the RPP model has allowed formal proof of upper bound for makespan using Latin Squares in the distribution of data and tasks on computational grids.

CHAPTER 5

GRID SIMULATION

This chapter presents a simple grid simulator designed specifically to compare the behavior of various data and task arrangements, including RPP, in the face of node failure. The grid simulator consists of a C# implementation of an object oriented program containing a node class that simulates grid execution hosts, along with a master node class and job class. The probability of failure for execution hosts may be arbitrarily set, along with the number of job repetitions. Additional features include:

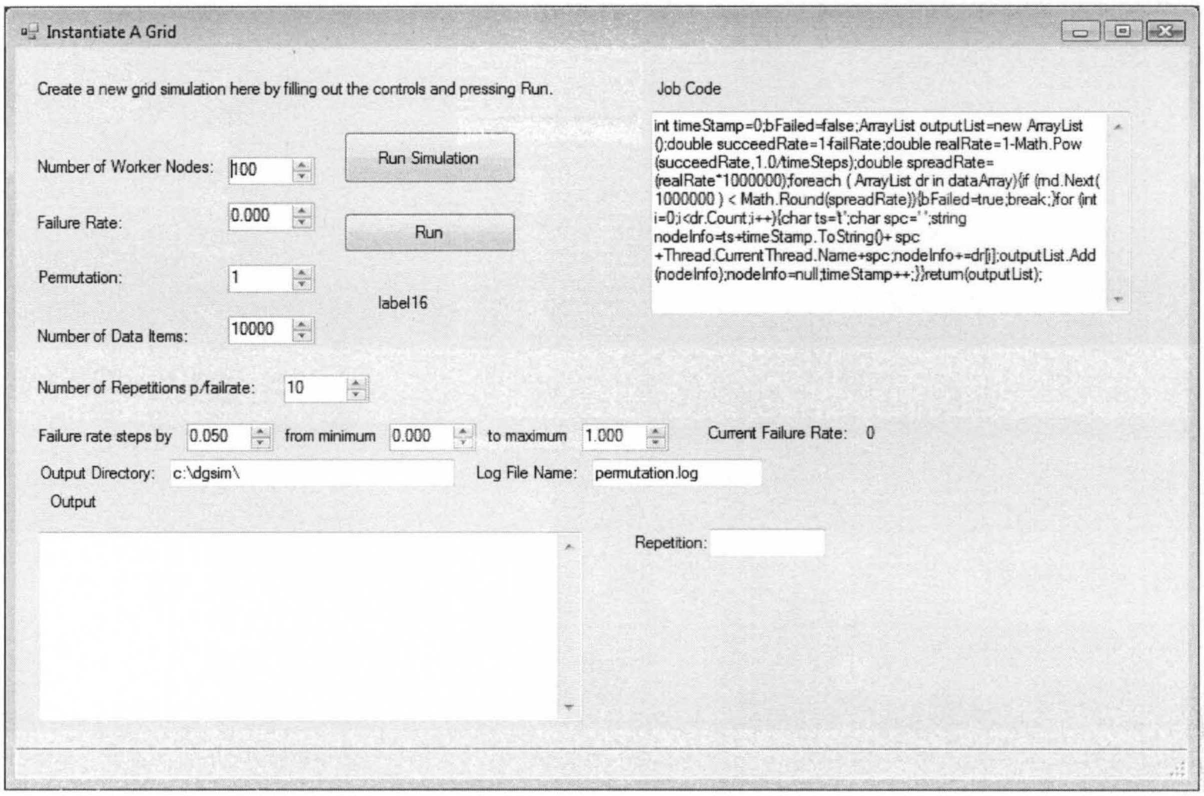
- Permutations of data may be defined and passed through a job class to various computational node classes running in separate threads.
- Nodes may be failed with a specific probability using a pseudorandom distribution.
- Small functions may be written in C# and dynamically compiled without requiring recompilation of the simulator.
- Makespan may be captured.

The simulator allows test jobs to be submitted and run with different data replication, permutation and failure conditions. It allows testing traditional mirroring and multiple replicas, both with a variety of failure rates, against the RPP approach with a variety of failure rates. Data may be passed to node class instances in any of several configurations. The code that will be run on the grid, in other words the set of identical tasks, is compiled at runtime and passed to execution hosts and executed in independent threads. Each thread considers a data item, generates a pseudorandom number to determine whether to fail according to the probability of failure set by the researcher and then executes one timestep by consuming one data item.

5.1 Hardware and Software

The simulator, DGSim or DGSimulator (Data Grid Simulator) was developed in C# on Microsoft Visual Studio to run on machines running the Windows operating system. During the simulation runs the software was executed on an HP Compaq tc4200 notebook computer running Windows Vista on a single-core, 32-bit Intel Pentium M 1.86 GHz processor with 1,500 MB RAM. Runtime for the grid simulation was in the range of a few minutes to hours depending on the number of hosts, data items and job repetitions. A typical run included ten jobs each with 10,000 data items on 100 threads. Essentially the processing required by the software was a simple random number generation to determine whether the processor was to fail and then an output of the data item, so the jobs were computationally not particularly lengthy. The hardware was chosen for portability, availability and ease of use.

C# and Microsoft Visual Studio were chosen because the visual development environment allows faster development times. C# is a relatively new language that tends to be internally consistent and easy to use with garbage collection and a wide range of functionality for creating threads, generating pseudorandom numbers and for creating objects in general. Its C++-like syntax makes it familiar to many developers.


Instantiate A Grid

Create a new grid simulation here by filling out the controls and pressing Run.

Number of Worker Nodes: 100

Failure Rate: 0.000

Permutation: 1

Number of Data Items: 10000

Number of Repetitions p/failrate: 10

Failure rate steps by 0.050 from minimum 0.000 to maximum 1.000 Current Failure Rate: 0

Output Directory: c:\dgsim\ Log File Name: permutation.log

Output

Repetition:

Job Code

```
int timeStamp=0;bFailed=false;ArrayList outputList=new ArrayList
();double succeedRate=1-failRate;double realRate=1-Math.Pow
(succeedRate,1.0/timeSteps);double spreadRate=
(realRate*1000000);foreach ( ArrayList dr in dataArray){if (rnd.Next(
1000000 ) < Math.Round(spreadRate)){bFailed=true;break;}for (int
i=0;i<dr.Count;i++){char ts='\t';char spc=' ';string
nodeInfo=ts+timeStamp.ToString()+ spc
+Thread.CurrentThread.Name+spc;nodeInfo+=dr[i];outputList.Add
(nodeInfo);nodeInfo=null;timeStamp++;}}return(outputList);
```

Figure 5.1: The DGSimulator GUI

In a typical run of the DGSim software, the program is initialized and the form class run. The GUI is instantiated as shown in Figure 5.1. The form requires the following parameters to be set:

- The number of worker nodes
- The permutation number
- The number of replicants
- The number of data items
- The number of repetitions per failure rate
- The minimum failure rate, the maximum failure rate and the step size.

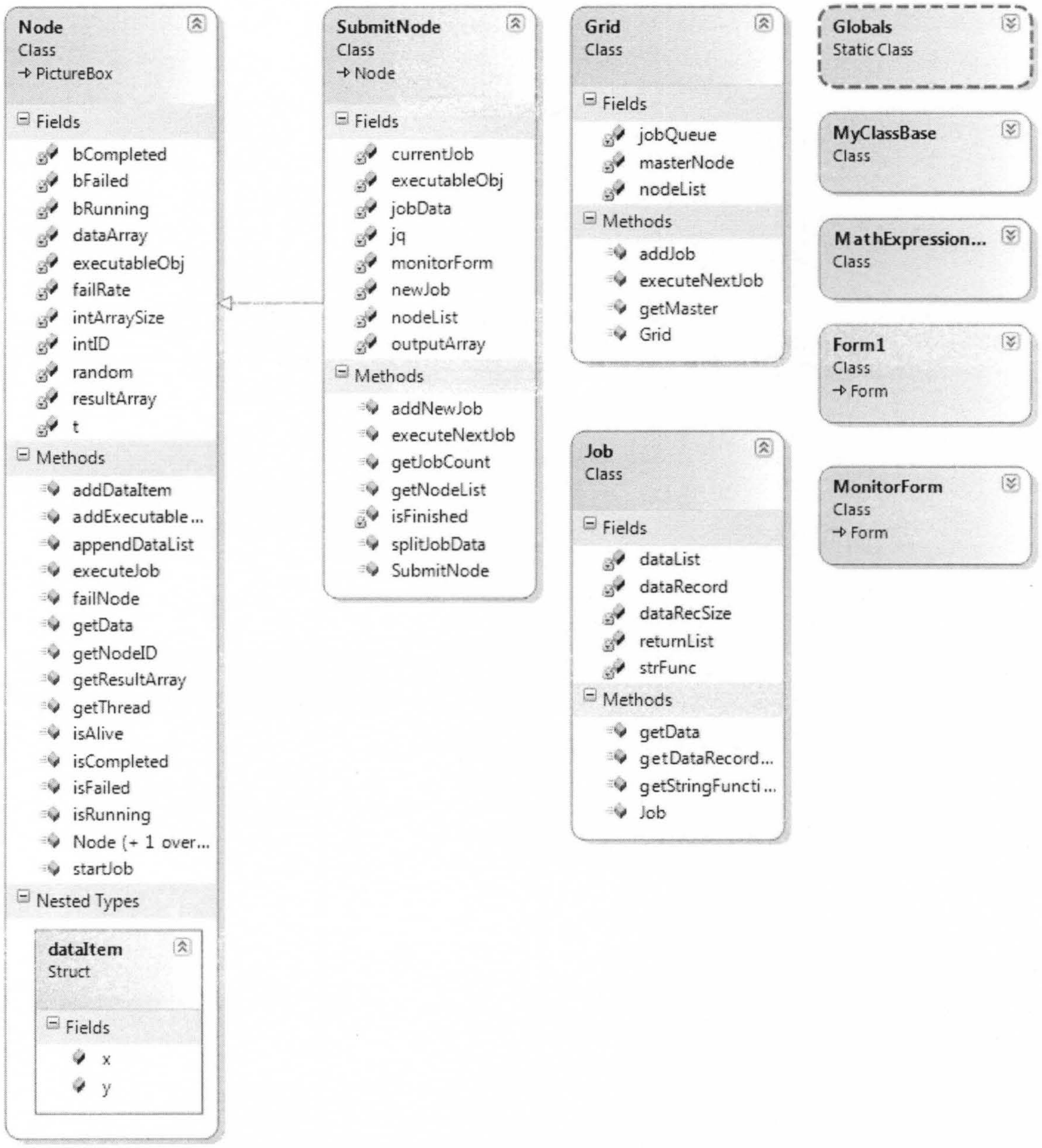


Figure 5.2: The DGSim class diagram

If the failure rate is set at a minimum of 0 to a maximum of 1 with a step size of .01 with 10 repetitions per failure rate, the software will run a grid with a failure rate of 0 for 10 times and then a failure rate of .01 ten times, and a failure rate of .02, 10 times and so on. The current failure rate is shown as the software steps through the process.

The simulator code has been written to do no replication, simple mirroring, reverse mirroring and multiple replicants of data/task pairs across the nodes depending on the permutation selected in the GUI. The complete code is included in Appendix A.

The general user case for the DGSim software is outlined as follows:

- The grid class is called to initialize the grid to the appropriate size—nodes are listed to a C# Array List and a master node is added. See
- The node class is called and instantiated for each node added to the grid. Each node contains a thread. Each node contains a set of random numbers.
- A submit node is instantiated with a new job queue, monitorForm and output array and set equal to the master node in the grid class. (The monitor class, primarily a graphical function, wasn't needed during the simulation runs.)
- The addjob function is called and instantiates a new job using the job class. The job class includes a data size, record size, a program itself and data.
- The submit node's splitJobData function reads the type of permutation of data needed and distributes it appropriately among the instantiated computational nodes.
- The submit nodes calls helper classes that compile the code and the executable object is given to each node. The executable is a c# program in the form of a string. It is compiled on the fly using reflection emit.
- The threads (nodes) are executed and the submit node waits until the threads finish with a status of failed or completed. (The monitor class can be invoked here which shows the thread function in real time graphically.)
- Results are returned and parsed. The salient information about the job including the number of hosts, number of data items, failure rate, whether it failed or completed and on what timestep it failed among other data is all returned to a log file.
- The process is repeated for the next repetition of the same timestep or perhaps for the next timestep.

Some of the actual work in determining whether the job fails is included in the test job code. At each timestep generation of a random number determines whether the run continues or aborts.

5.2 Failure in DGSimulator

The failure rate for each node in the simulated grid is assumed to be the same as any other. In other words there is no a priori knowledge of how a particular host will fail. The failure of individual hosts is assumed to be uniform and random over the makespan of a particular job. The hosts in a grid are either available or not available at any particular time. Once a node is unavailable it is assumed to be unavailable for the remainder of the job. Because of the "yes" or "no" nature of availability of a particular host in a particular timestep it seems to be appropriate to model the failure rate, the rate at which hosts become unavailable, as a Bernoulli process, a series of Bernoulli trials, where success (p) is equal to a node failure in a particular timestep.

The implication of modeling node failures as individual Bernoulli trials is that the probability of failure doesn't change but remains constant, and the process is exponential; eventually all nodes will fail. This would seem to fly in the face of real-world evidence that the probability of failure changes with respect to the time of day or that a Weibull distribution might be a more accurate representation of failure depending on the perameters used. (A complete discussion is included in Appendix C.) The intention, however, is to test all methods for reliability, including the RPP paradigm and any variants of it, under controlled conditions of node failure. The purpose of this particular research was not to develop a precise model of desktop grid usage.

Previous research has shown that task length is another issue of importance in simulating the effects of host failure, and that host failures are essentially independent. Failure results when a task fails to complete on time. The longer the task, the greater the probability that a node will become unavailable during task execution. Variation of task length is of interest in determining the effectiveness of RPP but does not influence design of the basic model. As discussed in Chapter 3, in the special case where the probability of failure of each component in the series, F_i, is the same for n components, the probability of system failure is given by

$$(5.1) \quad F=1-(1-F_i)^n$$

Equation 5.1 is derived from the fact that the overall probability of failure in a series system is one minus the probability of the product of success of each step. Because failure at each timestep is considered independently as a Bernoulli trial we derive the node failure rate for a particular job into a probability of failure at a particular timestep using Equation 5.1 as a basis for Equation 5.2.

$$(5.2) \quad F_i = 1 - \sqrt[n]{1-F}$$

Where F is the assigned failure rate of the node, n is the total number of timesteps and F_i is the failure rate in each timestep.

After a failure rate has been assigned to the individual node, the failure rate for each timestep is determined as shown in Equation 5.2. Because different data and task distribution paradigms require a different number of timesteps some set value of n must be used when comparing different permutations of data. The value chosen is arbitrary and affects the failure rate at each timestep. For the simulator, n is set by taking the number of timesteps required for a simple mirror of the data across all of the nodes. Essentially it is twice the number of data items divided by the number of nodes.

In terms of implementation the node failure rate is set on the program GUI indicating for example that any particular node has a .2 or 20 percent chance of failure over the length of the job. The failure rate for a particular timestep is calculated using Equation 5.2. At each timestep t_i, the next integer r in a series of random integers is generated over some interval n corresponding to the probability of failure in the individual timestep using the C# Random.Next function. If r is equal to n/2 the code returns The importance of using a reliable and appropriate algorithm for generating pseudorandom numbers can't be overstated as Park and Miller point out in their 1988 paper "Random Number Generators: good ones are hard to find." [117] Microsoft's C# random class is based on Donald Knuth's subtractive random number generator algorithm. [130] Knuth points out that the important factor in this algorithm is generating an appropriate seed. [131] By default, Microsoft's Random class uses a seed value from the system clock. Values will not be repeated so long as sufficient time is allowed between initializations, which the grid application does.

5.3 Experimental Procedure and Results

After verification of the simulator output and failure rates, the simulator was set up to instantiate a grid of 100 nodes processing 10,000 data items. Each of six different data arrangements was processed at each probability of failure from 0 to 1 stepping by .01. Each simulation was conducted ten times and the mean job completion rate gathered along with the mean number of timesteps to completion for the ten runs. The following input data permutations were simulated:

- Perm1 – The data, d0, d1, ..., d9999, was distributed across the 100 nodes with no redundancy with 1000 data items per node.
- Perm2 – The data was mirrored or duplicated on the nodes with half of the nodes getting half of the data.
- Perm3 – The data was mirrored with the mirrored permutations reversed in order as explained previously in the RPP heuristic.
- Perm4 – The RPP arrangement of data was implemented with Perm3 followed by a Latin Square data arrangement.
- Perm5_5 – Five replicas of the data were duplicated across the nodes.
- Perm5_10 – Ten replicas of the data were duplicated across the nodes.
- Perm6 – Data was arranged across the nodes in a Latin Square arrangement.

There are some caveats that should be taken into consideration; data is assumed to be perfectly checkpointed to the master node so that all data processed up until the time of a node failure is assumed to be available. Also the type of failure is assumed to be of no importance with emphasis given to grids with a large amount of volatility caused by human interaction. All nodes in a grid are allowed to fail where a design more closely aligned with the real world would allow for job restarts and nodes which rarely fail. The Latin Square design would eventually complete with one node operable, albeit at a very large makespan.

Figures 5.3 and 5.4 show job completion and makespan respectively. Figure 5.3 shows the mean job completion rate for each of the six data permutations as it relates to node failure rate.

As can be seen from Figure 5.3, jobs continue to complete when using RPP, even with node failures in the range of .18 to .20 probability. Job completion when using simple replication, Perm 5_5 and Perm 5_10 is much less, dropping off when the probability of node failure reaches 0.08 to 0.12. In Figure 5.8, makespan is shown in terms of timesteps to job completion or until the last remaining node processes its final piece of data in the case of incomplete jobs.

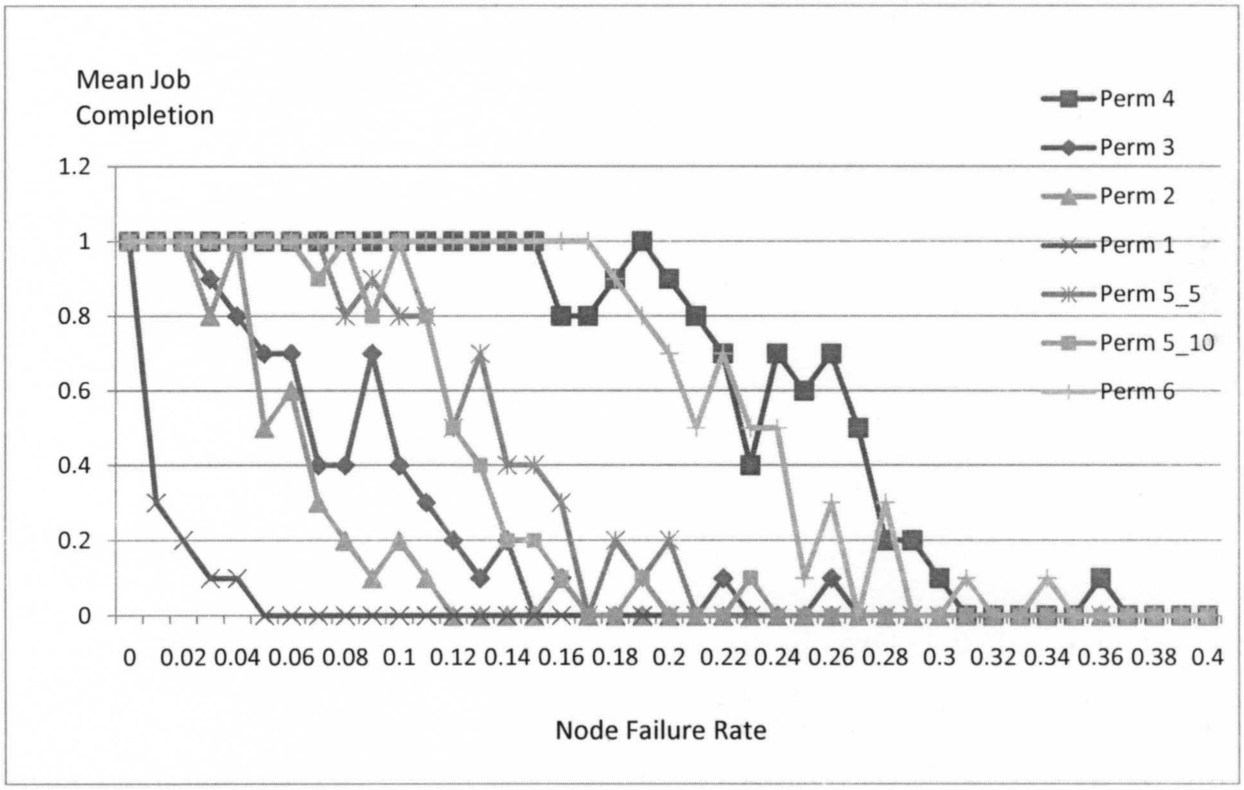


Figure 5.3: The chart shows decreasing job completion as node failures increase.

As expected because of the exponential nature of the node failure model, as discussed previously, all nodes failed and computation ended when the individual node failure rate reached .37 (at least for the large makespan of the Latin Square design). From that point and above, no job could complete before all of the nodes in the grid failed.

Perhaps the two most important results were those predicted by the RPP model. As expected, reverse mirroring had the smallest makespan in the face of lower node failure rates, and the Latin Square design had the highest mean rate of job completion with increasing node failure rates. Of course, as node failures become large the makespan required by the Latin Square design to complete the job increases, which is what occurs at an individual node failure rate of about 37 percent. An interesting result is that the makespan using Perm 6, the pure Latin Square design, is nearly as efficient as the reverse mirroring in Perm3.

It is important to full understanding to note that the reasons for job failure differ between the permutations that include Latin Square designs and those that don't. The RPP, which includes the Latin Square design, fails at large runtimes when all of the nodes finally fail. The other designs fail when all of the replicas of a particular task fail. Figure 5.4 shows the average makespan of the various permutations as they relate to failure rate.

The makespan is shown in timesteps where the timestep t_i is the timestep at which the job completed, meaning all data had been processed or that the final data item was processed whether the job was completed or not. Note that the makespans of the RPP model vary considerably with node failure rates while the others do not.

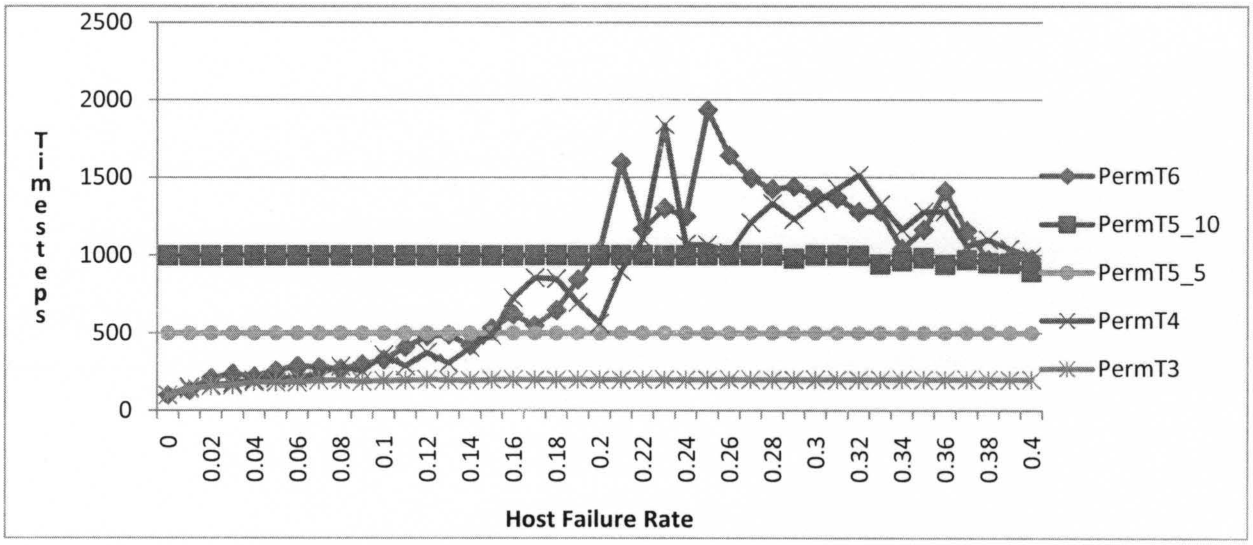


Figure 5.4: Total timesteps, left, required by each permutation are shown at various node failure rates.

5.4 Simulation Conclusions

The results from the previous section indicate that the RPP paradigm can result in enhanced grid reliability. More specifically, the RPP approach:

- Results in smaller makespan than traditional course-grained replication in the absence of failed hosts.
- Result in increased job completion rates under most failure conditions.
- Matches the performance of mirroring in the worst failure case.
- Completes so long as one functioning processor remains.

One other interesting observation was obtained. RPP includes two basic features -- reverse mirroring and the Latin Square design. Both showed essentially the same performance at lower failure rates. Reverse mirroring results in job failure at high node failure rates, however, while the Latin Square distribution paradigm does not.

Although the work is aimed toward providing a grid application programming paradigm for reliability, the work also could have implications for the design of grid middleware and other system software for grid computing. A potential issue with regard to RPP is the amount of data that may be transferred, n^2 where n is the number of data items. When the node failure rate is low, most of the data need never be traversed and hence transferred in the first place. Jobs complete in a small number of timesteps under reliable conditions. So a "lazy" or delayed mode of data transfer is preferable.

CHAPTER 6

LATIN SQUARE DISTRIBUTION AND PHARMACOPHORE DISCOVERY

In this case study we show that Inductive Logic Programming (ILP) algorithms for pharmacophore discovery run efficiently and reliably on a grid of desktop computers using a Latin Square distribution of heterogeneous data and homogeneous tasks. Pharmacophore discovery was chosen as the subject of a case study involving data distribution for reliability primarily because it was a motivating factor in the initial research. The case study is not meant to duplicate results of the simulator in a live grid situation, although doing so is a consideration for future research. It is possible to compare results of the case study and simulator results, however, which is discussed in Chapter 7.

6.1 Previous Results with Inductive Logic Programming

In 2006 work at the University of Louisville showed that inductive logic programming (ILP) algorithms for pharmacophore discovery can run efficiently on a grid of inexpensive computers.[6] Designing a new drug is a long, tedious, and very expensive process that can take many years to complete. Machine learning techniques and ILP algorithms have been shown to be valuable aids in speeding discovery of candidate molecular structures. The 2006 paper described a case study utilizing structure activity relationships and ILP for pharmacophore discovery on an "Xgrid" of Apple G4 computers available at high schools in Kentucky. With this architecture, an algorithm that requires about nine hours on a single processor, and a little less time on an older tightly-coupled cluster computer, executed in 32 minutes using the donated idle cycles of a large number of loosely-coupled processors in a computational grid. Unfortunately processing of the job required manual resubmission of subtasks because of host failures in the grid. In the current research, use of a Latin Square data distribution paradigm allowed the job to complete in a couple of minutes, even in the face of large numbers of host failures.

The concept of the pharmacophore model is key to the search for new and interesting medicinal drugs. A pharmacophore is a set of structural features in a molecule that acts on some target molecule to produce biological activity [118]. Many molecules may share these structural features and hence might show some of the same biological activity. The search for a pharmacophore involves finding a group of several mixtures of molecules that have some activity and determining the common features that make them active – in other words, finding the pharmacophore. Inductive logic programming (ILP) provides one method for finding new pharmacophores. In particular ILP provides a method for looking at a group of active compounds and a group of inactive ones and discovering pharmacophores, those structural attributes that might make a compound active and another inactive. [119]

Understanding current methods for pharmacophore discovery, and parallel ILP in particular, depends on the understanding of a couple of basic underlying concepts that often are misused -- even in the medical literature. The first such concept is that of a structure activity relationship or SAR. [119] A SAR is a set of mathematical relationships linking chemical structure and pharmacological activity for a set of compounds. A pharmacophore is a set of features that provide optimal activity on some biological target. Generally some set of SAR is searched for the pharmacophore. So, by examining the SAR of a set of compounds that are biologically active in a desired area, it is possible in some fashion or other to discover one or more pharmacophores, or sets of features that provide optimal activity. Using a pharmacophore, one might determine other compounds that would be likely candidates for consideration as a drug. ILP is a particularly interesting candidate for examining SAR and discovering the interesting pharmacophores to which their activity adheres.

Much previous work has been done using ILP and structure activity relationships for pharmacophore discovery by Michael Sternberg and Stephen Muggleton, among others. Their 2003 paper in QSAR and Combinatorial Science, "Structure Activity Relationships (SAR) and Pharmacophore Discovery Using Inductive Logic Programming," provides an excellent overview of ILP as well as other major methodologies for pharmacophore discovery.[119] In general, ILP involves listing positive examples, negative examples and background knowledge. The background knowledge is typically some set of features or attributes of the set of positive and negative examples. The combinations of features that provide the best logical cover of the positive examples and the least cover of the negative example is a hypothesis. In the case of pharmacophore search, a set of biologically active compounds provides the positive examples, some set of negative compounds, the negative examples and some set of interesting features, the background knowledge. The best cover for the active compounds over the feature space and the least cover of the negative compounds over the same space provide a hypothesis regarding the set of features that make up a pharmacophore.

It also is possible to run such ILP in parallel, supplying an almost linear speedup to the computationally intensive search. Researchers at the University of Louisville, in concert with others, have succeeded in implementing ILP for pharmacophore discovery in a tightly-coupled, distributed memory parallel environment. In 1999, these researchers used an initial data set consisting of 48 mixtures of pseudopeptides synthesized by modified solid phase methods and cleaved from the polystyrene matrix. Each mixture had a recorded level of activity against Pseudomonas Aeruginosa bacterium and consisted of eight compounds. [120] The goal was to find the largest three dimensional substructure present in at least one member of every active mixture and not present in any member of any inactive mixture.

Curtis, Page, Graham and Spatola conducted the first set of experiments on a SUN Ultra computer with a run-time of about two weeks. [121] A second set of experiments was conducted by Wild on a Beowulf cluster with 8 processors [120] with a runtime of about two days. The same job was implemented on a 112 processor IBM RS6000 SP2 supercomputer in 2002, executing in about 2.3 hours. [122] In 2006, the work was moved from such clusters to a unique and very loosely coupled grid of widely disbursed Apple computers located at various high schools across the Commonwealth of

Kentucky. [6] A runtime of about nine hours on a single Apple computer was reduced to about 30 minutes of actual execution time on the grid. However, because some tasks weren't completed they had to be resubmitted, as mentioned previously.

The next section presents an overview of the hardware and software configuration of the grid. Section 6.3 presents experimental in terms of reliability and execution time. Section 6.4 presents conclusions and directions for future research.

6.2 Grid ILP Using Latin Square Distribution

The intent of the current study is somewhat different from that of the previous 2006 effort. The intent of the earlier study was to duplicate previous ILP searches with Pseudomonas Aeruginosa running on tightly coupled clusters in order to obtain timing data and to test the feasibility of running an ILP pharmacophore search on a loosely-coupled grid. Various Perl scripts were used to submit the jobs through the Xgrid software. The job consisted of a simple Bash shell script that accepted various conformations as command line arguments. The script then called the Prolog ILP program, which reads the environmental variable into a list of conformations to be searched.

In the current study, emphasis has been placed on the reliability of the grid job in the face of failures of the individual subtasks while using RPP (a Latin Square task and data distribution paradigm) on a Condor grid at the Dahlem Supercomputer Laboratory at the Speed Engineering School, University of Louisville. In the 2010 case study, jobs were intentionally stopped to test the efficacy of the Latin Square data distribution scheme in providing reliability while reducing makespan.

Various Windows batch scripts were used to submit the jobs through Condor software. The job consisted of a simple Windows command file script that accepted various conformations as command line arguments. The script then called the Prolog ILP program, which reads the environmental variable into a list of conformations to be searched. As previously the serial code was divided so that each instance of the program, rather than considering all conformations of a particular seed molecule, only considered the seed molecule and some subset of total conformations for that molecule. Each instance was given all of the data in the beginning so there was no communication until the output of the code was returned at the end of the run in the form of a distinct file.

Because SWI-Prolog was not available on machines in the grid, the program was compiled on a development machine, and the Prolog executable was then bundled with everything needed to run as a completely standalone executable including some dynamic linking libraries not available on the grid machines. The program along with the script had to be sent to each machine on the grid where the software was to run.

6.2.1 Hardware and Software

The grid hardware consisted mainly of host desktop computers located in the Dahlem Supercomputer Lab at the Speed Engineering School, University of Louisville. Approximately 50 computers were used, although the available number varied. Most of the jobs were run in the evening and nighttime hours to control the number of desktop users and attempt (successfully) to add obtain some consistency in the computer pool.

Many of the grid computational hosts are Dell systems with three gigahertz, dual core, 64-bit Intel processors and 4 GB of internal RAM. The systems were running 32-bit Windows Vista. Although the grid comprises other systems as well, the grid software was used to restrict computations to the aforementioned systems.

The grid itself is constructed using Condor software. [7;123] Full descriptions of Condor are available many places including the software website [124]. Briefly however, a Condor master machine schedules jobs on a computational hosts and provides scheduling and queuing functionality. Jobs are submitted to the queue through submit machines with the use of job control files called description files. A variety of commands are provided to submit, delete and view jobs among other functions. Individual tasks are submitted to available processors and the results returned to the submit machine at the completion of the job.

Here, for example, is a sample condor description file for the Latin Square data distribution job:

```
Executable = D:\workingCondorCode\50_systems\h1.bat
Universe = vanilla
output = D:\workingCondorCode\50_systems\h1.out
error = D:\workingCondorCode\50_systems\h1.err
Log = D:\workingCondorCode\50_systems\h1.log
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files       =       D:\workingCondorCode\serial3.exe,
D:\workingCondorCode\50_systems\h1.bat,
D:\workingCondorCode\libpl.dll,
D:\workingCondorCode\pthreadVC.dll,
D:\workingCondorCode\callSerial.bat
Requirements   = (OpSys == "WINNT60" && Arch == "INTEL"
&& CAE_LAB=?= True)
Queue
```

The Condor description files indicate the type of Condor job, where outputs should go as well as the required input files. Several files are required to run the RPP or Latin Square distribution job, including the Prolog program itself – serial3.exe, two dynamic linking libraries required because Prolog is actually transferred with the program, and two windows batch files. Prolog is not installed on the computational hosts so the Prolog engine and necessary dynamic linking libraries have to be sent along to each machine.

Because of the large number of jobs to be sent and the requirements of a Latin Square configuration – 50 jobs must be sent to each of 50 machines – the task would be difficult to manage manually. Job submission for the case study is actually accomplished using a small C# console program, Latin Grid, which writes the batch and description files necessary to submit the jobs, as shown in Appendix B.

The code produces two batch files and a job description file for each of the 50 hosts and then submits the jobs in either a Latin Square or mirrored configuration. One set of batch files is named by host, such as h2.bat and the second set is simply callSerial.bat.

Here are examples of each:

H2.BAT

```
FOR %%a IN (  c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15
c16 c17 c18 c19 c20 c21 c22 c23 c24 c25 c26 c27 c28 c29 c30 c31
c32 c33 c34 c35 c36 c37 c38 c39 c40 c41 c42 c43 c44 c45 c46 c47
c48 c49 c50 c1 ) DO call callSerial.bat %%a
```

callSerial.bat:

```
set serialEnv=%1
echo %serialEnv%
call serial3
```

Notice that the h2.bat file calls callSerial.bat with the name of each conformation. CallSerial then sets the conformation in an environment variable on the execution host and then calls the actual Prolog executable, serial3.exe, where the environment variable is read and processed.

Serial3, which contains the prolog code itself, remains the same and was simply recompiled to Windows instead of Apple's operating system.

Data returned from each host included a list of conformations processed and the amount of time required to process each conformation. Output data was concatenated into one file and then sorted by timestep. After all of the runs, another small C# program processed the output, determining the timestep in which all of the data had been processed, and produced a log file showing the timestep in which the job was completed.

6.2.2 Experimental Model and Results

The purpose of this case study was to determine the feasibility of running an actual job on a grid of computers using the Latin Square data and task distribution portion of RPP. In addition live data was used to compare the performance of simple duplication of tasks and data (simple mirroring) to Latin Square redundancy.

For the Latin Square case study, 50 conformations were sent for processing on each of 50 processors along with all necessary data. Wall time processing for each conformation varied between approximately 57 and 60 seconds, roughly three times faster than the previous grid experiments. Wall time is not, however, the important factor in this study. Rather, the behavior of the job under conditions of failure is most important. In the previous research, node failures produced severe outliers and missing results that caused jobs to be manually resubmitted the following day.

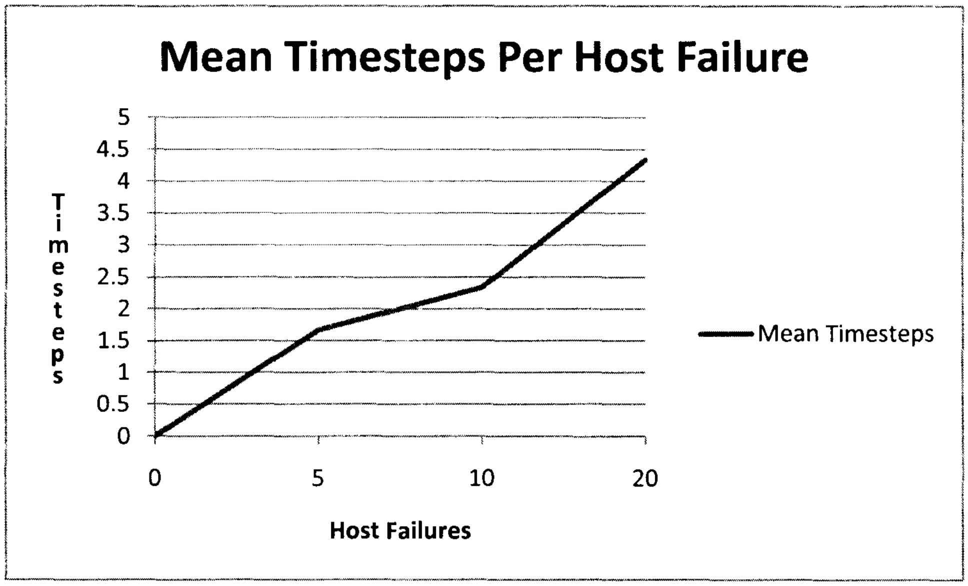


Figure 6.1 shows the average timestep in which a job finished in the face of 5, 10 and 20 failures on a 50 processor grid.

Because work comparing various data distribution schemes using the Grid Simulation software indicated that the Latin Square data distribution was effective in the face of node failure (as discussed in Chapter 5), and because of the features of the Latin

Square distribution in proof of maximum makespan in the face of hardware failure (as discussed in Chapter 4), the 50 conformations were submitted to the processors in a Latin Square configuration.

Experimental algorithm:

- For example, a job "h1" consisting of conformations {C1, C2, ..., C50} was sent as a task to one processor. Another job, "h2," consisting of conformations { C2, C3, ...,C50, C1} was sent to another, and so on until a set of jobs {h1, h2, ..., h50} had been submitted.
- A set of pseudo-random numbers was generated – either 0, 5, 10, or 20 such numbers in the range of 1 to 50.
- In the first three runs no hosts were killed. In the next three runs, five jobs were killed using the condor remove command and using the five previously generated random numbers. Each number was matched with a task name so that a random 25, for example, cause job h25 to be killed. In the next set of three runs 10 tasks were killed, and in the final set of runs, 20 tasks were killed.
- Processing one conformation was considered one timestep. Information about the makespan of each of the nine jobs was collected.

There are a couple of caveats. All failures were immediate and no data was returned from a failed task, as though the host had failed on startup. In addition, partial data was not collected. All hosts computed all conformations and then the data was analyzed to determine by what timestep job completion actually occurred.

Figure 6.1 shows the mean timestep in which all 50 conformations were completed in the face of 0, 5, 10 and 20 failed tasks (host failures). When no hosts failed the job was completed in Timestep 0 or about 58 seconds. When five hosts failed the jobs complete at an average timestep 1.6 as well as at average timestep 2.33 for 10 failures and 4.33 for 20 failures.

The results are encouraging when one considers that the job completed at an average of timestep 4.33 when about 40 percent of the grid failed.

Although the purpose of the case study was to determine the feasibility of running tasks and data in Latin Square fashion across a grid, data about what might happen under similar failure conditions using simple mirroring provides a nice comparison of the two methods.

In the case of simple mirroring, one copy of the 50 conformations was placed on 25 processors and another 50 on another 25 processors. The jobs, each two conformations long, required about two minutes of processor time to complete. Because the jobs completed so quickly it was difficult to halt them prior to completion. Instead, random numbers corresponding to hosts were generated, indicating that that particular host failed. The data resulting from that host was then removed from the output as though the host had suffered failure on startup or infant mortality.

Three such "runs" were conducted for each of 5, 10 and 20 host failures. The resulting data was analyzed, and if two hosts running the same mirrored data failed then the job was determined to have failed.

Of the nine runs, two succeeded, one job with 10 host failures and a job with five host failures. The remaining attempts to complete the job failed.

The intention of the case study was to show that the proposed RPP paradigm, at least in the form of Latin Square data distribution, may easily be applied to an actual job running on a grid. In addition it is important to observe the behavior of the job on host processors in the face of real failures on the grid. Of course the failures were purposely generated in random fashion, but the results are interesting none the less. The main conclusion is that the Latin Square data and task distribution is feasible and affords robust protection in the face of failures.

CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

The purpose of this research was to examine the use of permutation with redundancy as a method of improving reliability in computational grid applications. Three primary avenues of exploration were delineated early on and have been accomplished – development of a model of grid data and task redundancy, development of grid simulation software and testing Replication and Permutation Paradigm (RPP) against other methods of fault tolerance through redundancy and finally running a program on a live grid using RPP. This chapter presents the conclusions from this research and an overview of further research directions.

7.1 Conclusions

Each of the research areas has produced important results. The redundancy model provided tools to analyze redundancy in a logical and somewhat rigorous fashion. Using the model allowed development of two theorems and subsequent proof by mathematical induction regarding Latin Squares. Interesting in their own right, the theorems have implications for redundancy. Basically the theorems describe the changing position of symbols between the rows of a standard Latin Square. When a symbol is missing because a column is removed the theorems provide a basis for determining the next row and column of the missing symbol. In terms of the redundancy model this allows one to state the maximum makespan in the face of missing computational hosts when using Latin Square redundancy. Maximum makespan is important because it can provide an indication of when a computation should stop, saving valuable computational resources. In addition, grid failure rates are directly related to job length, and predicting job length is important in and of itself.

The DGSimulator software was developed and used to compare six different data and task distribution schemes on a simulated grid. The software clearly showed the advantage of running RPP, including reverse mirroring and/or the Latin Square distribution methods. Both resulted in faster completion times in the face of computational host failures. The Latin Square method also fails gracefully in that jobs complete with massive node failure while increasing makespan. The major caveat involved with the Latin Square method is that a large amount of data must be transferred. So a delayed or "lazy" data transfer method needs to be examined along with various methods of determining when a job has completed so that data transfer may be stopped.

Finally inductive logic programming was used to implement pharmacophore search on a Condor grid.in the Dahlem Lab at the University of Louisville Speed School of Engineering. The primary purpose was to examine the behavior of Latin Square distribution on a "live" grid running a computationally intensive job. Latin Square distribution was chosen because it offers the most promise in terms of reliability as indicated by the results of the simulator runs. The results were encouraging. All jobs completed, even in the face of large numbers of randomly generated computational host failures. In addition the live results comfirm the general result of the simulation in the sense that makespan increases slowly in the face of increasingly large numbers of node failures. Even with 40 percent of the live grid failing, the number of timesteps required increased from one to five of 50 possible timesteps. The simulator results show a similar increase in makespan in the face of node failure.

The main conclusion is that RPP, including Latin Square data and task distribution and Reverse Mirroring, is feasible and affords robust protection in the face of failures.

7.2 Future Directions

Many interesting opportunities remain for research into Latin Square and other types of redundancy for fault tolerance. There are, of course, unanswered questions remaining, which are discussed briefly in the following paragraphs.

First, how would Latin Square replication perform against other types of redundancy on a live grid with large numbers of actual users? Although RPP in the form of Latin Square redundancy worked well in the face of node failure on a live grid, other forms of redundancy, including reverse mirroring, were not implemented. A study comparing results on an actual grid might produce interesting results. Additional work comparing the results of the simulator to a comparable job on the live grid would provide additional validation for the simulator enhancing its usefulness in more thorough future analysis of grid systems.

Secondly, one could consider how well does Latin Square replication scale? The number of replicants in Latin Square replication is n^2 where n is the number of computational hosts. Obviously some method of reducing bandwidth and memory concerns is necessary. In very large pools of hosts the job length would be greater because of the way makespan can grow when using Latin Squares. Is there an upper limit to the redundancy before job length becomes so great that all processors fail before the job is completed? Such a scenario might not be a likely outcome but should be investigated.

A final issue is how does one estimate job completion? It can be difficult to determine when all of the data has been processed and all results reported on a loosely coupled system. One nice result of this research is the ability to estimate maximum makespan in the face of computational host failures. Exactly how to use that information to determine when to stop processing is an interesting question.

Although work remains to be done in the area of redundancy and over-provisioning for reliability, RPP provides valuable insight and a methodology for making computational grids, the largest computers in existence today, even better tools for complex calculations.

REFERENCES

[1] M. Baker, R. Buyya, and D. Laforenza, "Grids and Grid technologies for wide-area distributed computing," *Software-Practice and Experience*, vol. 32, no. 15, pp. 1437-1466, 2002.

[2] I. Foster, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Euro-Par 2001 Parallel Processing: 7th International Euro-Par Conference, Manchester, UK, August 28-31, 2001: Proceedings*, 2001.

[3] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer, "BEOWULF: A parallel workstation for scientific computation," *Proceedings of the 24th International Conference on Parallel Processing*, vol. 1, pp. 11-14, 1995.

[4] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50-55, 2008.

[5] S. Jha, A. Merzky, and G. Fox, "Using clouds to provide grids higher-levels of abstraction and explicit support for usage modes," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 8, pp. 1087-1108, 2009.

[6] N. P. Johnson and J. H. Graham, "An Application of Grid Computing to Pharmacophore Discovery Using Inductive Logic Programming," 2006, pp. 418-423.

[7] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor-a hunter of idle workstations," *Distributed Computing Systems, 1988. , 8th International Conference on*, pp. 104-111, 1988.

[8] I. Foster and C. Kesselman, *The grid: blueprint for a new computing infrastructure* Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1998.

[9] A. Braverman, "Father of the Grid," *The University of Chicago Magazine*, 2007.

[10] I. Foster and C. Kesselman, "Globus: a Metacomputing Infrastructure Toolkit," *International Journal of High Performance Computing Applications*, vol. 11, no. 2, p. 115, 1997.

[11] I. Foster, "What is the Grid? A Three Point Checklist," *Grid Today*, vol. 1, no. 6, pp. 22-25, 2002.

[12] M. L. Bote-Lorenzo, Y. A. Dimitriadis, and E. Gomez-Sanchez, "Grid Characteristics and Uses: A Grid Definition," *Grid Computing: First European Across Grids Conference, Santiago de Compostela, Spain, February 13-14, 2003: Revised Papers*, 2004.

[13] A. Grimshaw, "What Is A Grid?," *Grid Today*, 2002.

[14] H. Stockinger, "Defining the grid: a snapshot on the current view," *The Journal of Supercomputing*, vol. 42, no. 1, pp. 3-17, 2007.

[15] "CoreGRID," 2008.

[16] I. Foster, "The Grid: A new infrastructure for 21st century science," *Grid Computing: Making the Global Infrastructure a Reality*, 2003.

[17] P. ASADZADEH, R. Buyya, C. L. KEI, D. NAYAR, and S. Venugopal, "Global Grids and Software Toolkits: AStudy of Four Grid Middleware Technologies," *High Performance Computing: Paradigm and Infrastructure*, 2006.

[18] M. Baker, R. Buyya, and D. Laforenza, "The Grid: International Efforts in Global Computing," *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, 2000.

[19] "Open Grid Forum," 2008.

[20] "EGEE," 2008.

[21] R. Bolze, F. Cappello, E. Caron, M. Daydθ, F. Desprez, E. Jeannot, Y. Jθgou, S. Lanteri, J. Leduc, and N. Melab, "Grid'5000: a large scale and highly reconfigurable experimental Grid testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, p. 481, 2006.

[22] J. Yu and R. Buyya, "A Taxonomy of Workflow Management Systems for Grid Computing," *Journal of Grid Computing*, vol. 3, no. 3, pp. 171-200, 2005.

[23] R. B. M. M. Klaus Krauter, "A taxonomy and survey of grid resource management systems for distributed computing," *Software: Practice and Experience*, vol. 32, no. 2, pp. 135-164, 2002.

[24] M. de Assuncao, K. Nadiminti, S. Venugopal, T. Ma, and R. Buyya, "An Integration of Global and Enterprise Grid Computing: Gridbus Broker and Xgrid Perspective," *Grid and Cooperative Computing-Gcc 2005: 4th International Conference, Beijing, China, November 30--December 3, 2005, Proceedings*, 2005.

[25] "TeraGrid," 2008.

[26] F. Berman, G. Fox, and T. Hey, "The Grid: past, present, future," *Grid Computing: Making the Global Infrastructure a Reality*, p. 12, 2003.

[27] D. Nussbaum and A. Agarwal, "Scalability of parallel machines," *Communications of the ACM*, vol. 34, no. 3, pp. 57-61, 1991.

[28] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *AFIPS Conference Proceedings*, vol. 30, no. 8, pp. 483-485, 1967.

[29] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532-533, 1988.

[30] D. De Roure, M. A. Baker, N. R. Jennings, and N. R. Shadbolt, "The evolution of the Grid," *Grid Computing: Making the Global Infrastructure a Reality*, 2003.

[31] P. Lyster, L. Bergman, P. Li, D. Stanfill, B. Crippe, R. Blom, C. Pardo, and D. Okaya, "CASA gigabit supercomputing network: CALCRUST three-dimensional real-time multi-dataset rendering," *Proc. Supercomputing*, vol. 92 1992.

[32] T. DeFanti, I. Foster, M. Papka, R. Stevens, and T. Kuhfuss, "Overview of the I-WAY: Wide area visual supercomputing," *International Journal of Supercomputer Applications*, vol. 10, no. 2, pp. 123-130, 1996.

[33] L. Smarr and C. E. Catlett, "Metacomputing," *Communications of the ACM*, vol. 35, pp. 44-52, 1992.

[34] J. Herrera, E. Huedo, R. S. Montero, and I. M. Llorente, "Execution of Typical Scientific Applications on Globus-Based Grids," *Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar'04)-Volume 00*, pp. 177-183, 2004.

[35] A. S. Grimshaw and W. A. Wulf, "The Legion vision of a worldwide virtual computer," *Communications of the ACM*, vol. 40, no. 1, pp. 39-45, 1997.

[36] M. Romberg, "The UNICORE architecture: seamless access to distributed resources," *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pp. 287-293, 1999.

[37] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," *Open Grid Service Infrastructure WG, Global Grid Forum, June*, vol. 22, p. 2002, 2002.

[38] L. Childers, T. Disz, R. Olson, M. E. Papka, R. Stevens, and T. Udeshi, "Access Grid: Immersive Group-to-Group Collaborative Visualization," *Proc. 4th International Immersive Projection Technology Workshop*, 2000.

[39] J. F. Shoch and J. A. Hupp, "The Worm Programs: Early experience with a distributed computation," *Communications of the ACM*, vol. 25, no. 3, pp. 172-180, 1982.

[40] N. T. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface," *Arxiv preprint cs. DC/0206040*, 2002.

[41] R. Wolski, D. Nurmi, J. Brevik, H. Casanova, and A. Chien, "Models and Modeling Infrastructures for Global Computational Platforms," *Workshop on Next Generation Software, IPDPS, April*, 2005.

[42] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," *5th IEEE/ACM International Workshop on Grid Computing*, pp. 365-372, 2004.

[43] Woltman and Kurowski, "Great Internet Mersenne Prime Search," 2008.

[44] "distributed.net," 2008.

[45] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@ home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56-61, 2002.

[46] D. Anderson, "Public Computing: Reconnecting People to Science," *Conference on Shared Knowledge and the Web*, pp. 17-19, 2003.

[47] D. Stainforth, J. Kettleborough, A. Martin, A. Simpson, R. Gillis, A. Akkas, R. Gault, M. Collins, D. Gavaghan, and M. Allen, "Climateprediction.net: Design Principles for Public-Resource Modeling Research," *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing Systems*, 2002.

[48] "LHC@Home," 2008.

[49] "Predictor@home," 2008.

[50] K. Pearson, 2008.

[51] J. Bohannon, "DISTRIBUTED COMPUTING: Grassroots Supercomputing," *Science*, vol. 308, no. 5723, p. 810, 2005.

[52] G. Fedak, C. Germain, V. Neri, and F. Cappello, "XtremWeb: A Generic Global Computing System," *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRIDÆ01)*, 2001.

[53] A. Chien, B. Calder, S. Elbert, and K. Bhatia, "Entropia: Architecture and Performance of an Enterprise Desktop Grid System," *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 597-610, 2003.

[54] R. Bunduchi, M. Gerst, I. Graham, and R. Williams, "Driving Grid Standardisation-the role of the business community," *Proc. 10th EURAS Workshop*.

[55] D. P. Anderson, E. Korpela, and R. Walton, "High-Performance Task Distribution for Volunteer Computing," *e-Science and Grid Computing, First International Conference on*, pp. 196-203, 2005.

[56] H. K. E. B. SungJin Choi, MaengSoon Baik, SungSuk Kim, ChanYeol Park, and ChongSun Hwang, "Characterizing and Classifying Desktop Grid," *Seventh IEEE International Symposium on Cluster Computing and the Grid*, 2007.

[57] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao, "Cluster Computing on the Fly: P2P Scheduling of Idle Cycles in the Internet," *Peer-To-Peer Systems III: Third International Workshop, IPTPS 2004, La Jolla, CA, USA, February 26-27, 2004: Revised Selected Papers*, 2004.

[58] A. J. Chakravarti, G. Baumgartner, and M. Lauria, "The organic grid: self-organizing computation on a peer-to-peer network," *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, vol. 35, no. 3, pp. 373-384, 2005.

[59] A. Montresor, H. Meling, and O. Babaoglu, "Messor: Load-Balancing through a Swarm of Autonomous Agents," *Agents and Peer-To-Peer Computing: First International Workshop, Ap2PC 2002, Bologna, Italy, July 15, 2002: Revised and Invited Papers*, 2003.

[60] L. Zhong, D. Wen, Z. W. Ming, and Z. Peng, "Paradropper: a general-purpose global computing environment built on peer-to-peer overlay network," *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pp. 954-957, 2003.

[61] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal, "Alchemi: A .NET-Based Enterprise Grid Computing System," *Proceedings of the 6th International Conference on Internet Computing (ICOMP'05)*, pp. 27-30, 2005.

[62] D. Kondo, M. Taufer, C. L. Brooks, H. Casanova, and A. A. Chien, "Characterizing and evaluating desktop grids: an empirical study," *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.

[63] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11-33, 2004.

[64] A. Rueda and M. Pawlak, "Pioneers of the reliability theories of the past 50 years," *Reliability and Maintainability, 2004 Annual Symposium-RAMS*, pp. 102-109, 2004.

[65] Y. S. Dai, M. Xie, and K. L. Poh, "Reliability analysis of grid computing systems," *Dependable Computing, 2002. Proceedings. 2002 Pacific Rim International Symposium on*, pp. 97-104, 2002.

[66] D. Kondo, D. Kondo, G. Fedak, F. Cappello, A. A. Chien, and H. Casanova, "Availability Traces of Enterprise Desktop Grids," *Grid Computing, 7th IEEE/ACM International Conference on*, pp. 301-302, 2006.

[67] D. Kondo, G. Fedak, F. Cappello, A. A. Chien, and H. Casanova, "Characterizing resource availability in enterprise desktop grids," *Future Generation Computer Systems*, vol. 23, no. 7, pp. 888-903, 2007.

[68] D. Kondo, B. Kindarji, G. Fedak, and F. Cappello, "Towards Soft Real-Time Applications on Enterprise Desktop Grids," *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)-Volume 00*, pp. 65-72, 2006.

[69] D. Kondo, G. Fedak, F. Cappello, A. A. Chien, and H. Casanova, "On Resource Volatility in Enterprise Desktop Grids," *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, 2006.

[70] A. K. Sinha, B. Ludaescher, B. Brodaric, C. Baru, D. Seber, A. Snoke, and C. Barnes, "GEON: Developing the Cyberinfrastructure for the Earth Sciences-A Workshop Report on Intrusive Igneous Rocks, Wilson Cycle and Concept Spaces," *Submitted to GSA Today.*

[71] J. Brevik, D. Nurmi, and R. Wolski, "Quantifying machine availability in networked and desktop grid systems," *Proceedings of CCGrid04*, 2004.

[72] S. J. Choi, M. S. Baik, C. S. Hwang, J. M. Gil, and H. C. Yu, "Volunteer availability based fault tolerant scheduling mechanism in desktop grid computing environment," *Network Computing and Applications, 2004. (NCA 2004). Proceedings. Third IEEE International Symposium on*, pp. 366-371.

[73] J. Brevik, D. Nurmi, and R. Wolski, "Automatic methods for predicting machine availability in desktop Grid and peer-to-peer systems," *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pp. 190-199, 2004.

[74] M. A. W. S. CHERN, "On the computational complexity of reliability redundancy allocation in a series system," *Operations research letters*, vol. 11, no. 5, pp. 309-315, 1992.

[75] G. Kandaswamy, A. Mandal, and D. A. Reed, "Fault tolerance and recovery of scientific workflows on computational grids,", 8 ed pp. 777-782.

[76] M. Adler, Y. Gong, and A. L. Rosenberg, "Optimal sharing of bags of tasks in heterogeneous clusters," *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pp. 1-10, 2003.

[77] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauve, F. A. B. Silva, C. O. Barros, and C. Silveira, "Running Bag-of-Tasks applications on computational grids: the MyGrid approach," *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pp. 407-416, 2003.

[78] D. P. da Silva, W. Cirne, and F. V. Brasileiro, "Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids," *Euro-Par 2003 Parallel Processing: 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003: Proceedings*, 2003.

[79] N. Fujimoto and K. Hagihara, "Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid," *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pp. 391-398, 2003.

[80] D. Kondo, A. A. Chien, and H. Casanova, "Scheduling Task Parallel Applications for Rapid Turnaround on Enterprise Desktop Grids," *Journal of Grid Computing*, vol. 5, no. 4, pp. 379-405, 2007.

[81] M. Maheswaran, S. Ali, H. J. Siegal, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasksonto heterogeneous computing systems," *Heterogeneous Computing Workshop, 1999. (HCW'99) Proceedings. Eighth*, pp. 30-44, 1999.

[82] D. E. Bakken and R. D. Schlichting, "Tolerating failures in the bag-of-tasks programming paradigm," *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pp. 248-255, 1991.

[83] W. Cirne, F. Brasileiro, D. Paranhos, L. F. W. Goes, and W. Voorsluys, "On the efficacy, efficiency and emergent behavior of task replication in large distributed systems," *Parallel Computing*, vol. 33, no. 3, pp. 213-234, 2007.

[84] E. LAWLER, J. LENSTRA, A. RINNOOYKAN, and D. SHMOYS, "Sequencing and scheduling: Algorithms and complexity," *Handbooks in OR & MS, volume 4*, 1989.

[85] Y. Zhang, C. Koelbel, and K. Kennedy, "Relative performance of scheduling algorithms in grid environments," Citeseer, 2007.

[86] Y. Zhang, A. Mandal, C. Koelbel, and K. Cooper, "Combined Fault Tolerance and Scheduling Techniques for Workflow Applications on Computational Grids," IEEE Computer Society, 2009, pp. 244-251.

[87] M. O. Neary and P. Cappello, "Advanced eager scheduling for Java-based adaptive parallel computing," *Concurrency and Computation Practice and Experience*, vol. 17, no. 7-8, pp. 797-819, 2005.

[88] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Google, Inc.*, 2004.

[89] M. Canonico, "Scheduling Algorithms for Bag-of-Tasks Applications on Fault-Prone Desktop Grids," *Ph. D. dissertation, University of Turin*, 2006.

[90] N. Fujimoto and K. Hagihara, "A comparison among grid scheduling algorithms for independent coarse-grained tasks," *Applications and the Internet Workshops, 2004. SAINT 2004 Workshops. 2004 International Symposium on*, pp. 674-680, 2004.

[91] G. D. Ghare and S. T. Leutenegger, "Improving Speedup and Response Times by Replicating Parallel Programs on a SNOW," *Job Scheduling Strategies for Parallel Processing: 10th International Workshop, JSSPP 2004, New York, NY, USA, June 13, 2004: Revised Selected Papers*, 2005.

[92] D. Kondo, A. A. Chien, and H. Casanova, "Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids," *Proceedings of the Proceedings of the ACM/IEEE SC2004 Conference (SC'04)-Volume 00*, 2004.

[93] O. H. Ibarra and C. E. Kim, "Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors," *Journal of the ACM (JACM)*, vol. 24, no. 2, pp. 280-289, 1977.

[94] P. E. Crandall and M. J. Quinn, "Block data decomposition for data-parallel programming on aheterogeneous workstation network," *High Performance Distributed Computing, 1993. , Proceedings the 2nd International Symposium on*, pp. 42-49, 1993.

[95] E. Dovolnov, A. Kalinov, and S. Klimov, "Natural block data decomposition for heterogeneous clusters," *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, p. 10, 2003.

[96] A. Kalinov and S. Klimov, "Multidimensional static block data decomposition for heterogeneous clusters," *IEEE Parallel and Distributed Processing Symposium*, 2003.

[97] C. Lee and D. Talia, "Grid Programming Models: Current Tools, Issues and Directions," in *Grid Computing: Making the Global Infrastructure a Reality,* 21 ed 2003, pp. 555-578.

[98] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems," *Future Generation Computer Systems*, vol. 18, no. 4, pp. 561-572, 2002.

[99] M. Chtepen, F. Claeys, B. Dhoedt, F. De Turck, P. Vanrolleghem, and P. Demeester, "Providing fault-tolerance in unreliable grid systems through adaptive checkpointing and replication," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4487, p. 454, 2007.

[100] M. Chtepen, B. Dhoedt, F. De Turck, P. Demeester, F. H. A. Claeys, and P. A. Vanrolleghem, "Evaluation of Replication and Rescheduling Heuristics for Grid Systems with Varying Resource Availability," *Evaluation*, 2006.

[101] S. Choi and R. Buyya, "Group-based adaptive result certification mechanism in Desktop Grids," *Future Generation Computer Systems*, vol. In Press, Corrected Proof.

[102] C. Christensen, T. Aina, and D. Stainforth, "The Challenge of Volunteer Computing with Lengthy Climate Model Simulations," *First International Conference on e-Science and Grid Computing*, 2005.

[103] D. Caromel, A. di Costanzo, and C. Delbθ, "Peer-to-Peer and fault-tolerance: Towards deployment-based technical services," *Future Generation Computer Systems*, vol. 23, no. 7, pp. 879-887, 2007.

[104] P. Domingues, A. Andrzejak, and L. M. Silva, "Using Checkpointing to Enhance Turnaround Time on Institutional Desktop Grids," *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, 2006.

[105] P. Townend and J. Xu, "Fault Tolerance within a Grid Environment," *Proceedings of AHM2003*, vol. 1, no. S2, p. S3, Jan.2003.

[106] T. Wilfredo, "Software Fault Tolerance: A Tutorial," *NASA Langley Technical Report*, 2000.

[107] L. Chen and A. Avizienis, "N-VERSION PROGRAMMINC: A FAULT-TOLERANCE APPROACH TO RELIABILITY OF SOFTWARE OPERATION," *Fault-Tolerant Computing, 1995,'Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on*, 1995.

[108] I. L. Yen, E. L. Leiss, and F. B. Bastani, "A repetitive fault tolerance model for parallel programs," *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, vol. 2 1993.

[109] S. V. Anastasiadis, K. C. Sevcik, and M. Stumm, "Maximizing Throughput in Replicated Disk Striping of Variable Bit-Rate Streams," *Proceedings of the Annual USENIX Technical Conference*, pp. 191-204.

[110] D. Kondo, G. Fedak, F. Cappello, A. A. Chien, and H. Casanova, "Resource Availability in Enterprise Desktop Grids," *Dept. of Computer Science, INRIA, Technical Report*, vol. 994 2006.

[111] D. Kondo, F. Araujo, P. Malecot, P. Domingues, L. M. Silva, G. Fedak, and F. Cappello, "Characterizing Result Errors in Internet Desktop Grids," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4641, p. 361, 2007.

[112] D. Kondo, "Scheduling Task Parallel Applications For Rapid Turnaround on Desktop Grids," *Doctoral Dissertation, University of California, San Diego*, 2005.

[113] H. Norton, "The 7x7 Squares," *Annals of Eugenics*, vol. 9, pp. 268-307, 1939.

[114] P.A.MacMahon, *Combinatory Analysis* 1915.

[115] E. W. Weisstein, "Latin Square From MathWorld--A Wolfram Web Resource.," 2009.

[116] G. Birkhoff and S. Mac Lane, *A Survey of Modern Algebra* Macmillan, 1965.

[117] S. K. Park and K. W. Miller, "Random number generators: good ones are hard to find," *Communications of the ACM*, vol. 31, no. 10, pp. 1192-1201, 1988.

[118] P. Gund, "Three-dimensional pharmacophoric pattern searching," *Prog. Mol. Subcell. Biol*, vol. 5, pp. 117-143, 1977.

[119] M. J. E. Sternberg and S. H. Muggleton, "Structure Activity Relationships(SAR) and Pharmacophore Discovery Using Inductive Logic Programming(ILP)," *QSAR & Combinatorial Science*, vol. 22, no. 5, pp. 527-532, 2003.

[120] A. Wild, "Parallel Inductive Loic Programming." Master's University of Louisville, 1999.

[121] D. Page, S. Curtis, J. Graham, and A. Spatola, "A Case Study in Machine Learning for Combinatorial Chemistry," 1998.

[122] A. Kamal, "Parallel Inductive Logic Programming For Pharmacophore Discovery." Ph.d University of Louisville, 2002.

[123] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The Condor experience," *Concurrency and Computation Practice and Experience*, vol. 17, no. 2-4, pp. 323-356, 2005.

[124] "Condor Project Home Page," 2010.

[125] G. A. Klutke, P. C. Kiessler, and M. A. Wortman, "A critical look at the bathtub curve," *Reliability, IEEE Transactions on*, vol. 52, no. 1, pp. 125-129, 2003.

[126] M. Trachtenberg, G. E. Aerosp, and N. J. Moorestown, "A general theory of software-reliability modeling," *Reliability, IEEE Transactions on*, vol. 39, no. 1, pp. 92-96, 1990.

[127] D. Nurmi, J. Brevik, and R. Wolski, "Modeling machine availability in enterprise and wide-area distributed computing environments," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 3648, p. 432, 2005.

[128] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," Citeseer, 2006.

[129] A. Iosup, M. Jan, O. Sonmez, and D. H. J. Epema, "On the dynamic resource availability in grids," IEEE Computer Society, 2007, pp. 26-33.

[130] R. Bhagwan, S. Savage, and G. Voelker, "Understanding availability," *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPSÆ03)*, pp. 256-267, 2003.

APPENDIX A

DGSIMULATOR CODE

```
using System;
using System.Threading;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Reflection;
using System.Reflection.Emit;
using System.IO;


namespace DGSimulator
{

    public partial class Form1 : Form
    {
        Grid thisGrid;
        public struct jobResults
            {
                public string fileName;
            public int permutation;
            public double defaultRate;
            public int numNodes;
            public int defaultDataSize;
            public int timestamp;
            public int failedCount;
            public int completedCount;
            public int status;

            }

                public class accJobResults
                {
                    public string fileName;
                    public int permutation;
                    public double defaultRate;
                    public int numNodes;
                    public int defaultDataSize;
                    public int timestamp;
                    public int failedCount;
                    public int completedCount;
                    public double status;
                    public int reps;

                  public accJobResults()
                    {
                        fileName = "";
                        permutation = 1;
                        defaultRate = 0;
                        numNodes = 0;
                        defaultDataSize = 0;
                        timestamp=0;
                        failedCount=0;
                        completedCount=0;
                        status=0;
                        reps=0;

                    }

                    public void accumulate(jobResults jr)
                    {

                        fileName = jr.fileName;
                        permutation = jr.permutation;
                        defaultRate = jr.defaultRate;
                        numNodes = jr.numNodes;
                        defaultDataSize = jr.defaultDataSize;
                        timestamp+=jr.timestamp;
                        failedCount+=jr.failedCount;
                        completedCount+=jr.completedCount;
                        status=status+(double) jr.status;
                        reps++;
                    }
                    public void getMeanResults()
                    {
                        timestamp=timestamp/reps;
                        failedCount=failedCount/reps;
                        completedCount=completedCount/reps;
                        status=status/reps;
                    }
                }


        public Form1()
        {
            InitializeComponent();
        }

        private void label1_Click(object sender, EventArgs e)
        {

        }

        private void cmdRun_Click(object sender, EventArgs e)
        {

                thisGrid = new Grid(Globals.numNodes);
                thisGrid.addJob();
                ArrayList outputListsByHost =
thisGrid.executeNextJob(Globals.numNodes);
```

```
                int numTimeSteps = (int)outputListsByHost[outputListsByHost.Count
- 1];
                outputListsByHost.RemoveAt(outputListsByHost.Count - 1);
                outputListsByHost.TrimToSize();
                string standardHeader = "Permutation Failrate NumNodes NumData";
                string header = Globals.permutation.ToString() + "         " +
Globals.defaultRate + "     " + Globals.numNodes.ToString() + "     " +
Globals.defaultDataSize;
                double Rate = Globals.defaultRate * 100;
                string strRate = Rate.ToString();
                string fileName = "P" + Globals.permutation.ToString() + "FR" +
strRate + "NN" + Globals.numNodes.ToString() + "ND" +
Globals.defaultDataSize.ToString() + "_" + Globals.repetition;
                if (!Directory.Exists(Globals.outFileSpec))
Directory.CreateDirectory(Globals.outFileSpec);
                string path = Globals.outFileSpec + fileName;
                TextWriter tw = new StreamWriter(path);
                tw.WriteLine(standardHeader);
                tw.WriteLine(header);
                ArrayList outputByTimestep = new ArrayList();
                outputByTimestep.Capacity = numTimeSteps;
                for (int i = 0; i < outputByTimestep.Capacity; i++)
                {
                    outputByTimestep.Insert(i, null);
                }
                foreach (ArrayList outputByNode in outputListsByHost)
                {
                    foreach (ArrayList outputValue in outputByNode)
                    {

                        int counter = 0;
                        string strTmp = null;
                        foreach (string o in outputValue)
                        {

                            tw.WriteLine(o);
                            outputByTimestep[counter] =
(string)outputByTimestep[counter] + o + " ";
                            counter++;
                        }

                        if (counter < outputByTimestep.Capacity - 1)
                        {
                            for (; counter < outputByTimestep.Capacity; counter++)
                            {
                                strTmp = "t- h- d-";
                                outputByTimestep[counter] =
(string)outputByTimestep[counter] + strTmp + " ";
                            }
                        }

                    }
                }

                tw.Close();
                outputByTimestep.TrimToSize();
                TextWriter tw1 = new StreamWriter(path + ".TSP");
                tw1.WriteLine(standardHeader);
                tw1.WriteLine(header);
                foreach (string o in outputByTimestep)
                {
                    tw1.WriteLine(o);
                }
                tw1.Close();
                TextWriter tw2 = new StreamWriter(path + ".STP");
                tw2.WriteLine(standardHeader);
                tw2.WriteLine(header);
                foreach (string o in outputByTimestep)
                {

                    string strpd = o;
                    int idx;
                    string tmp;
                    while ((idx = strpd.IndexOf("t")) != -1)
                    {
                        int hdex = strpd.IndexOf("h");

                        tmp = strpd.Remove(idx, hdex - idx);
                        strpd = tmp;
                        strpd.Trim();
                        hdex = strpd.IndexOf("h");

                        tmp = strpd.Remove(hdex, strpd.IndexOf("d") - hdex);
                        strpd = tmp;
                        strpd.Trim();
                        int ddex = strpd.IndexOf("d");
                        tmp = strpd.Remove(ddex, 1);
                        strpd = tmp;
                        strpd.Trim();
                    }

                    strpd.Trim();
                    tw2.WriteLine(strpd);
                }

                tw2.Close();
                TextReader tr = new StreamReader(path + ".stp");
                char[] dataDelimiters = new char[] { ' ' };
                tr.ReadLine(); string thisLine = tr.ReadLine();
                //Permutation Failrate NumNodes NumData
                string[] macrodata = thisLine.Split(dataDelimiters,
StringSplitOptions.RemoveEmptyEntries);
                string wholefile = tr.ReadToEnd();
                char[] lineDelimiters = new char[] { '\r', '\n' };
                string[] lines = wholefile.Split(lineDelimiters,
StringSplitOptions.RemoveEmptyEntries);
                int timestamp = 0;
                bool[] testArray;
                string line;
                string[] data;
                int numData = Convert.ToInt32(macrodata[3]);
                testArray = new bool[numData];
                bool finishedFlag = false;
                int dataCount = 0;
                int hostcount = 0;
                label5.Text = " FileName " + standardHeader + " TimeStamp " + "
Host " + " Failed" + " Completed" + " Status" + "\r\n";
                for (int i = 0; i < lines.Length; i++)
                {
                    line = lines[i];
                    data = line.Split(dataDelimiters,
StringSplitOptions.RemoveEmptyEntries);
                    hostcount = 1;
                    int d;
                    foreach (string datum in data)
                    {

                        string dtmp = datum;
                        if (dtmp == "-")
                        {
                            d = -1;
                        }
                        else d = Convert.ToInt32(datum);
                        //                      d = Convert.ToInt32(datum);
                        if (d != -1 && testArray[d] == false)
                        {
                            testArray[d] = true;
                            dataCount++;
                            if (dataCount == numData)
                            {
                                finishedFlag = true;
                                break;
                            }
                        }
                        hostcount++;
                    }
                    if (finishedFlag == true)
                    {
                        break;
                    }
                    timestamp++;
                }
                tr.Close();
                SubmitNode thisMasterNode = thisGrid.getMaster();
                ArrayList nodelist = thisMasterNode.getNodeList();
                int failedCount = 0;
                int completedCount = 0;
                int nodecount = nodelist.Count;///-1?
                foreach (Node n in nodelist)
                {
                    if (n != null)
                    {
                        if (n.isFailed()) failedCount++;
                        else if (n.isCompleted()) completedCount++;
                    }
                }
                string logPath = Globals.outFileSpec + txtLogName.Text;
                tw2 = new StreamWriter(logPath, true);
                string status;
                if (finishedFlag == true) status = "1";
                else status = "0";
                tw2.WriteLine(fileName + " " + header + " " + timestamp.ToString()
+ " " + hostcount.ToString() + " " + failedCount + " " + completedCount + " " +
status + "\r\n");
                tw2.Close();
                txtOutput.Text = txtOutput.Text + fileName + " " + header + " " +
timestamp.ToString() + " " + hostcount.ToString() + " " + failedCount + " " +
completedCount + " " + status + "\r\n";
        }

        private jobResults runGrid()
        {
            thisGrid = new Grid(Globals.numNodes);
            thisGrid.addJob();
            ArrayList outputListsByHost =
thisGrid.executeNextJob(Globals.numNodes);

            int numTimeSteps = (int)outputListsByHost[outputListsByHost.Count -
1];
            outputListsByHost.RemoveAt(outputListsByHost.Count - 1);
            outputListsByHost.TrimToSize();
            jobResults jr = new jobResults();
            string standardHeader = "Permutation Failrate NumNodes NumData";
            string header = Globals.permutation.ToString() + "         " +
Globals.defaultRate + "     " + Globals.numNodes.ToString() + "     " +
Globals.defaultDataSize;
            double Rate = Globals.defaultRate * 100;
            string strRate = Rate.ToString();
            string fileName = "P" + Globals.permutation.ToString() + "FR" +
strRate + "NN" + Globals.numNodes.ToString() + "ND" +
Globals.defaultDataSize.ToString() + "_" + Globals.repetition;
           // jr.fileName = fileName;
            if (!Directory.Exists(Globals.outFileSpec))
Directory.CreateDirectory(Globals.outFileSpec);
            string path = Globals.outFileSpec + fileName;
            TextWriter tw = new StreamWriter(path);
            tw.WriteLine(standardHeader);
            tw.WriteLine(header);
            ArrayList outputByTimestep = new ArrayList();
            outputByTimestep.Capacity = numTimeSteps;
            for (int i = 0; i < outputByTimestep.Capacity; i++)
            {
                outputByTimestep.Insert(i, null);
            }

            foreach (ArrayList outputByNode in outputListsByHost)
            {
                foreach (ArrayList outputValue in outputByNode)
                {
                    int counter = 0;
                    string strTmp = null;
                    foreach (string o in outputValue)
                    {

                        tw.WriteLine(o);
                        outputByTimestep[counter] =
(string)outputByTimestep[counter] + o + " ";
                        counter++;
                    }

                    if (counter < outputByTimestep.Capacity - 1)
                    {
                        for (; counter < outputByTimestep.Capacity; counter++)
                        {
                            strTmp = "t- h- d-";
                            outputByTimestep[counter] =
(string)outputByTimestep[counter] + strTmp + " ";
                        }
                    }

                }
            }

            tw.Close();
            outputByTimestep.TrimToSize();
            TextWriter tw1 = new StreamWriter(path + ".TSP");
            tw1.WriteLine(standardHeader);
            tw1.WriteLine(header);
            foreach (string o in outputByTimestep)
            {
                tw1.WriteLine(o);
            }

            tw1.Close();
            TextWriter tw2 = new StreamWriter(path + ".STP");
            tw2.WriteLine(standardHeader);
            tw2.WriteLine(header);
            foreach (string o in outputByTimestep)
            {

                string strpd = o;
                int idx;
                string tmp;
                while ((idx = strpd.IndexOf("t")) != -1)
                {
                    int hdex = strpd.IndexOf("h");

                    tmp = strpd.Remove(idx, hdex - idx);
                    strpd = tmp;
                    strpd.Trim();
                    hdex = strpd.IndexOf("h");

                    tmp = strpd.Remove(hdex, strpd.IndexOf("d") - hdex);
                    strpd = tmp;
                    strpd.Trim();
                    int ddex = strpd.IndexOf("d");
                    tmp = strpd.Remove(ddex, 1);
                    strpd = tmp;
                    strpd.Trim();
                }

                strpd.Trim();
                tw2.WriteLine(strpd);
            }
              tw2.Close();
            TextReader tr = new StreamReader(path + ".stp");
            char[] dataDelimiters = new char[] { ' ' };
            tr.ReadLine(); string thisLine = tr.ReadLine();
            //Permutation Failrate NumNodes NumData
            string[] macrodata = thisLine.Split(dataDelimiters,
StringSplitOptions.RemoveEmptyEntries);
            string wholefile = tr.ReadToEnd();
            char[] lineDelimiters = new char[] { '\r', '\n' };
            string[] lines = wholefile.Split(lineDelimiters,
StringSplitOptions.RemoveEmptyEntries);
            int timestamp = 0;
            bool[] testArray;
            string line;
            string[] data;
             int numData = Convert.ToInt32(macrodata[3]);
            testArray = new bool[numData];
            bool finishedFlag = false;
            int dataCount = 0;
            int hostcount = 0;
            label5.Text = " FileName " + standardHeader + " TimeStamp " + " Host "
+ " Failed" + " Completed" + " Status" + "\r\n";
            for (int i = 0; i < lines.Length; i++)
            {
                line = lines[i];
                data = line.Split(dataDelimiters,
StringSplitOptions.RemoveEmptyEntries);
                hostcount = 1;
                int d;
                foreach (string datum in data)
                {

                    string dtmp = datum;
                    if (dtmp == "-")
                    {
                        d = -1;
                    }
                    else d = Convert.ToInt32(datum);
                    if (d != -1 && testArray[d] == false)
                    {
                        testArray[d] = true;
                        dataCount++;
                        if (dataCount == numData)
                        {
                            finishedFlag = true;
                            break;
                        }
                    }
                    hostcount++;
                }
                if (finishedFlag == true)
                {
                    break;
                }
                timestamp++;
            }
            tr.Close();
            SubmitNode thisMasterNode = thisGrid.getMaster();
            ArrayList nodelist = thisMasterNode.getNodeList();
            int failedCount = 0;
            int completedCount = 0;
            int nodecount = nodelist.Count;///-1?
            foreach (Node n in nodelist)
            {
                if (n != null)
                {
                    if (n.isFailed()) failedCount++;
                    else if (n.isCompleted()) completedCount++;
                }
            }
            string logPath = Globals.outFileSpec + txtLogName.Text;
            tw2 = new StreamWriter(logPath, true);
            string status;
            if (finishedFlag == true) status = "1";
            else status = "0";
            tw2.WriteLine(fileName + " " + header + " " + timestamp.ToString() + "
" + failedCount + " " + completedCount + " " + status + "\r\n");
            tw2.Close();
            txtOutput.Text = txtOutput.Text + fileName + " " + header + " " +
timestamp.ToString() + " " + failedCount + " " + completedCount + " " + status +
"\r\n";
            jr.fileName = fileName;
            jr.permutation = Globals.permutation;
            jr.defaultRate = Globals.defaultRate;
            jr.numNodes = Globals.numNodes;
            jr.defaultDataSize = Globals.defaultDataSize;
            jr.timestamp = timestamp;
            jr.failedCount = failedCount;
            jr.completedCount = completedCount;
            jr.status = int.Parse(status);
            return jr;
        }
        private void button1_Click(object sender, EventArgs e)
        {

            Globals.numNodes = (int)numericNumNodes.Value;
            Globals.repetition = 0;
            Globals.defaultDataSize = (int)numericNumData.Value;
            Globals.defaultRecordSize = 1;
            Globals.defaultRate = (double)numericFailRate.Value;//probability of
failure of each host or entire JOB?
            Globals.defaultInterval = 500;
            Globals.outFileSpec = txtOutputDirectory.Text;
            Globals.permutation = (int)numericPermutation.Value;
            Globals.numReplicants = (int)numericReplicants.Value;
            Random rnd = new Random();
            Globals.rnd = rnd;
            int numReps;
            if (numericFailStepRate.Value == 0) numReps = 1;
            else numReps = 1+(int) ( (numericMaxFailRate.Value -
numericMinFailRate.Value) / numericFailStepRate.Value);
            double stepRate = (double) numericFailStepRate.Value;
            double minRate = (double)numericMinFailRate.Value; double maxRate =
(double) numericMaxFailRate.Value;
            for (int j = 0; j < numReps; j++)
            {
                Globals.defaultRate = minRate + j * stepRate;
                numericFailRate.Value = (decimal) Globals.defaultRate;
                lblCurrentFailureRate.Text = Globals.defaultRate.ToString();
                accJobResults ajr = new accJobResults();
                for (int i = 0; i < numericRepetitions.Value; i++)
                {
                    Application.DoEvents();
                    Globals.repetition = i;
                    txtRepetition.Text = i.ToString();
                    ajr.accumulate(runGrid());//accumulates returned results in
ajr class
                }
                ajr.getMeanResults();
                string logPath = Globals.outFileSpec + "meanLog.log";
                StreamWriter tw2 = new StreamWriter(logPath, true);
               // Filename  Permutation   Failure Rate  Host Count    Data Count
       TimeStamp     Host Count    Failed Completed     Status
                tw2.WriteLine(ajr.fileName+" "+ajr.permutation+"
"+ajr.defaultRate+" "+ajr.numNodes+" "+ajr.defaultDataSize+" "+ajr.timestamp+"
"+ajr.failedCount+" "+ajr.completedCount+" "+ajr.status);
                tw2.Close();
            }
        }
        private void Form1_Load(object sender, EventArgs e)
        {
            numericFailRate.Value = (long) Globals.defaultRate;
            codeBox.Text = Globals.defaultProgram;
            txtOutputDirectory.Text = Globals.outFileSpec;
        }

    }
    public static class Globals
    {
        public static int numNodes = 100;
        public static int defaultDataSize = 1000;
        public static int defaultRecordSize = 1;
        public static double defaultRate = .9;//probability of failure of each
host or entire JOB?
        public static int defaultInterval = 500;
        public static string outFileSpec = @"c:\dgsim\";
        public static int permutation = 4;
        public static Random rnd = new Random();
        public static int repetition = 0;
        public static int numReplicants;
        public static string defaultProgram =
        "int timeStamp=0;" +
        "bFailed=false;" +
        "ArrayList outputList=new ArrayList();" +
        "double succeedRate=1-failRate;"+
        "double realRate=1-Math.Pow(succeedRate,1.0/timeSteps);"+
        "double spreadRate= (realRate*1000000);" +
        "foreach ( ArrayList dr in dataArray)" +
        "{" +

            "if (rnd.Next( 1000000 ) < Math.Round(spreadRate))" +
            "{"+
            "bFailed=true;"+
            "break;" +
            "}"+
            "for (int i=0;i<dr.Count;i++)"+
            "{"+
                "char ts='t';" +
                "char spc=' ';" +
                "string nodeInfo=ts+timeStamp.ToString()+ spc
+Thread.CurrentThread.Name+spc;"+
                "nodeInfo+=dr[i];"+
                "outputList.Add(nodeInfo);"+
                "nodeInfo=null;"+
                "timeStamp++;" +

            "}"+

        "}"+
        "return(outputList);";
    }

    class Grid
    {
        private SubmitNode masterNode;//Master node of the grid
        private ArrayList nodeList;//Item 0 is mainNode and rest are batch nodes
        private ArrayList jobQueue;//List of job items
        public Grid(int gridSize)
        {
            int masterId = 0;
            nodeList = new ArrayList();
            nodeList.Capacity = gridSize + 1;
            nodeList.Add(masterNode);
            double rate = Globals.defaultRate;
            int ti = Globals.defaultInterval;
            decimal nodeArraySize = Globals.defaultDataSize / gridSize;
            int size = (int) System.Math.Ceiling(nodeArraySize);
            for (int ID = 1; ID <= gridSize; ID++)
            {
                Node thisNode = new Node(ID,size,rate,ti);
                nodeList.Add(thisNode);
            }
            masterNode = new
SubmitNode(masterId,Globals.defaultDataSize,rate,ti,nodeList);
        }

        public int addJob()
        {
            //public Job(int dataListSize, int dataRecordSize, string s, params
double [] dataRecord)
            ArrayList data = new ArrayList();
            for (int i = 0; i < Globals.defaultDataSize; i++)
            {
                int j = i;
                data.Add( "d" + j.ToString());
            }
            Job thisJob=new Job(Globals.defaultDataSize,
Globals.defaultRecordSize,Globals.defaultProgram,data);//DATA LIST SIZE MATCHES
DEFAULT SIZE
            masterNode.addNewJob(thisJob);
            return masterNode.getJobCount();
        }
        public ArrayList executeNextJob(int numNodes)
        {
            ///dequeue the job and execute it
            ArrayList output = masterNode.executeNextJob(numNodes);
            return output;
        }
        public SubmitNode getMaster()
        {
            return masterNode;
        }
    }

   public class Job
    {

        //A job has a function and some data
       private ArrayList dataList;
       private ArrayList dataRecord;
       private ArrayList returnList;
       private int dataRecSize;
       private String strFunc;

       public Job(int dataListSize, int dataRecordSize, string s, ArrayList
jobDataList)

       {

            dataList= new ArrayList();
            dataRecord=new ArrayList();
            dataList.Capacity = dataListSize;
           for (int i=0;i<dataListSize;)
           {
               for (int j=0;j<dataRecordSize;j++)
               {
                   dataRecord.Insert(j,jobDataList[i+j]);
               }
               dataList.Add(dataRecord);
               dataRecord = new ArrayList();
               i+=dataRecordSize;
           }
            strFunc=s;
        }
       public int getDataRecordSize()
       {
           return dataRecSize;
       }
       public ArrayList getData()
        {
            return dataList;//
        }
       public String getStringFunction()
       {
           return strFunc;
       }
     }

    class MonitorForm : Form
    {

        ArrayList nodeList;
ArrayList failedNodeList;
ArrayList completedNodeList;
int numNodes;
System.Windows.Forms.Timer timer1;

public MonitorForm(ArrayList n)
{
    nodeList=new ArrayList();
    nodeList=n;
}
public void runMonitorForm()
{
    failedNodeList = new ArrayList();
    completedNodeList = new ArrayList();
    Text = "Grid Monitor";
    BackColor = Color.Blue;
    numNodes = nodeList.Count-1;//Minus one for master
    int top=1;
    int left = 1;
    timer1 = new System.Windows.Forms.Timer();
    timer1.Interval = 100;
    timer1.Tick+=new EventHandler(timer1_Tick);
    timer1.Start();
    for (int i = 1; i < nodeList.Count ; i++)
    {
        Node thisNode = (Node) nodeList[i];
        thisNode.Top = top;
        thisNode.Left = left;
        thisNode.Tag = i;


        thisNode.MouseClick += new MouseEventHandler(node_MouseClick);
        left+=20;
        if  (left>200){
            top += 40;
            left = 1;
        }
        this.Show();
        this.Controls.Add(thisNode);
    }


}
private void timer1_Tick(object sender, EventArgs e)
{
    foreach(Node n in nodeList)
    {
       if ((n != null) && n.isFailed())
        {
            n.Visible = !n.Visible;
        }
        else if (n !=null)
        {
            n.Visible = true;
        }
    }
}
```

```
    class SubmitNode : Node
    {
        Job newJob;
        Job currentJob;
        Queue jq;
        ArrayList nodeList;
        MyClassBase executableObj;
        ArrayList outputArray;
        ArrayList jobData;
        MonitorForm monitorForm;
        public SubmitNode(int ID,int size, double rate,int interval,ArrayList
nodes): base(ID, size, rate,interval)
        {
            jq=new Queue();
            nodeList = nodes;
            outputArray=new ArrayList();
            monitorForm = new MonitorForm(nodeList);
        }
        public int addNewJob(Job j)//String is an expression of the form f(x,y)
such as x*y*Math.Sin(x+y)
        {
            jq.Enqueue(j);
            return jq.Count;

        }
        public int getJobCount()
        {
            return jq.Count;
        }

        private bool isFinished(ArrayList nodeList)
        {
                bool  bLiveThread = false;
                foreach (Node n in nodeList)
                {

                    if (n != null && n.isAlive())
                    {
                        bLiveThread = true;
                        break;
                    }
                }
                return bLiveThread;
        }
        public ArrayList executeNextJob(int numNodes)
        {
            //Dequeue the job
            currentJob = (Job)jq.Dequeue();
            splitJobData(currentJob, nodeList);
            //Compile the job
            MyClassBase executableObj = new MyClassBase();
            MathExpressionParser p = new MathExpressionParser();
            p.init(currentJob.getStringFunction());
            foreach (Node n in nodeList)
            {
                if (n != null)
                {
                    n.addExecutableObj(p);
                    n.startJob();

                }
            }
            Application.DoEvents();
            Thread.Sleep(1);
            int loopcounter = 0;
            bool bLiveThread=true;
            while (bLiveThread)
            {
                Application.DoEvents();
                Thread.Sleep(1);
                bLiveThread = false;
                foreach (Node n in nodeList)
                {

                    if (n != null && n.isAlive())
                    {
                        bLiveThread = true;
                        break;
                    }
                }

            }
            ArrayList resultArrayList=new ArrayList();
            int maxTimeStep=0;
            ArrayList tmpAL = new ArrayList();
            foreach (Node n in nodeList)
            {


                if (n != null)
                {


                    tmpAL =(ArrayList) n.getResultArray().Clone();
                    resultArrayList=(ArrayList) tmpAL[0];
                    if (maxTimeStep < resultArrayList.Count) maxTimeStep =
resultArrayList.Count;
                    outputArray.Add(n.getResultArray());

                }
            }

            outputArray.Add(maxTimeStep);
            return outputArray;

        }

        public ArrayList getNodeList()
        {
            return nodeList;
        }

        public int splitJobData(Job j, ArrayList nodeList)
        {
            int nodeCount = nodeList.Count-1;
            jobData = new ArrayList();

            jobData=j.getData();
            if (((jobData.Count % nodeCount) != 0) || ((nodeCount % 2) != 0) )
            {
                MessageBox.Show("Data records do not divide evenly into nodes or
nodes not divisible by 2", "Division Error",
                MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
                return nodeCount;
            }
            int counter=0;
            int permutation = Globals.permutation;
            int nodeCounter = 0;
            switch (permutation)
            {
                case 1:
                ///Standard permutation no mirroring
                            foreach (Node n in nodeList)
                            {

                                if (n != null)
                                {
                                    for (int i = counter; i < counter +
jobData.Count / nodeCount; i++)
                                    {
                                        n.addDataItem((ArrayList)jobData[i]);
                                    }
                                    counter = counter + jobData.Count / nodeCount;
                                }
                            }

                ///Standard permutation with mirroring
                    break;
                case 2:
                  foreach (Node n in nodeList)
                  {
                      if (n != null)
                      {
                          nodeCounter++;
                          if (nodeCounter <= nodeCount / 2)
                          {
                              for (int i = counter; i < counter + jobData.Count /
(nodeCount / 2); i++)
                              {
                                  n.addDataItem((ArrayList)jobData[i]);
                              }
                              counter = counter + jobData.Count / (nodeCount / 2);
                          }
                          else if (nodeCounter > nodeCount / 2)
                          {
                              if (counter >= jobData.Count) counter =0;
                              for (int i = counter; i < counter + jobData.Count /
(nodeCount / 2); i++)
                              {
                                  n.addDataItem((ArrayList)jobData[i]);
                              }
                              counter = counter + jobData.Count / (nodeCount / 2);
                         }
                    }

                }

                //////Reverse Permutation with mirroring
                    break;
                case 3:
                    foreach (Node n in nodeList)
                    {
                        if (n != null)
                        {
                            nodeCounter++;
                            if (nodeCounter <= nodeCount / 2)
                            {
                                for (int i = counter; i < counter + jobData.Count
/ (nodeCount / 2); i++)
                                {
                                    n.addDataItem((ArrayList)jobData[i]);
                                }
                                counter = counter + jobData.Count / (nodeCount /
2);
                            }
                            else if (nodeCounter > nodeCount / 2)
                            {
                                if (counter >= jobData.Count) counter--;
                                for (int i = counter; i > counter - jobData.Count
/ (nodeCount / 2); i--)
                                {
                                    n.addDataItem((ArrayList)jobData[i]);
                                }
                                counter = counter - jobData.Count / (nodeCount /
2);
                            }
                        }

                    }
                    break;
                case 4:
                    ArrayList pList = new ArrayList();//this is a list of
permutations
                    foreach (Node n in nodeList)
                    {
                        if (n != null)
                        {
                            ArrayList tempP = new ArrayList(); //this is a
permutation
                            for (int i = counter; i < counter + jobData.Count /
nodeCount; i++)
                            {
                                tempP.Add((ArrayList)jobData[i]);
                            }
                            counter = counter + jobData.Count / nodeCount;
                            pList.Add(tempP);
                        }
                    }

                    counter = 0;
                    foreach (Node n in nodeList)
                    {
                        if (n != null)
                        {
                            nodeCounter++;
                            if (nodeCounter <= nodeCount / 2)
                            {
                                for (int i = counter; i < counter + jobData.Count
/ (nodeCount / 2); i++)
                                {
                                    n.addDataItem((ArrayList)jobData[i]);
                                }
                                counter = counter + jobData.Count / (nodeCount /
2);
                            }
                            else if (nodeCounter > nodeCount / 2)
                            {
                                if (counter >= jobData.Count) counter--;
                                for (int i = counter; i > counter - jobData.Count
/ (nodeCount / 2); i--)
                                {
                                    n.addDataItem((ArrayList)jobData[i]);
                                }

                                counter = counter - jobData.Count / (nodeCount /
2);
                            }
                        }
                    }


                ArrayList tempDataRecord = new ArrayList();
                for (int i = 0;i<pList.Count;i++)
                {
                          tempDataRecord=(ArrayList)pList[pList.Count-1];
                          int lastIndex=pList.Count-1;
                          pList.RemoveAt(lastIndex);
                          pList.Insert(0, tempDataRecord);
                          int nodePointer = 1;
                          nodeCount = nodeList.Count;
                    foreach (ArrayList p in pList)
                    {
                        if (nodePointer<nodeCount)
                        {
                            Node n = (Node) nodeList[nodePointer];
                            if (n != null)
                            {
                                n.appendDataList(p);
                            }
                            nodePointer++;
                        }
                        else
                        {
                            MessageBox.Show("Error in Latin Square 1");
                        }
                    }
                 }
                 int zz=1;
```

```
            break;
        case 5:
            int dataCounter=0;
            int numReplicants = Globals.numReplicants;
            int dataPerNode = (numReplicants * jobData.Count) / nodeCount;
            foreach (Node n in nodeList)
            {
                if (n != null)
                {
                    for (int i = 0; i < dataPerNode; i++)
                    {

                        n.addDataItem((ArrayList)jobData[dataCounter]);
                        if (dataCounter < jobData.Count-1) dataCounter++;
                        else dataCounter = 0;
                    }
                }
            }
           break;
           case 6:
               pList = new ArrayList();//this is a list of permutations
               foreach (Node n in nodeList)
               {

                   if (n != null)
                   {
                       ArrayList tempP = new ArrayList(); //this is a
permutation

                       for (int i = counter; i < counter + jobData.Count /

nodeCount; i++)

                       {
                           tempP.Add((ArrayList)jobData[i]);
                       }
                       counter = counter + jobData.Count / nodeCount;
                       pList.Add(tempP);
                   }
               }
               counter = 0;
               tempDataRecord = new ArrayList();
               for (int i = 0; i < pList.Count; i++)
               {

                   tempDataRecord = (ArrayList)pList[pList.Count - 1];
                   int lastIndex = pList.Count - 1;
                   pList.RemoveAt(lastIndex);
                   pList.Insert(0, tempDataRecord);
                   int nodePointer = 1;
                   nodeCount = nodeList.Count;
                   foreach (ArrayList p in pList)
                   {
                       if (nodePointer < nodeCount)
                       {
                           Node n = (Node)nodeList[nodePointer];
                           if (n != null)
                           {
                               n.appendDataList(p);
                           }
                           nodePointer++;
                        }
                        else
                        {
                            MessageBox.Show("Error in Latin Square 1");
                        }
                    }
               }
                break;
            default:
                break;
        }//end case
        return nodeCount;
    }
}


class Node:PictureBox
{
    /// <summary>
    ///Basic Node Class
    /// </summary>
    private int intID;
    private int intArraySize;
    private double failRate;//failrate is in form of .1 for 10 percent
    private ArrayList dataArray;
    private MathExpressionParser executableObj;
    private Thread t;
    private bool bFailed;
    private bool bCompleted;
    private bool bRunning;
    Random random;
    struct dataItem
        {
             public double x;
             public double y;
        }
    private ArrayList resultArray;
    public Node()
    {
    }
    public Node(int ID,int size,double rate,int ti)
    {
        dataArray = new ArrayList();
        resultArray = new ArrayList();
        intArraySize = size;
        dataArray.Capacity = intArraySize;
        intID = ID;
        t = new Thread(executeJob);
        t.Name = "h" + getNodeID().ToString();
        this.Width = 15;
        this.Height = 20;
        this.Name = "Node_" + intID.ToString();
        bFailed = false;
        bCompleted = false;
        failRate = rate;
        random = new Random();
    }
```

```
public bool isCompleted()
{
    if (bCompleted)
    {
        return true;
    }
    else { return false; }
}
public bool isFailed()
{
    if (bFailed){
        return true;
    } else {return false;}
}
public bool isRunning()
{
    if (bRunning)
    {
        return true;
    }
    else { return false; }
}
public bool isAlive()
{
    if (!isFailed() && !isCompleted())
    {
        return true;
    }
    else return false;
}
public Thread getThread()
{
    return t;
}
public void startJob()
{
    t.Start();
}
public void failNode()
{
    t.Abort();
    bFailed = true;
}

public int getNodeID()
{
    return intID;
}
public ArrayList getData()
{
    return dataArray;
}

public void appendDataList(ArrayList dataList)
{
    dataArray.AddRange(dataList);
}
```

```
        public void addDataItem( ArrayList dataRecord)

        {
                dataArray.Add(dataRecord);
         }
        public void addExecutableObj(MathExpressionParser obj)
        {
            executableObj = obj;
        }
        public void executeJob()
        {

            Application.DoEvents();
            bool bF=false;
            if (executableObj != null)
            {
                    resultArray.Add(
executableObj.eval(dataArray,failRate,Globals.defaultDataSize/Globals.numNodes,Glo
bals.rnd,out bF));//this returns a double but resultARRAY
            }
            bFailed = bF;
            bCompleted = !bFailed;
            t.Abort();
        }
        public ArrayList getResultArray()
        {
            return resultArray;
        }
    }

    //Beginning of compiler stuff
    public class MyClassBase
    {
        public MyClassBase()
        {
        }
        public virtual Object eval(ArrayList list, double failRate,int
timeSteps,Random rnd, out bool bFailed)//pass data to code here
        {
            bFailed = false;
            return null;
        }
    }
    public class MathExpressionParser
    {
        MyClassBase myobj = null;
        MyClassBase returnObj;
        public MathExpressionParser()
        {
        }
        public MyClassBase init(string expr)
        {
            Microsoft.CSharp.CSharpCodeProvider cp = new
Microsoft.CSharp.CSharpCodeProvider();
            System.CodeDom.Compiler.ICodeCompiler ic = cp.CreateCompiler();
            System.CodeDom.Compiler.CompilerParameters cpar
                = new System.CodeDom.Compiler.CompilerParameters();
            cpar.GenerateInMemory = true;
            cpar.GenerateExecutable = false;
            cpar.ReferencedAssemblies.Add("system.dll");
            cpar.ReferencedAssemblies.Add("DGSimulator.exe");
            string src = "using System;" +

                "using System.Collections;" +
                "using System.Threading;" +
                "class myclass:DGSimulator.MyClassBase" +
                "{" +
                "public myclass() {}" +
                "public override object eval( ArrayList dataArray, double
failRate, int timeSteps, Random rnd,out bool bFailed )" +
                "{" +
                 expr +
                "} }";
            //Compile it
            System.CodeDom.Compiler.CompilerResults cr
                = ic.CompileAssemblyFromSource(cpar, src);
            //Capture any compile errors
            foreach (System.CodeDom.Compiler.CompilerError ce in cr.Errors)
                MessageBox.Show("Error compiling Job: "+ce.ErrorText);
            if (cr.Errors.Count == 0 && cr.CompiledAssembly != null)
            {
                Type ObjType = cr.CompiledAssembly.GetType("myclass");
                try
                {
                    if (ObjType != null)
                    {
                        myobj = (MyClassBase)Activator.CreateInstance(ObjType);
                    }
                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.Message);
                }
                return myobj;
            }
            else return myobj;
        }

        public ArrayList eval(ArrayList list,double failRate,int timeSteps, Random
rnd, out bool bFailed)//timesteps is set to base amount of data per node
        {
            ArrayList outputList = null;
            bool bF=false;
            if (myobj != null)
            {
                double gridFailRate = failRate;
                outputList = (ArrayList)
myobj.eval(list,failRate,timeSteps,rnd,out bF);

            }
            bFailed = bF;

            return outputList;
        }
    }
    //End of compiler stuff
```

```
}
```

APPENDIX B

LATINGRID PROGRAM FOR CONDOR JOB SUBMISSION

```
namespace latinGrid
{
   class Program
   {
      static void Main(string[] args) //numData numHosts outputSubDir
      {
         int numData;
         int numHosts;
         string subdir;
         bool mirror = true;

         if (args.Length == 3)
         {

            numData = int.Parse(args[0]);
            numHosts = int.Parse(args[1]);
            subdir = args[2];
         }
         else
         {
            numData = 50;
            numHosts = 50;
            subdir = "mirror_test";
         }

            if (numHosts > numData) numHosts = numData;

         int pl = (int)( numData / numHosts);

         String s=null;
         String fn=null;
         int mirror_counter1=0;
         int mirror_counter2=0;
         for (int h = 0; h < numHosts; h++)
         {
                fn = "h"+(h+1).ToString();
                if (mirror == false)
                {
                  for (int x = 0; x < numHosts; x++)
                  {

                    for (int y = 0; y < pl; y++)
                    {
                      s += " c" + (((h + x + y * numHosts) % numData) +
1).ToString();
                    }

                  }
                }
                if (mirror == true)
                {

                  pl = (int)((2 * numData) / numHosts);
                  if (h < numHosts / 2)
                  {
                    for (int y = 0; y < pl; y++)
                    {

                      mirror_counter1++;
                      s += " c" + mirror_counter1.ToString();
                    }

                  }

                  if (h >=numHosts / 2)
                  {
                    for (int y = 0; y < pl; y++)
                    {

                      mirror_counter2++;
                      s += " c" + mirror_counter2.ToString();
                    }

                  }


                }

                  string t="FOR %%a IN ( "+s+" ) "+ "DO call callSerial.bat %%a";
                  StreamWriter sw;
```

```
                    Directory.CreateDirectory("D:\\workingCondorCode\\"+subdir);
                    sw                                                                    =
File.CreateText("D:\\workingCondorCode\\"+subdir+"\\"+fn.TrimStart()+".bat");
                    sw.WriteLine(t);
                    sw.Close();
                    //WRITE CONDOR SUBMIT FILE THAT CALLS BATCH FILE
                    sw      =       File.CreateText("D:\\workingCondorCode\\"       +
subdir+"\\"+fn.TrimStart() + ".txt");
                    sw.WriteLine("#Example description file foo.cmd for job foo");
                    sw.WriteLine("Executable = D:\\workingCondorCode\\" + subdir +
"\\" + fn.TrimStart() + ".bat");
                    sw.WriteLine("Universe = vanilla");
                    sw.WriteLine("#input = test.data");
                    sw.WriteLine("output = D:\\workingCondorCode\\" + subdir + "\\" +
fn.TrimStart() + ".out");
                    sw.WriteLine("error = D:\\workingCondorCode\\" + subdir + "\\" +
fn.TrimStart() + ".err");
                    sw.WriteLine("Log = D:\\workingCondorCode\\" + subdir + "\\" +
fn.TrimStart() + ".log");

                    sw.WriteLine("should_transfer_files = YES");
                    sw.WriteLine("when_to_transfer_output = ON_EXIT");
                    sw.WriteLine("transfer_input_files                               =
D:\\workingCondorCode\\serial3.exe, " + "D:\\workingCondorCode\\" + subdir + "\\" +
fn.TrimStart()        +        ".bat,        D:\\workingCondorCode\\libpl.dll,
D:\\workingCondorCode\\pthreadVC.dll, D:\\workingCondorCode\\callSerial.bat");
                    sw.WriteLine("Requirements  = (OpSys == \"WINNT60\" && Arch
== \"INTEL\" && HAS_ARENA_SOFTWARE =?= True)");

                    sw.WriteLine("Queue");
                    sw.Close();
                    Thread.Sleep(2000);

                    String commandString = "d: & cd workingCondorCode & cd " +
subdir   +   "   &   condor_submit   "   +   fn.TrimStart()   +   ".txt";//+   "     &
c:\\condor\\bin\\condor_submit.exe " + fn.TrimStart() + ".txt";
                    System.Diagnostics.Process.Start("cmd.exe",                    "/C
"+commandString);//"cmd", "/c " + command
                    s = null;
                    Console.Write("This      is      the      console      output:
D:\\workingCondorCode\\" + subdir + "\\" + fn.TrimStart() + ".txt Submitted\n\n");
                }

            Console.ReadKey();

        }
```

```
    }
}
```

APPENDIX C
GRID FAILURE MODELS

Before discussing a model for the failure of nodes (or hosts) in a grid it is important to discuss briefly the concept of failure, whether such failures are truly independent and whether they can be treated as such. The failure rate is defined as failure per unit time. Many possible definitions of node failure are possible. One might consider only hardware and network failures, power failures and software bugs as the sorts of faults that cause failure. In fact, on a traditional computer system it is entirely appropriate to limit the scope of discussion. For a computer vendor, hardware and operating system software faults might constitute a failure mode.

C.1 Classic Notions of Failure

Hardware failures often follow the bathtub curve which describes infant mortality of the hardware, followed by a period of stability and then another rise in failure rates as hardware ages. Computer chips in particular tend to follow this failure mode.

The Weibull distribution also is often used to describe hardware failure rates because it can be shifted to show infant mortality followed by a long slope of reasonably low failure rates. Software failures may be considered as a product of average error size, error density and workload. Such "classical" reasons for failure might appear at first blush to be relevant to grid computing.

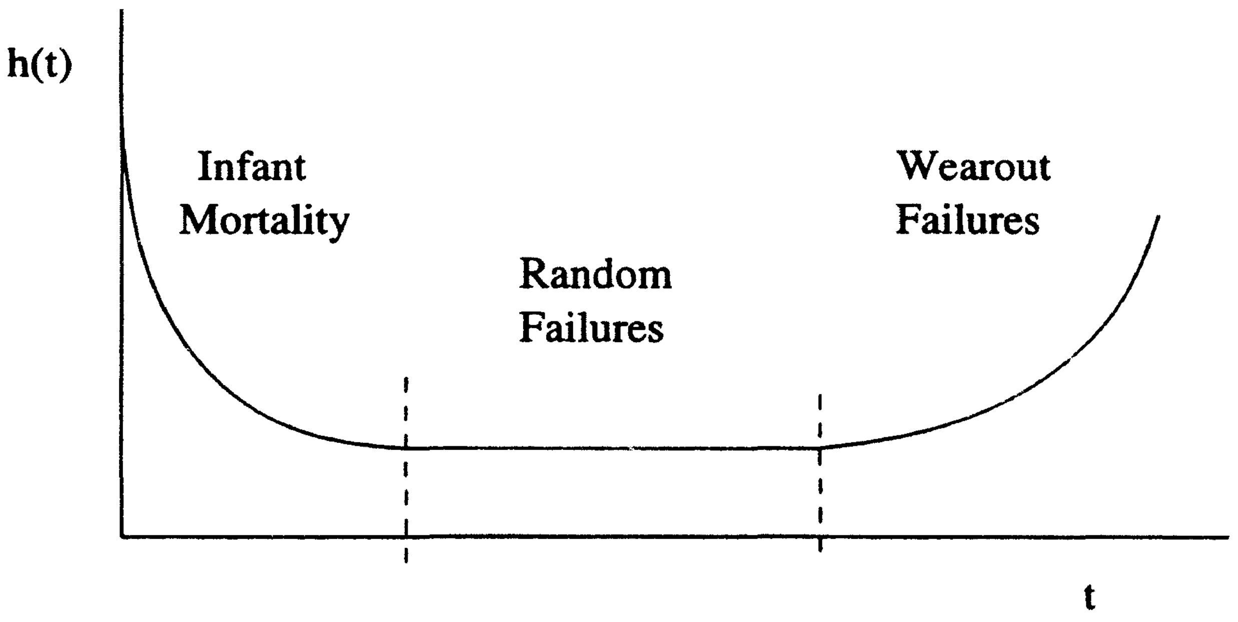


Figure C.1: The bathtub curve is composed of three hazard functions. Adapted from [125].

Although such occurrences might be technically applicable to grid computing, their effect is insignificant in the relatively short software runtimes when compared to the magnitude of user intervention errors. [126] In fact, the Weibull likely is the most accurate way to describe the failure rate on an actual grid. However the shape of the Weibull differs depending on how the shape and scale parameters are set. In [127] the authors note:

> "Our 2-parameter Weibull, as mentioned above, has parameters for shape and scale. Given a set of sample data $\{x_1 \ldots x_n\}$, there are many common techniques for estimating the two parameters based on some set of sample data, including visual inspection (e.g. using a two-dimensional graph) and analytic methods."

Zhang and others looked at reliability modeling in 2009 [86] where they point out that various authors show that the mean time between failures on high performance clusters is modeled by a Weibull, "However the shape and scale parameters are different for each study." Two studies showed that hazard rates decrease with time [127;128] while another study indicated an increasing hazard rate. [129] Some systems were not actually grids but were clusters. Without further information it is difficult to know how to accurately proceed with a Weibull model of failure on a simulated grid. When the Weibull slope $\beta = 1$, the Weibull reduces to an exponential distribution.

C.2 Failure and Independence in a Grid Environment

Consider node failure in the desktop grid environment. A task is sent to a node for execution. Either the task returns a result within some arbitrary unit of time or it does not. If it does not return by the deadline then the node can be considered to have failed. Most likely the exact cause of the failure will remain unknown. What is known is that the execution host became unavailable for some reason and the task was not completed by the deadline. So the concept of failure is linked with that of availability. In [110] [67] Kondo and others discuss three types of availability, any of which can cause failure. For a complete discussion see Section 2.5

In general however, failure of hardware components and transient software failures can compromise host availability as well as network failure. But the largest by far are users who leave the grid system either by using their computers for some other task or by turning it off. Bhagwan and others point out in [130] that, "A new intermittent component of availability is introduced by users periodically leaving and joining the system again at a later time. Moreover, the set of hosts that comprise the system is continuously changing as new hosts arrive the system and existing hosts depart it permanently on a daily basis," In their study of peer to peer systems they find that host availability is "roughly independent of the availability of other hosts" but is dependent on the time of day as shown in Figure C.2. The authors also consider the availability of one host given that another is available.

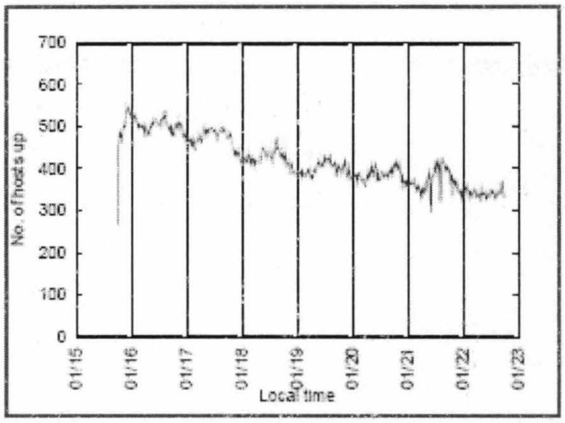


Figure C.2: The X axis is marked at midnight of the labeled day showing diurnal patterns in availability. Adapted from [130].

Given hosts X and Y they determine the conditional probability of Y being available given that X is available at a time of day t: P(Y=1/X=1). If P(Y=1/X=1) is equal to P(Y=1) meaning Y is available whether or not X is available So X and Y are independent.The authors calculated P(Y=1/X=1) and P(y=1) for every host in the peer to peer network they studied for every hour in the 7 day period. The probability density function of the difference between the two functions is shown in Figure C.3. Some 30 percent show no difference and 80 percent are between +0.2 and -0.2, showing significant independence. Correlation is to time of day. Any small sample of hosts should prove to be independent of one another.

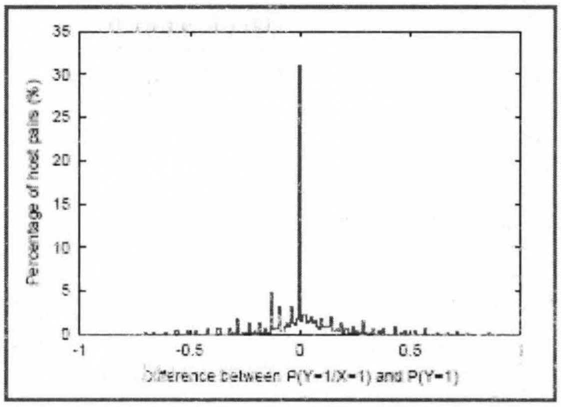


Figure C.3: Probability density function of the difference between P(Y=1/X=1) and P(Y=1). Adapted from [130]

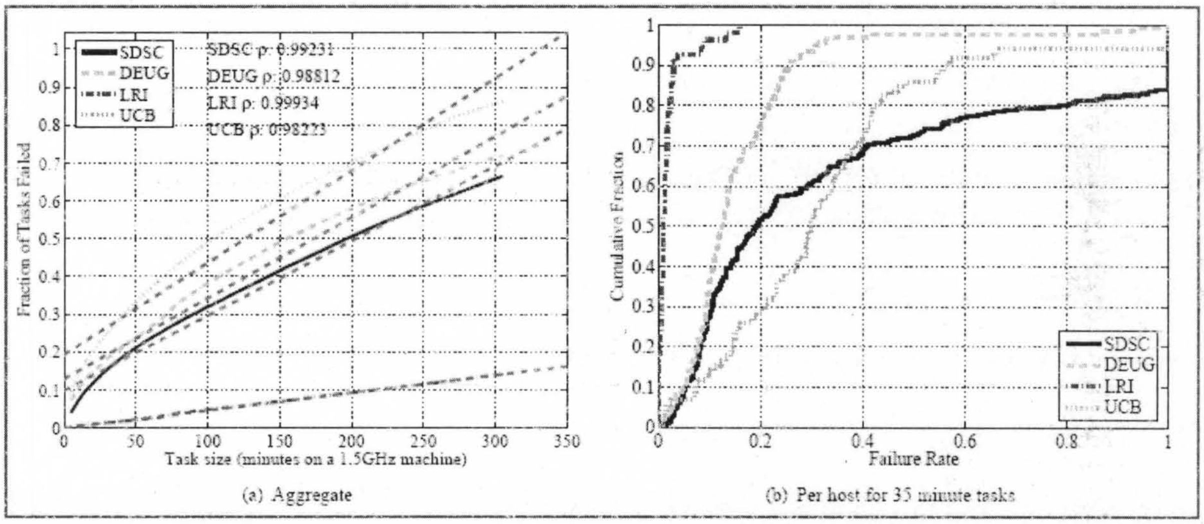


Figure C.4: Task failure rate at various task sizes (length of runtime). From [110].

Kondo and others look at a variety of grids in [110], expanding the model to examine the temporal structure of host availability and pointing out that "The successful completion of a task is directly related to the size of availability intervals, i.e., intervals between two consecutive periods of unavailability." By examining the intervals of availability during business and non-business hours they eventually come to the important result, which is the task failure rate at various task sizes (length of runtime) as described by Figure C.4.

C.3 Failure Models

Based on the above research of functioning peer-to-peer and desktop systems it appears that the important factors in determining failure rates on grid systems are availability intervals, length of task and time of day. There is little or no correlation between the availability of any two hosts in the system. Add to that the caveat that hosts will not be allowed to return to the grid after failure, at least not within the makespan of the job to be simulated in DGSimulator. The purpose is to test the overall job outcome, the makespan, when individual hosts become available and subtasks fail. Recall that makespan is the time from the beginning of the first subtask of the job to the end of the last. The purpose of the model and ensuing simulation is to test software using the RPP model against software running on a grid simulator with no redundancy and against running with course-grained task mirroring, both of which are commonly used in actual grid systems.

C.4 Homogeneous Failure Model

The failure rate for each node in the grid is assumed to be the same as any other. In other words we have no a priori knowledge of how a particular host will fail, although we do have some information about how the grid as a whole will perform based on studies of actual grids. The failure of individual hosts is assumed to be uniform and random over the makespan of a particular job. The hosts in a grid are either available or not available at any particular time. Once a node is unavailable it is assumed to be unavailable for the remainder of the job. This is justified in the following way: The definition of failure is failure of a job to return by a deadline. Whether a node is available, becomes unavailable and then becomes available again to complete the job, or whether it remains available the entire time does not matter in terms of job failure or success. If the deadline is met then the job succeeds. If it is not met then the job fails. If the job fails then the node can be assumed to be unavailable for the purposes of job completion. In fact, in actual systems, node availability is measured using application traces, whether applications meet deadlines.

Time could be modeled using the internal clock of the computer or as a timestep in a program. Because of the yes or no nature of availability of a particular host in a particular timestep it seems to be appropriate to model the failure rate, the rate at which hosts become unavailable, as a Bernoulli Process, a series of Bernoulli trials where success (p) is equal to a node failure in a particular timestep. Failure (q) is equal to a node being availability in a particular timestep. Basically the availability may be modeled as a binomial distribution.

$$\boxed{\binom{n}{k} p^k (1-p)^{n-k}} \qquad \binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{C.1}$$

Binomial Probability Mass Function Where n is number of trials, k the number of successes and p the probability of success.

The probability of having k node failures in n nodes failing with probability p is indicated by Equation C.1. In terms of accomplishing this in practice, a pseudo random number would be generated at each timestep driving a failure function. The following, for example, would fail the host with a probability of 1/n:

```
Fail()
{
        if ( rand(n) == int(n/2)) then return true
        else return false
}
```

The implication of modeling node failures as individual Bernoulli trials is that the probability of failure doesn't change but remains constant. The intention is to test all methods for reliability, including the RPP paradigm and any variants of it, under controlled conditions of node failure. Each time the probability is changed, a completely new and separate series of Bernoulli trials will be conducted over a controlled number of timesteps T > makespan M. Nodes will be assumed to fail at the same probability in each trial but with differing probabilities in different trials. Therefore changes in probability of individual node failure will take place "manually" rather than automatically according to time of day as part of the computer simulation.

Task length is another issue of importance in simulating the effects of host failure. Failure results when a task fails to complete on time. The longer the task, the greater the probability that a node will become unavailable during task execution. Variation of task length is of interest in determining the effectiveness of the RPP paradigm but does not influence design of the basic model.

C.5 Other Failure Models

Time of Day: Of course there are other ways to model the failure of individual nodes. As mentioned previously, a failure rate could be assigned according to time of day in the simulation in an attempt to model the changing failure rate on an actual running grid. In this model, each node would have the same rate of failure which would change according to time of day and might be modeled with a Weibull distribution.

The benefit of this sort of model would be to preclude questions about how realistic the simulation in fact is. In this model, the failure rate in a particular timestep would be set to increase or decrease depending on the simulated time of day. All actual grids have differing schedules of users and differing rates of change throughout the day in the failure rate. Approximation of aggregate failure rate is available from a handful of published sources. Approximation of the change in failure depending on time of day would be more difficult to obtain, and is not the essential point of this research. So the change in failure rate of any individual grid would have to be arbitrary because data likely would not be available. Also, it is difficult to see how a steadily decreasing or increasing change in the failure rate could affect the outcome of a RPP trial. An increasing failure rate makes it more likely that replicants of data will be destroyed and the job will last longer. A decreasing rate of failure makes it less likely. Also, in terms of obtaining a statistic, this model is similar to the Bernoulli. In essence the mean or aggregate probability of failure for the life of the job is the same for each machine even though individual machine probability changes at every timestep. In any case job failure can be adequately simulated and more easily controlled by doing more than one run in a model with static probability of failure rather than allowing the probability to change with each timestep.

A Priori Availability and Failure: Various systems have been described in research involving grid scheduling that attempt to use knowledge about past availability or even job success to determine which job to send to a particular node. Such a model allows scheduling heuristics such as longest job to best node or many others. Using a priori information lends itself to modeling on a simulated grid system but would require a priori knowledge about each machine in the grid. Such information is at best difficult to obtain and often impossible. Again, setting appropriate parameters for the Weibull based on actual grid operation might provide the best model of reliability. In fact, replication strategies such as the RPP paradigm are an attempt to obtain reliability without such knowledge.

In any case, it would be possible to somewhat arbitrarily assign an individual failure rate to each machine on the grid based on some a priori knowledge about the failure rates of particular machines. Rate of failure at each timestep would be geared toward past performance of the machine.

The probability distribution of availability, the times between failures, created by such a uniform set of random failures on each machine could be modeled by the exponential distribution.

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & , x \geq 0, \\ 0 & , x < 0. \end{cases} \quad \text{(C.2)}$$

Equation C.2 defines the density function of an exponential distribution where λ represents the failure rate.

In summary, considering the independence of failure among execution hosts in a grid and the nature of the outcome being tested – task failure in the face of replication and data permutation – a reliability model that includes an exponential distribution seems reasonable.

CURRICULUM VITAE

Name

Nathan Patrick Johnson

Address

Dahlem Supercomputer Lab
Vogt 204 Speed School of Engineering
University of Louisville
Louisville, KY 40292

Education

Masters of Science, Computer Science, University of Louisville, Speed Scientific School, 2002
Bachelor of Arts, Journalism, Western Kentucky University

Publications

Nathan P. Johnson, James H. Graham, " Predicting Makespan with Latin Square Replication on Computational Grids," 25th Intl. Conference on Computers and Their Applications (CATA-2010).

Nathan P. Johnson, James H. Graham: "Reliability through Replication on Desktop Grids," ISCA 22nd International Conference on Parallel and Distributed Computing and Communication Systems PDCCS-2009), 83-90

N. Johnson, J. Graham, "An Application of Grid Computing to Pharmacophore Discovery Using Inductive Logic Programming," 21st Intl. Conference on Computers and Their Applications (CATA-2006), Seattle, WA, March 2006, pp. 418-423.

N. Johnson, "A Fuzzy System For Grading Symbolic Math Problems," accepted for 11th ICIS Conference on Emerging Technologies, July 18-20, 2002

Work Experience

2000 - present University of Louisville, Louisville, KY
Assistant Director, Dahlem Supercomputer Lab
(Linux System Administrator)

- Installed and administered SUSE linux Adelie cluster with dual head 4 terabyte NAS with 19 dual processor nodes, the main server for Speed Engineering School.
- Administered Kybrin Cluster used by various schools running bioinformatics programs. Had NAS 42 CPUs with database software, web services etc.
- Installed 256 processor cluster for physics condensed matter group.
- Administered 112 processor IBM RS6000 SP2 running AIX
- Administered/installation of 26 processor Linux cluster running Redhat Linux
- Installed numerous other servers and systems including database and license servers
- Administered/installation of access grid node internet conference system
- Worked extensively with users to port/write programs in C and Fortran to run on parallel systems
- Wrote/ported numerous C and BASIC programs for users on Unix and Windows systems
- Installed numerous software packages including Ansys, Ideas. Fluent, Gaussian, Amber, PBS, MPICH, Sendmail and a variety of others
- Taught several seminars in parallel programming using MPI
- Aided in decision-making for hardware upgrades and administration of the Dahlem Supercomputer Lab
- Worked with vendors and obtained quotes and specifications for various systems

Contract Configuration Management Database system Vanderlande Inc. United Parcel Service Hub 2000 project

- 2002: The system allowed multiple views of the hundreds of devices in the UPS Hub 2000 sorting system and produced an output file suitable for use with their Sun supercomputer routing system

1999-2000 Indiana University Southeast, New Albany, IN Visiting Lecturer

- Taught C201 Introduction to Computer Programming with Visual Basic

- C311 Survey of Programming Languages (Design of Computer Languages using a variety of languages to show design principles)
- C106 Introduction to Computers and Their Use

1996-1999 Indiana University Southeast, New Albany, IN
Adjunct Faculty

- Taught C201 Introduction to Computer Programming with Visual Basic
- C203 Introduction to COBOL
- C320 Advanced COBOL
- C106 Introduction to Computers and Their Use
- C100 Microsoft Word 97, Access, Excel, PowerPoint, Windows
- C100 Novel WordPerfect, Paradox, Quattro Pro for Windows

1995-1997 Consultant States News Service, a Washington, D.C. based radio news service supporting reporters in 50 states transmit stories to the main system

1990-1992 The News-Enterprise, Elizabethtown, KY
Assistant Editor

- Supervised reporters and photographers
- Edited locally produced copy
- Served on the Editorial Board

Opinion Page Editor

- Editorial writing
- Opinion page content
- Column writing
- Editing letters and reader hotline

1984-1989 Hattiesburg American, Hattiesburg, MS
Weekend News Editor

- Special projects reporting
- Headed computer graphics department including training at Gannett Graphics Network in Washington, D.C

- Saturday and Sunday front page layout
- Copy editing/headline writing

Reporter

- Government reporter covering state legislature, city-county government
- Wrote political news analysis
- Wrote variety of feature pieces

1982-1984 Kentucky Standard, Bardstown, KY
Reporter

- Covered courts, county and city government, wrote features, shot and photos, did some page layout and editing

1981 Kentucky Standard, Bardstown, KY
Reporting intern

1980-1982 College Heights Herald, Bowling Green, KY
Reporter/Copy Editor/Chief (administration and budget) reporter

- Newspaper won the national Pacemaker award

Awards and Honors

- Reviewer 2009, PDCCS Conference, Louisville, KY
- Graduate Deans Citation, 2002, University of Louisville
- 2nd place Best Column, 1992 Kentucky Press Association; 1st place, Best General News Story, 1991 Kentucky Press Association; 1st place, Best Front Page Layout, 1990 Mississippi Press Association; 2nd place, Best Feature Story, 1986 Mississippi Press Association; 2nd place, Best Investigative story, 1983 Kentucky Press Association; 3rd place. news story, honorable mention, column, 1982 Kentucky Press Association, 1st place feature writing, 1981 Kentucky Intercollegiate Press Association, 1st place feature story, 1979 University of Kentucky Community College system and other intra-company and staff awards
- National merit semifinalist
- 1987 President South Mississippi Chapter of the Society of Professional Journalists

Languages

C#, C/C++, Visual C++, Visual Basic, SQL, COBOL, some Fortran, some Java, Bash and AWK script, Message Passing Interface for parallel systems, Maple programming language, 3D graphics programming with OpenGL, ODBC, JDBC, some Smalltalk, ModSim, Lisp on Linux or Windows platforms

Systems

2002: Master's Thesis -- "A System For Grading Symbolic Mathematical Expressions Using Maple With Fuzzy Sets" -- Compared two symbolic mathematical expression and returned the degree of similarity using Maple Programming Language, C++, Java with ODBC and JDBC as well as artificial intelligence techniques

- 1997: Maple Assisted Math Grading Program
- Primarily responsible for the central control module and client interface for a Maple assisted math grading system.

Speed projects

- Network Search Using Multiple Networked Processors in UNIX -- My partner and I used Parallel Virtual Machine to search a theoretical network. PVM is a C library that allows remote procedure calls. The project spawned jobs among the computers on an HP system at the University of Louisville in an attempt to find the fastest route from one point to another through a simulated network. The purpose was to work toward establishing real-time routes for multimedia.
- Topological Surface Rendering -- I designed and implemented a system for converting U.S. Geological Survey elevation files into an accurate three dimensional rendering of the earth's surface. Specifically I used Visual C++ and OpenGL to render the topology of Louisville, KY in three dimensions.
- Student Advising System -- My primary responsibility was writing a Visual C++ client interface to allow academic advisors to view a student's academic requirements, prerequisites completed and other course information stored in a Microsoft SQL Server database. i also participated in the database design. The client module used ODBC to connect to the database.