Abstract of Dissertation

The computing requirements of very large engineering and scientific applications are continuing to increase and surpass the capabilities of current computing platforms. Trying to address these requirements by developing more sophisticated, elaborate and expensive computer architectures may not be feasible. Instead, this dissertation will describe the design of a *Virtual Computing Environment* (VCE) which allows applications to be designed for, and executed on, a collection of distributed heterogeneous high performance computers. By executing each portion of an application on the computing system best suited to it, application performance can be improved without investing in the design and fabrication of new hardware platforms.

The VCE provides a suite of tools to permit scientists and engineers to develop and execute distributed applications without mastering the intricacies of distributed or parallel programming. The VCE design is modular. It includes a software engineering environment, a run time environment, and an *Application Description Framework* which is the main interface between the two. The software engineering environment separates application development from program development. Applications are developed by identifying a set of tasks or objects and indicating their communication and synchronization requirements. Task implementations are general and reusable. Each task can be implemented on a variety of architectures so the choice of where to run a task can be made by the VCE run time system. The application description framework is used to capture information about the structure of the application. The application description is parsed by the VCE run time system, which dynamically determines what types of machines are available and their load conditions. With this information it selects machines to host application tasks. When load conditions permit, redundant scheduling is used to maximize application performance.

The design of the VCE has been validated by constructing simulated and prototype systems and applications. These systems were used to demonstrate the advantages of redundant scheduling policies. Several experiments have been performed to compare the performance of redundant scheduling to more conventional selective scheduling approaches in a variety of hardware configurations and load conditions.

The Design of a Distributed Virtual Computing Environment

by

Philip Rousselle

BS SUNY College at Buffalo, 1983

MS Union College, 1989

Dissertation

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Engineering
in the Graduate School of Syracuse University

May 1996

Approved _____

Prof. Salim Hariri

Date _____

© Copyright 1996
Philip Rousselle

Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Research Objectives 3

1.1.1 The Needs of Scientific Computing 4

1.1.2 Applications of Multiple High Performance Computers 5

1.1.3 The Needs of Workstation Users 7

1.2 Limitations of Current Approaches 8

1.2.1 Distributed Operating Systems 9

1.2.2 Remote Procedure Call, Message Passing and Distributed Shared Memory Environments 10

1.2.3 Load Sharing Environments 12

1.2.4 Workstation Based Parallel Programming Environments . . . 13

1.2.5 High Performance Application Development Tools 13

1.3 The VCE Approach 14

1.3.1 Structure of the VCE 14

1.3.2 Contributions 17

1.4 Organization of the Dissertation 18

2 Distributed Computing Environments 20

2.1 Basic Distributed Computing Tools 20
2.2 RPC and the Client Server Model 21
2.3 Message Passing Systems 24
2.4 Distributed Shared Memory Environments 24
2.5 Distributed Operating Systems 25
2.6 Load Sharing Environments 26
2.6.1 Load Sharing on Homogeneous Workstation Clusters 26
2.6.2 Load Sharing Among Heterogeneous Workstation Clusters . . 28
2.6.3 Workstation Cluster Based Parallel Processing 28
2.7 Parallel Program Development Tools 31
2.8 Distributed Application Description Languages 32
2.8.1 Durra System Overview 33
2.8.2 Durra Tasks and Applications 34
2.9 Distributed Object Systems 37
2.9.1 DSOM System Overview 38
2.9.2 DSOM Application Development 42
2.10 High Performance Computing Over the World Wide Web 42
2.11 Separating Coordination from Computation 43
2.12 The VCE Approach 43

3 Overview of the VCE Architecture **47**
3.1 VCE Building Blocks 47
3.2 The Software Development Suite 52
3.2.1 Example Application Development Activities 54
3.3 The Application Description Framework 56
3.3.1 Supporting Object Oriented and Other Development Methods 57
3.4 The Execution Suite 58
3.4.1 Components of the Execution Suite 59
3.5 Load Balancing 61
3.6 Communication 63

3.7 Design Attributes 64

4 The VCE Application Development Model 66

4.1 The VCE Application Construction Approach 67

4.2 The VCE Application Description Framework 69

4.2.1 VCE Platforms 69

4.2.2 VCE Tasks 70

4.2.3 Intertask vs. Intratask Communication 70

4.2.4 VCE Task Description Syntax 71

4.2.5 Task Input/Output Semantics 73

4.2.6 VCE Application Description Syntax 74

4.2.7 Object Oriented Input/Output Semantics 79

4.2.8 Other Communication Semantics 80

4.2.9 Designing VCE Tasks and Applications 80

4.2.10 Extensions to the Application Description Framework 81

4.3 Application Execution 82

5 The VCE Execution Environment 83

5.1 VCE Scheduling Techniques 84

5.1.1 The VCE Execution Environment Components 86

5.1.2 Elements of the Load Balancing Strategy 86

5.1.3 Overview of Scheduling Activities 96

5.1.4 Rationale 97

5.2 Managing Communications Within the VCE 102

5.2.1 VCE Application Communication Issues 103

5.2.2 Components of the VCE Intertask Communication System . . 105

5.2.3 Managing Communication Among Concurrent Redundant Tasks 107

5.2.4 Managing Communication Among Predecessor and Successor Tasks 115

5.2.5 Managing Communication Among VCE System Components . 116

5.2.6 Managing Location Information in the VCE 120

5.2.7 Managing Dynamic Changes in the VCE 122

6 VCE Design Validation **124**

6.1 A Workstation Cluster Based Prototype 125

6.1.1 Scope of the Prototype 125

6.1.2 Low Priority Dispatching 126

6.1.3 Workstation Load Measurement 127

6.1.4 Task Placement 129

6.1.5 Experimentation 129

6.2 A Multiple Parallel Computer Based Prototype 131

6.2.1 Scope of the Prototype 132

6.2.2 Hardware Platforms 133

6.2.3 Applications 135

6.2.4 Performance 136

6.3 A VCE Simulation Model 139

6.3.1 Evaluating Distributed System Designs 140

6.3.2 Basic Elements of the Simulation Model 141

6.3.3 Parallel Applications 149

6.3.4 Workstation Clusters 151

6.3.5 Modeling Processor Workload 151

6.3.6 Modeling VCE Components 152

6.3.7 Instrumentation 153

6.3.8 Simulator Results 154

6.3.9 Use of the Simulator to Design VCE Configurations 160

6.4 Summary 161

7 Conclusions and Future Work **162**

7.1 Conclusions 162

7.2 Future Work 163

7.2.1 Application Description Framework Enhancements 163

7.2.2 Implementation of a VCE Communication System 166

7.2.3 Refinement of the VCE User Interface 166
7.2.4 Simulation Model Enhancements 167

Bibliography **169**

Biographical Data **182**

List of Tables

1 Performance of Two Components of a High Performance Application on Various Computing Platforms [74] 7
2 Capabilities of HPC Environments 45
3 Communication Characteristics of VCE Components 119
4 Performance Benchmark 130
5 Text Processing Application Performance 131
6 Performance of the Linear Equation Solver Application 138
7 Example Simulated Tasks 148
8 Processor Simulation Example 148
9 Application Performance at Low Load Levels With Selective Placement 155
10 Application Performance at Low Load Levels Without Selective Placement 156
11 Application Performance at Moderate Load Levels With Selective Placement 158
12 Application Performance at Moderate Load Levels With Low Priority Workstation Dispatching 158
13 Application Performance at Moderate Load Levels With Low Priority Dispatching on All Machines 159

List of Figures

1 Computing and Storage Requirements of Grand Challenge Applications [58] 5
2 Major Components of the VCE 16
3 A Conventional Data Base Application 22
4 A Distributed Data Base Application Using RPC 23
5 Location Transparency in MOSIX [9] 27
6 Components of the Phish System [20] 30
7 Elements of the Durra System [12] 34
8 Durra Simple Task and Channel Descriptions [13] 35
9 Durra Producer-Consumer Application with Dynamic Reconfiguration [13] 36
10 Components of the DSOM Distributed Object System 39
11 DSOM Repository Files 39
12 Initiating a DSOM Remote Object Access 40
13 Completing a DSOM Remote Object Access 41
14 Structure of the VCE 48
15 The VCE Execution Environment 50
16 Task Graph of a Meteorological Application 54
17 Annotated Design Stage Task Graph 55
18 Coding Level Task Graph 56
19 The Virtual Computing Environment 58
20 Organization of Machines in the VCE 60
21 Communication via Proxies 64

22 VCE Task Description Syntax 71
23 VCE Task Descriptions for Linear Algebra Operations 73
24 VCE Communication Operations 74
25 VCE Application Description Syntax 75
26 A VCE Application 77
27 VCE Load Thresholds 89
28 Load Measurement Algorithm 90
29 Load Measurement Algorithm Example 91
30 Execution Process Scheduling Activities 98
31 Group Leader Scheduling Activities 99
32 Group Member Scheduling Activites 100
33 The Function of VCE Communication Daemons 106
34 Input Data Table Entry Format 107
35 Output Data Table Entry Format 107
36 Task Instance Output Message Format 108
37 Format of Data Sent Between Communication Daemons 108
38 An Algorithm for Managing Communication Among Concurrent Tasks 111
39 Intertask Communication Pattern Examples 113
40 Components of the VCE Communication System 117
41 Communication to Support Multiple VCE Tasks on a Single Host . . 118
42 Computing Environment at NPAC 134
43 Multi-Target Tracker System 136
44 A Typical Node Model 143
45 The VCE Node Model 144
46 Processor Event Simulation Algorithm 147
47 Matrix Preprocessor Task Description 164
48 A Dynamic Application Description 165

Chapter 1

Introduction

The evolution of modern electronic computers began only 50 years ago and has been driven to a large extent by rapid advances in electronic technology during that period. The first computers ran one application at a time. Without the benefit of operating systems or compilers, application programmers were responsible for managing all aspects of the hardware. The introduction of compilers allowed programmers to express algorithms in abstract terms without being concerned with the bit level details of their implementation. Time sharing operating systems allowed hardware resources to be efficiently utilized by several applications simultaneously.

As electronics evolved from vacuum tubes to transistors to integrated circuits computers became faster and more powerful. The complexity of hardware and systems software increased to support larger numbers of users and complex applications running on faster and faster systems.

More recently the trend towards complex monolithic computer systems has changed. VLSI advances over the last 15 years have made computers cheaper and smaller. This has enabled computer manufacturers to deliver small systems that fit on or beside a desk and can satisfy the computing needs of many users more cost effectively than timesharing systems. One drawback to this approach is that it results in huge amounts of excess computing capacity: few users can generate enough work to keep their systems busy all the time.

Another trend that can be observed over the past 50 years is that scientists, mathematicians and engineers have consistently been able to envision applications that cannot be accommodated by the computers available to them. This has led to the evolution of scientific computing technology separate from commercial computing.

Microelectronics have evolved to the point where future performance improvements of several orders of magnitude cannot be expected. Electronic signals cannot propagate faster than the speed of light. This will be a limiting factor in the improvements that can be expected in processor and memory system speed. Instead, computer engineers have sought to answer the challenges of scientific computing by designing hardware systems and algorithms with high levels of parallelism. Unfortunately, Parallel systems often lack generality. An architecture which provides very good performance on one problem or algorithm may perform poorly on others. Lack of generality may even be manifested as poor scalibilty, where a particular algorithm running on a given machine runs satisfactorily for a given instance of a problem but is inefficient on much larger or smaller instances.

Consequently, two problems currently facing the field of computer engineering can be identified: (1) The trend towards workstation based computing results in large amounts of underutilized processing resources, and (2) Specialized scientific computing architectures benefit some computing tasks more than others. This dissertation will outline how both these problems have been addressed by incorporating network attached workstations, and scientific computers of various architectures, into a distributed computing environment. The environment provides tools to allow users to develop applications in terms of an interconnection of constituent tasks and a run time system which manages execution of these tasks in shared high performance computers and workstations connected across a wide area network. This distributed computing environment, which enables users to define and efficiently execute heterogeneous distributed applications, is referred to as a Virtual Computing Environment (VCE).

1.1 Research Objectives

The main focus of the VCE is to:

- Provide a software engineering environment that will enable scientists and engineers to build applications that use multiple high performance computing systems.
- Make the computing power of idle and underutilized computers available to these applications.
- Provide a run time system that employs effective scheduling and dispatching methods to maximize performance.
- Allow VCE applications to execute on shared hardware without unduly impacting non-VCE users. That is, the VCE should not require dedicated hardware.
- Use existing tools and technologies whenever possible so that the the VCE will not be too complex or too expensive to implement.
- Avoid operating system modifications so the system will be portable and can support high levels of heterogeneity.
- Be modular and flexible enough so that future software engineering tools and hardware platforms can be incorporated easily.

The use of multiple high performance computers has been shown to be an effective way to improve the performance of real world applications (Section 1.1.2). However, dedicating multiple supercomputers to a single application is often not feasible. Also, the skill required to implement and run these applications is considerable. By meeting the objectives listed above, the VCE will allow such applications to be developed by a larger group of users (not just high performance computing experts) and executed on shared hardware. Existing distributed computing environments are not robust enough

to achieve this. Section 1.2 and Chapter 2 will discuss a wide variety of distributed computing environments and tools. None of them satisfies all these objectives.

The objective of the VCE design is to describe and validate a strategy for providing a set of software engineering and application execution tools which satisfies the items listed above. Central to this strategy is the differentiation between programming-in-the-small and programming-in-the-large [35]. Many tools exist to support parallel programming. The VCE, rather than focusing on development of individual parallel programs, is concerned with allowing users to build large scale applications from multiple parallel (or sequential) programs running on distributed heterogeneous computers. A VCE application designer specifies a set of programs and indicates how they are synchronized and communicate. Each program (task) in the application may have its own hardware requirements. The application designer is not concerned with how tasks are implemented. General purpose reusable task implementations are developed by parallel programming specialists. Application designers concentrate on the structure of applications in a problem area where they have expertise. The VCE run time system dynamically selects hosts for each task and manages their dispatching and communication. The VCE provides a technology independent environment for the development of reusable high performance tasks, development of applications that use these tasks and automated execution of the applications.

1.1.1 The Needs of Scientific Computing

As has been the case throughout the brief history of electronic computing, the computing needs of scientific and engineering applications continue to tax the capabilities of current hardware. The requirements of such applications have led to the development of specialized software systems to manage large numbers of dedicated workstations which work cooperatively on scientific applications [41] [88]. A variety of supercomputer architectures [58] such as SIMD, MIMD and vector machines have also been designed for use on such applications. Each architecture represents a different set of trade-offs and will give different levels of performance for any given problem. Further,

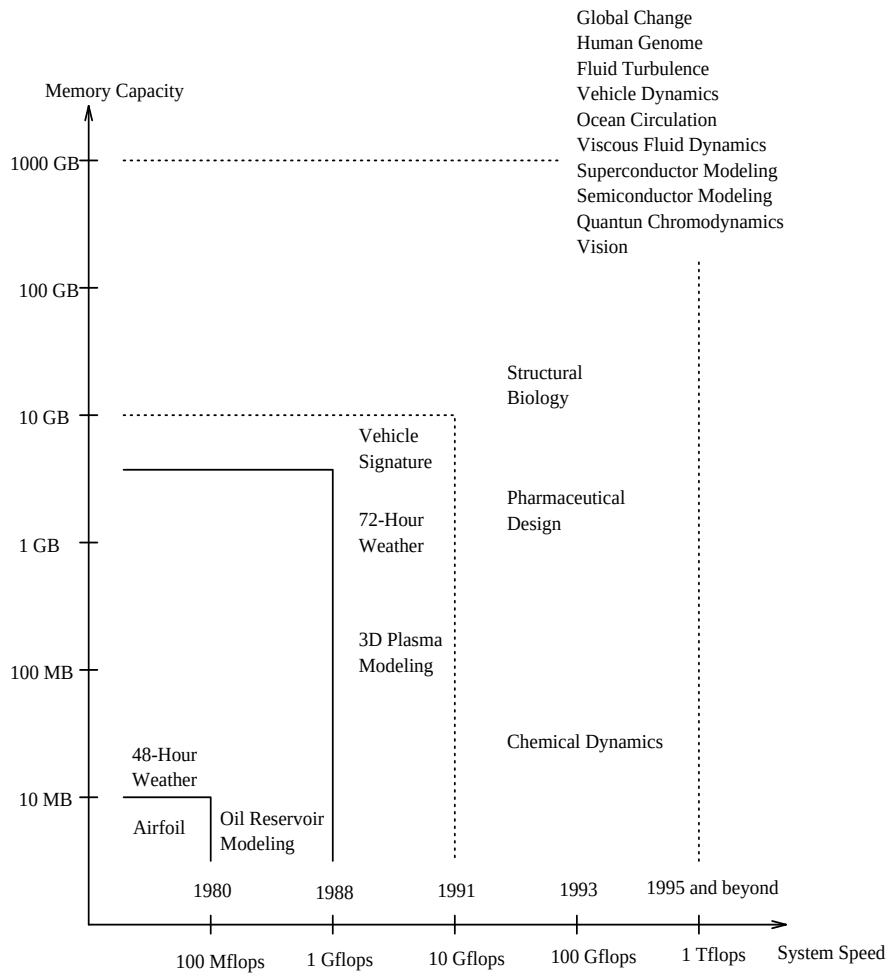


Figure 1: Computing and Storage Requirements of Grand Challenge Applications [58]

as illustrated in Figure 1 many applications that will be of interest to researchers in the near future will exceed the computing resources that could reasonably be packaged in any single processor complex.

1.1.2 Applications of Multiple High Performance Computers

The next step in developing these applications is likely to entail the use of multiple supercomputers to attack a single problem. An example of such an effort is described

in [74]. Part of the project involved execution of a global climate model. The model was implemented as two separate components: a physics model which computes the effect of non-atmospheric forces on climatelogical conditions, and a dynamics model which simulates the evolution of the fluid flows using appropriate equations. The results produced by the physics component are supplied to the dynamics model as forcing terms. These two application components have different computational properties. The physics model can be efficiently parallelized by dividing the simulation domain (the atmosphere) into vertical columns. Because the physics component is primarily concerned with calculating the effects of localized phenomenon, little communication between processors is needed and high performance can be achieved on massively parallel computers. Conversely, the atmospheric dynamics model incurs much higher communication overhead which reduces its efficiency when it is executed on a parallel machine. Table 1 shows the running time in seconds per simulated day for both components of the application on seven different high performance computers. When the two components are executed on different machines they can execute in parallel and communication operations can be overlapped with computation so that the distribution overhead is minimal. If both components of the application are executed on one machine, the best possible running time is 83 seconds per simulated day on the CRAY C90. By using two supercomputers, the CRAY C90 and the Intel Paragon, performance was improved to 40 seconds per simulated day.

The performance data in Table 1 reflects benchmarks performed on otherwise idle machines. If the application were run on shared hardware its performance would be effected by transient load conditions. Using benchmark performance data to staticly allocate application components to hardware resources may not achieve the best possible results. For instance, the data shows that the physics component performs equivalently on both the IBM SP1 and the Intel Paragon, which suggests that if both kinds of machines are available at run time, choosing the one that is less loaded will improve the application's performance. This suggests the need for automated scheduling technology to make and execute these run time decisions.

In addition to the need for computer systems with adequate processing power to

Table 1: Performance of Two Components of a High Performance Application on Various Computing Platforms [74]

Platform	Dynamics Component	Physics Component	Total Running Time
CRAY C90	29	54	83
CRAY Y-MP	61	104	165
Intel Paragon	132	40	172
Intel Delta	192	48	240
IBM SP1	100	40	140
CM-5	125	45	157
DEC ALPHA Cluster	137	116	253

tackle their applications, scientists also need an environment which allows them to construct applications efficiently [7] [53]. Scientists and engineers are concerned with expressing the structure of their computations and obtaining computer systems which achieve acceptable performance for those computations. Computing environments which require them to learn new programming languages or be concerned with load balancing and similar low level hardware issues are unlikely to succeed. Conversely, environments which allow users to build large applications by reusing and customizing existing software components are more likely to be embraced [7].

1.1.3 The Needs of Workstation Users

One goal of the Virtual Computing Environment is scalability. The VCE should provide benefits in configurations as small as a few homogeneous workstations up to networks containing many supercomputers of various architectures and large numbers of heterogeneous workstations. Because almost all anticipated VCE configurations will include some workstations, the current research gives considerable attention to issues involved in optimizing the use of such machines in distributed environments.

Studies of workstation usage patterns were conducted by Mutka [81] and Mutka

and Livny [80]. Such a study was also included as background material by Krueger and Chawla [67]. All these studies found that networks of workstations were largely underutilized even at peak times. The characterization of Krueger and Chawla [67] that "...unused capacity rarely falls below 87%" was typical of all three sources.

Mutka and Livny [80] found that most workstation users fell into one of three categories:

- **Casual users** who perform text editing and mail functions which never fully utilize their workstations.
- **Sporadic users** who occasionally run short non-interactive tasks such as compilations.
- **Frustrated users** who run large jobs on a regular basis and are constrained by the capabilities of their workstations.

The goal of the VCE is to provide relief to frustrated users by giving them friendly and intuitive tools for exploiting the excess resources controlled by other users. The capabilities of the VCE can also be used to provide benefits for sporadic users by allowing them to run large tasks on faster machines and by providing tools like parallel *make* [60, 109]. These benefits are realized without imposing significant performance penalties on casual or sporadic users. Such penalties could cause users to resist participating in the VCE. Workstation users expect to receive consistent service levels at all times (unlike users of timesharing systems who must accept performance degradations during the busiest times of the day).

1.2 Limitations of Current Approaches

Chapter 2 will discuss a wide variety of distributed system paradigms which have been proposed to address some of the issues which have motivated the VCE design. None of those technologies provide an adequate solution to the problems associated with designing, implementing and executing very large applications in a large shared

resource network environment. This sections will outline the shortcomings of current distributed system technologies so that the contribution of the VCE approach can be demonstrated.

1.2.1 Distributed Operating Systems

Not only will the VCE provide a platform for the development and execution of large applications and exploit idle resources in computer networks, it has been engineered so as to be cost effective. No software system can be considered successful if it costs so much to develop and maintain that it cannot be made available to the user community it was designed for.

The VCE can coexist with other software. Users who prefer to use other tools to do their work will not be excessively impacted by their machine's participation in the VCE. The VCE will give users a new option for running their applications. Any application that is already running in a more conventional environment and delivering satisfactory performance will not experience significant performance degradation due to a host machine's participation in the VCE.

The importance of these two criteria is demonstrated by the failure of distributed operating systems developed over the last 15 years to gain wide usage. The approach of building a special purpose kernel to run in each machine in a distributed environment has some merit. Significant performance gains can be realized by building customized communication protocols and other distributed computing support into the kernel. However, developing an operating system from scratch is an ambitious undertaking. The LOCUS project [105] for example, resulted in a kernel one third larger than the Unix kernel and required 50 man years of effort to design and develop. The costs of such a project, and the costs of maintaining such a large system, would be prohibitive.

Some distributed operating systems (such as Amoeba [78]) present user and programming interfaces which are not compatible with the familiar Unix environment. Processors in Amoeba are assigned to tasks dynamically and are largely unable to

function autonomously. Such a departure from the dominant computing model (the Unix workstation model) is unlikely to be successful. Recognition of this fact is a trend in the literature: from the distributed operating systems of the 1980's and early 1990's [11, 30, 78, 85, 105] to remote execution systems in the early 1990's that used small kernel modifications to achieve global scheduling and location transparency [67, 33] to very recent load sharing systems implemented completely above the kernel [20, 24, 60, 87, 109].

1.2.2 Remote Procedure Call, Message Passing and Distributed Shared Memory Environments

Many tools are available to ease the process of developing a single application with components spread over a number of networked machines (see Sections 2.2, 2.3 and 2.4). The focus of these tools is to provide programmers with abstract communication models which will be more usable than the communication primitives (like *sockets* [96]) provided by conventional operating systems.

Remote procedure call (RPC) systems provide mechanisms for binding a calling program with a procedure running on a different machine. RPC functions are generally used to allow users to access specialized resources such as file systems, data bases or high performance CPUs resident on remote machines. RPC functions are not usually suitable for high performance applications because the procedure call interface is inherently synchronous and awkward for use in parallel processing.

Message passing systems, compared to RPC paradigms, provide a similar but more flexible communication model. The similarity is that RPC allows a sending program to send a set of arguments to a remote process which receives them as input on its invocation. With message passing the arguments are passed to a *send* function and retrieved by a remote process using a *receive* function. The flexibility of message passing is achieved by providing blocking and non-blocking send and receive operations which facilitate parallel operations. Most message passing systems also provide *broadcast* operations which allow a master process to send parameters to a

collection of slave processes which perform some computation in parallel. Such an operation is awkward to replicate with RPC primitives because the procedure call model supports a one-to-one communication pattern.

The distributed shared memory model is an even more flexible paradigm. Instead of *send* and *receive* functions, *write* and *read* semantics are used to pass data among remote processes. A distributed shared memory application might have one master component that places sets of parameters into a shared buffer where they are read by slave processes which perform computations and place the results in the buffer for retrieval by the master process. Distributed shared memory allows the application programmer to create synchronization as needed using semaphore techniques. Atomic "read and remove" operations are usually provided which provide sufficient serialization for many applications. These facilities allow the user to achieve as much or as little synchronization as is necessary. They can be more friendly than message passing schemes which may require explicit data conversion by the application and require message senders to maintain and supply addresses or address lists of recipients.

None of these tools represent a wholly satisfactory environment for developing high performance distributed applications or for harnessing idle resources. They are all low level programming tools that must be incorporated into application programs written in conventional languages such as C, C++ or Fortran. The goal of computing environments which support grand challenge applications must be to provide scientists and engineers with the ability to fashion applications from existing components rather than requiring them to become or supervise programming experts.

Many RPC, message passing, and distributed shared memory tools require the programmer to declare specific machines for each process to run on. This practice precludes dynamic run time load balancing and limits the potential of such systems to reclaim idle resources on a large scale.

1.2.3 Load Sharing Environments

Load sharing environments, discussed in detail in Section 2.6, provide the capability to schedule work on idle or lightly loaded networked computers. This is accomplished in a variety of ways. One approach is to compile large CPU bound jobs with a special library which allows their I/O requests to be intercepted by a run time system. Such jobs are scheduled on idle machines and their I/O operations are forwarded to their originating machines for processing. These remote execution applications can be migrated among homogeneous hosts as load conditions change. Another way to achieve location transparency and process migration among homogeneous hosts is to furnish operating system modifications which embody the functions of the remote execution library.

These systems are useful for putting otherwise idle processor resources to productive use, but their scope is limited. Systems which employ operating system modifications are costly and not portable. Any load sharing system which migrates an execution image will not be able to support heterogeneous environments. The alternative to moving an execution image is to develop a check-point restart protocol which also incurs high overhead. When non-preemptive scheduling is used, a placement decision is made for each process when it initiates. This policy reduces the autonomy of privately owned workstations. When a machine is hosting foreign tasks the owner must execute his or her own jobs remotely to obtain reasonable performance and avoid worsening the load imbalance.

These systems do not provide robust parallel processing facilitates or support the participation of high performance parallel platforms. There goal is to either to optimize placement of all work within a cluster or allow remote execution of batch job. Using idle resources for parallel processing is not supported.

1.2.4 Workstation Based Parallel Programming Environments

Workstation based parallel programming environments (Section 2.6.3) incorporate two elements; a parallel programming paradigm or language which allows an application to be divided into a set of tasks that can execute independently in parallel, and a run time system which schedules and dispatches the tasks on idle workstations. These systems have goals which are similar to the VCE's. That is, they seek to use idle computing resources to run parallel programs. However the VCE design is more robust. Workstation based parallel programming environments tend to be sensitive to the granularity of the applications being run. If a task of an application is placed on a machine, and the load on the machine increases, it may have to be restarted on a different host. This means the tasks must be large enough to not be slowed by excessive communication overhead and small enough to complete before being preempted. Applications that do not meet this criteria will not perform well. These systems have the further drawback that applications must be specially designed and implemented for them, again requiring the designer to be expert in his or her problem domain and in parallel programming as well. Facilitates are not included to exploit multiple parallel computers.

1.2.5 High Performance Application Development Tools

Section 2.7 will discuss tools whose primary intent is to ease the task of developing programs to run on parallel computers. These include languages like High Performance Fortran and Mentat, which enhance existing languages by adding capabilities to start, communicate with, and manage parallel threads of execution. Another approach to parallel program development is to provide a set of highly optimized template programs which can be customized for a particular application. The goal of these technologies is to permit the application developer to realize good performance without having to be concerned with as many low level details as are required by message passing systems or other parallel programming environments. Unfortunately, these application development tools still require application developers to determine

how their problem can be partitioned among a set of processors. Automatic load balancing is not provided to any significant measure. Facilitates to manage tasks on multiple parallel machines simultaneously are not included.

1.3 The VCE Approach

The scope of the Virtual Computing Environment is broader than the distributed system technologies described in the previous section. Those systems generally address one issue such as load balancing, remote execution, or parallel program development. None of them is geared towards facilitating the use of multiple high performance computers in grand challenge applications like the one described in Section 1.1.2. The VCE allows scientists and engineers to easily construct applications from existing components and automatically execute them on any local or remote computing systems that are available at run time.

1.3.1 Structure of the VCE

In order to make development and deployment of very large high performance applications more feasible, tools are needed to ease their development and automate their execution. The VCE, therefore, is divided into two major components: a *software development module* which presents a suite of simple, graphical tools for application definition and development, and an *execution module*, which is the run time system that supports execution of VCE applications. Because high performance computing is a young discipline, The VCE uses a modular design so that as future design and development tools and hardware architectures become available they can be incorporated into the computing environment efficiently.

The current VCE design focuses on the capabilities required of the software development tools rather than detailing specific tools. The design does include a detailed specification of the interface between the software development module and the execution module. This *application description framework*, a concise syntax for describing

the characteristics of a distributed application, allows development tools to be selected and interchanged without impacting the run time system (Chapter 4).

The VCE design does address all key issues of the execution module in some detail. The overall structure of the VCE is shown in Figure 2. The tools of the software development module produce application description files, task description files, and executable files. Each machine that is participating in the VCE runs a VCE scheduling/dispatching daemon as a background process. Machines that are architecturally similar are organized as groups. One member of each group is designated as the group leader. When a machine has excess capacity (according to criteria discussed in Section 5.1) it notifies its group leader, which maintains a list of group members that are available for use by VCE applications. Execution of a VCE application is managed by a *VCE execution process*. The execution process is initiated by a user on his or her workstation and is passed the name of an application description file produced by the software development module. The application description file refers to task description files and executable files. The execution process uses the information in these files to determine the characteristics of the application. It also communicates with the group leader daemons that represent groups which are good candidates to host portions of the application. These group leaders return indicators of the amount of idle capacity available within their groups. Once the execution process has parsed the application and task description files, and received information about the run time load conditions from selected group leaders, it can proceed to make placement decisions about the application's components. The benefits of this approach are explored in Chapter 3.

In designing the execution process different scheduling policies have been investigated. Among these are random scheduling, selective scheduling (where only lightly hosts are considered) and redundant scheduling. As indicated by the performance data in Table 1, often an application component will have similar performance characteristics on several platforms. If several types of machines are available as candidates for a portion of an application, the execution process may dispatch several instances of the task in order to maximize the chances that superior performance will

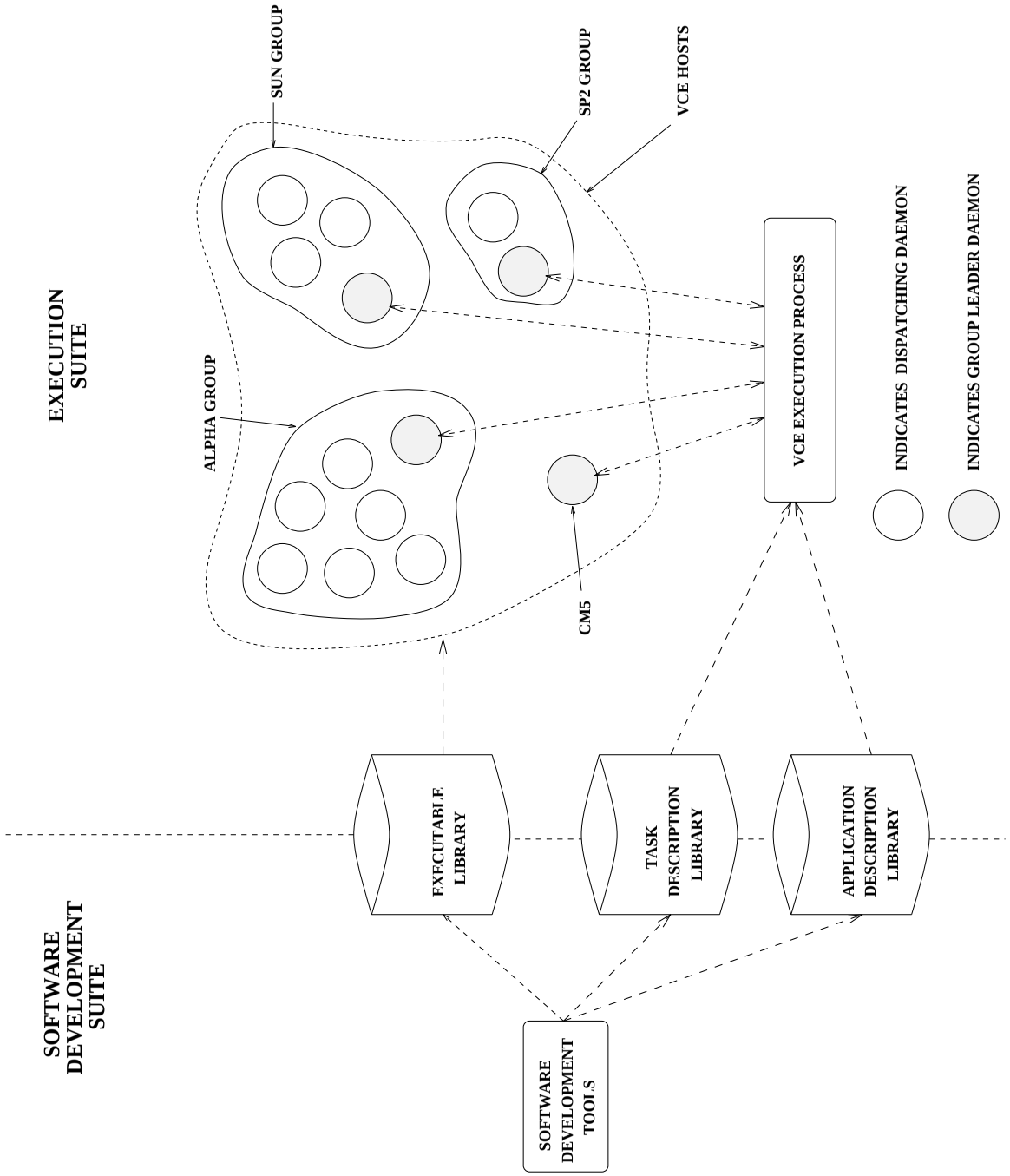


Figure 2: Major Components of the VCE

be achieved in spite of fluctuations in load conditions. Details of the mechanisms of redundant scheduling are included in Section 5.1.

Several prototype and simulated VCE systems have been constructed in order to determine the main characteristics of the VCE run time system. Experimentation has focused on compareing the VCE's redundant scheduling techniques to more conventional selective scheduling methods. Strong evidence of the advantages of redundant scheduling has been compiled. Experimental validation of the efficency of organizing host machines into groups has also been obtained. Further, a prototype implementation of the application description framework has demonstrated its utility and ease of implementation. Descriptions of the prototypes, simulations, experimental observations and analysis are included in Chapter 6

The VCE execution module also provides an infrastructure to support communication between application components (Section 5.2). This communication system makes redundant scheduling and remote execution transparent to application developers.

1.3.2 Contributions

The VCE represents a unique and valuable step forward in the design of distributed computing environments. Previous distributed computing systems have focused on load sharing (Section 2.6), easy construction of scientific applications (Section 2.7), development and execution of real time applications (Section 2.8) or support of distributed objects (Section 2.9). The VCE is a more comprehensive system. It includes a software design and development environment geared towards the creation of very large applications by scientists and engineers. The software design suite can accommodate procedural, object oriented or World Wide Web based development tools and can be extended to support new design paradigms in the future. The VCE execution environment is one of the first attempts to automatically manage distributed applications that require multiple high performance parallel computers. Advanced load measurement and scheduling techniques have been constructed for the VCE execution

environment which allow applications to achieve good performance by exploiting the excess capacity of networked high performance machines.

The contributions of current VCE research can be summarized as:

- The design of an extensible software development environment that separates program design (prograaming-in-the-small) from application design (programming-in-the-large).
- The definition of an application description framework which constitutes the interface between the development and execution environments.
- The design and validation of a set of technology independent scheduling, dispatching and communication methods to manage execution of applications across multiple parallel systems.
- Incorporation of the above elements into a coherent Virtual Computing Environment.

1.4 Organization of the Dissertation

The remainder of the dissertation presents details of the VCE design, related background material and descriptions of experimental VCE prototyping and simulation efforts. The material is organized as follows;

Chapter 2 discusses a broad range of distributed computing environments. Examples of RPC and messaging passing systems, workstation based load sharing systems, workstation based parallel computing systems and parallel programming environments will be discussed and compared to the VCE. Distributed application description languages and distributed object systems, which share some of the capabilities of the VCE application description framework, are also discussed.

Chapter 3 contains a detailed summary of the VCE design. The role and functions of all the VCE components and their interfaces are outlined.

Chapter 4 presents the VCE application model. The structure of VCE applications and their construction from reusable task and object implementations is discussed. The focus of Chapter 4 is a detailed description of the VCE application framework and comparison of it to other popular application description methods.

in Chapter 5 the techniques used to manage application scheduling and communication, which underlie the VCE execution suite, are discussed. Load measurement techniques, redundant scheduling methods and techniques for managing communication among redundant task instances are described.

Chapter 6 describes two prototype VCE systems and an object oriented simulation model that were used to validate the VCE design. Experience with VCE applications, the application description framework and various scheduling techniques are outlined.

Finally, Chapter 7 contains conclusions and possible future VCE research directions.

Chapter 2

Distributed Computing Environments

The previous chapter outlined the two computer engineering problems which are most relevant to the current work; difficulty in developing and executing Grand Challenge parallel and distributed applications, and the large amount of underutilized processing capacity present in networked computing environments. Before describing in detail how these problems are addressed by the VCE, previous approaches to solving them will be reviewed. This discussion will focus on how distributed computing has evolved. Beginning with a set of functions incorporated into general purpose operating systems that allowed programs on different hosts to communicate using I/O semantics, up to special purpose tools that ease the implementation of distributed fault tolerant systems, a trend towards more specialized and powerful tools will be evident. Finally, once current distributed systems technology has been examined, the roll of the VCE in addressing the needs of high performance distributed computing will be explored.

2.1 Basic Distributed Computing Tools

A distributed application is any one that involves processes on different computers communicating in real time. The most basic way to accomplish this is to connect

machines with network hardware and add functions to the operating systems which implement a network protocol [96]. Applications can then direct the operating system to perform basic operations like initiating or terminating a connection between two processes and transmitting data between them. This model of distributed computation is too limited for many applications. In this scheme the application programs are responsible for locating each other (addressing information may be hardcoded into programs or read from files). They may also be required to detect and recover from network errors and perform data conversion when host machines use different data representations.

Developing and managing distributed applications using only the functionality provided by a general purpose operating system is difficult [89]. The programmer must be familiar with the communication protocols being used. Porting programs is problematic because particulars of the operating system, hardware data formats and possibly machine address are incorporated into the programs. Further, all these issues are unrelated to, and a distraction from, the function of the application.

2.2 RPC and the Client Server Model

As stand alone microprocessor based workstations replaced mainframes as the most common computing environment, the need for distributed computing technology became critical. Facilities like file systems, peripheral equipment and data bases had to be shared among users. Resources such as fast or special purpose processors also had to be available to remote users. The practice of providing such facilities to all possible users in a dedicated mode would be inefficient and prohibitively expensive. Further, these issues needed to be addressed by a technology that would not make it unreasonably difficult to upgrade operating systems, add machines, or use hardware and software from multiple vendors.

These requirements led to the development of the *client server* computing model and the *remote procedure call (RPC)* communication paradigm [89]. Since the goal was to replace awkward operating system based communication semantics, a logical

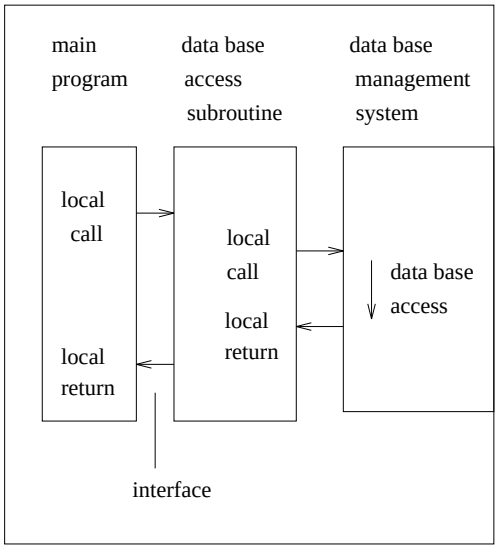


Figure 3: A Conventional Data Base Application

approach was to replace it with procedure call semantics which programmers already used for communication between non-distributed application components (that is, modules).

An example of the use of an RPC system is shown in Figures 3 and 4. In a conventional environment (Figure 3) a well structured application would call a procedure to access information in a data base management system. All the processing required to interface with the data base would be encapsulated in the procedure so that only the subroutine would be effected if the data base management system were modified or replaced.. A client/server computing environment would provide stubs and a run time RPC library to permit remote access to a shared data base (Figure 4). Stubs are code fragments which represent the interface of a remote procedure. The client stub receives a set of parameters from the calling program. The arguments are passed to the RPC run time library which transmits them over the network to the RPC run time library on the machine where the remote procedure is located. On that machine, a server stub is used to invoke the procedure which accesses the data base and returns a result by way of the stubs and RPC run time libraries. Stub code is usually kept

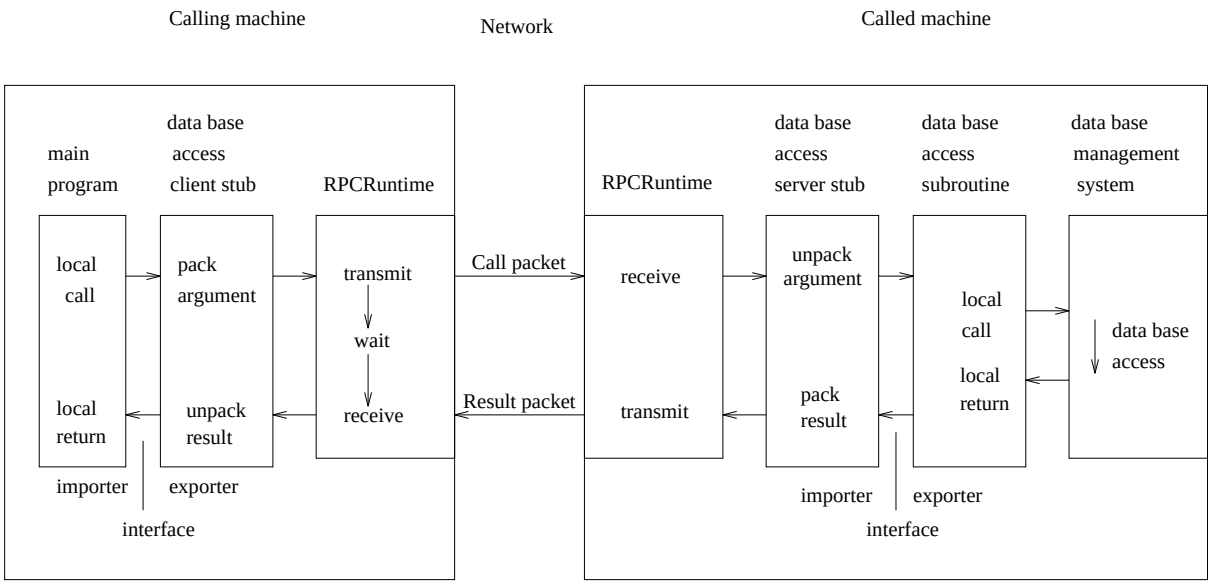


Figure 4: A Distributed Data Base Application Using RPC

in a common repository [89]. When the server is compiled its stub is *exported* to the repository where it is later *imported* into client modules.

Distributed application development is eased considerably by RPC facilities. Once the interface of the remote procedure is determined, a special compiler is used to generate the stub code. The procedure and calling program can be coded as if they were to run in a non-distributed environment, but at compile time they are linked with the stub code rather than each other. The RPC run time library manages the actual communication operations. The RPC system handles data conversion, addressing (server location), security, user authentication, network access and most error recovery in a way that is transparent to the application.

The client/server computing model is adequate for many distributed applications. As in the example above, it is easy to construct remote procedures (servers) which can be invoked by programs (clients) on any number of machines. This model is suitable for providing remote access to shared resources [89]. Developing client/server applications is straight forward because most communication issues are handled by the RPC run time library.

2.3 Message Passing Systems

The client/server computing model is not well suited to supporting parallel processing. The remote procedure call protocol is inherently synchronous [89]. That is, the calling program sends a set of parameters to a remote procedure and waits for a result to be returned. Also, it is assumed that server processes will be instantiated on machines that contain shared resources and service client requests as they are received. Facilities for applications to dynamically start and terminate tasks and broadcast parameters to, and collect responses from, multiple remote processes are not provided.

Message Passing tools represent a similar level of function and ease of use to RPC systems but they facilitate development of parallel distributed applications. Examples of message passing tools are Express [83], P4 [22] and PVM [48, 99]. These tools commonly provide functions to start tasks on remote machines, determine the addresses of parent, child and sibling processes and send, broadcast and receive messages. Both synchronous and asynchronous versions of the send and receive functions are usually included. These tools are very flexible and allow the programmer to realize a variety of parallel algorithms in an intuitive manner.

2.4 Distributed Shared Memory Environments

Distributed shared memory tools also seek to present an abstract communication model that will be useful for developing parallel applications. The most widely available distributed shared memory environment is Linda [23, 25]. The Linda system is built around a logically shared memory structure called a *tuple space*. Functions are provided to insert data structures in, and retrieve data structures from, a tuple space and start remote processes. A common way to structure a Linda application is to have a master process build a set of data structures which describe subdomains of a large problem and insert them into the tuple space. Then a group of slave processes are started which retrieve these data structures, perform a calculation and place the result in the tuple space. The master process can use a function which retrieves an

element from the tuple space but blocks if no suitable element is available. By using this function to retrieve result data structures from the tuple space as they are computed, the master process can monitor the progress of the application.

Communication via distributed shared memory is more flexible and intuitive than message passing [76]. In the example of the previous paragraph, if some slave processes are on faster machines than others, they can simply consume more subdomain data structures and produce more results. This is possible because a specific receiver does not have to be designated for each element being communicated, as is required for message passing. The main limitation of distributed shared memory systems are related to implementability and scalability [25].

2.5 Distributed Operating Systems

All the systems described above (RPC, message passing and distributed shared memory) seek to ease the development of distributed applications. They do not emphasize issues of load balancing. The goal of distributed operating systems is to entirely mask the complexities of distributed systems by presenting users and application programmers with the image of one large hardware processor. The same operating system kernel is placed on every machine in a distributed system. The machines can then cooperate to detect load imbalances and migrate processes as needed.

Examples of distributed operating systems which have been documented in the literature include V [30], Amoeba [78], the x-Kernel [85] and LOCUS [105]. The shortcomings of these systems were discussed in Section 1.2.1. They include excessive cost, size and complexity. Distributed operating systems often infringe on (or eliminate) the autonomy of workstation users. The range of hardware supported by such systems, and the ability to accommodate new hardware, is limited. None of them supports supercomputing platforms or emphasizes execution of parallel programs.

2.6 Load Sharing Environments

Many systems which seek to exploit idle workstation resources have been reported over the last 10 years. While some of them rely on operating system modifications to achieve some functions, these kernel modifications are generally far less extensive then are required by the distributed operating systems mentioned above. The goals of these systems are to use the processors of idle machines to run parallel programs, batch jobs or ordinary tasks for users on more constrained processors. Brief descriptions of some of the more interesting types of load sharing environments are included here.

2.6.1 Load Sharing on Homogeneous Workstation Clusters

One of the most successful and widely used load sharing systems is Condor [21]. Condor allows users to run CPU-bound batch jobs on idle workstations. The jobs are compiled using Condor's own library instead of the Unix Library. When a remotely executing Condor job issues a system call it is handled by the Condor run time library on the host machine rather than the machine's operating system. The system call is forwarded to a "shadow process" running on the originating machine which performs the call and forwards the results back to the remote process. System calls are the only way a Condor program can interact with its environment. Signals cannot be used. When the owner of a workstation hosting a Condor remote job resumes use of the machine, a Condor daemon checkpoints the job and returns it to the originating machine so it can be placed on another host [71].

Similar services are provided by the MOSIX system [9]. Although the designers of MOSIX call it a distributed operating system, its scope is more limited than the systems mentioned in Section 2.5. The objectives of MOSIX are to provide remote execution, process migration and load balancing within a homogeneous workstation cluster [10].

The MOSIX kernel runs on all the workstations in a cluster and provides a high degree of location transparency. The kernel is structured as shown in Figure 5. The "upper kernel" interfaces with user processes. The "linker" determines if requested

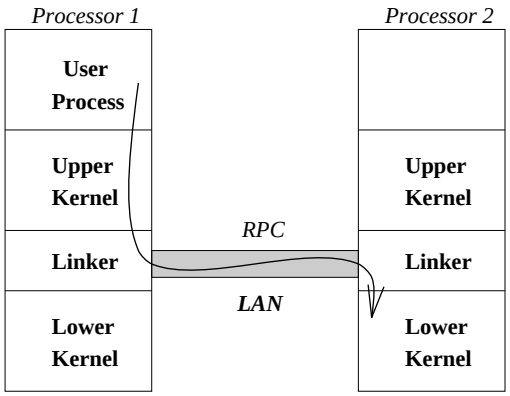


Figure 5: Location Transparency in MOSIX [9]

functions can be performed locally and communicates with the "linker" on remote machines if access to remote resources are needed. The "lower kernel" performs the actual operating system services. This structure allows Unix processes to be run under MOSIX without restrictions. Signals as well as system calls are received by the "upper kernel" and resolved transparently by the "linker ".

The MOSIX structure is similar to Condor's except that Condor, which executes outside of the operating system, is more restricted than MOSIX. The functions provided by the MOSIX "upper kernel" and "linker" are analogous to Condor's run time library and "shadow process".

The MOSIX load balancing scheme is rather sophisticated [9] [10]. Nodes disclose their loads at regular intervals to randomly selected neighbors. At any moment each node has knowledge of the loads of a subset of the cluster. If a node is more heavily loaded than the nodes it is aware of, it will migrate a process. Only long running processes are migrated. If a candidate process is communicating with a remote machine, and that machine is less busy that its current host, it will be the target for the migration. Alternatively, another lightly loaded node will be chosen. The algorithm includes a mechanism to prevent thrashing.

The Stealth Distributed Scheduler [67] extends the Mach operating system to provide functions similar to those provided for Unix by MOSIX and Condor. When

tasks are started on overloaded Stealth hosts they are migrated (non-preemptively) to idle machines an run at low priority. If local load on a host increases and causes guest processes to be starved they are migrated again (preemptively).

2.6.2 Load Sharing Among Heterogeneous Workstation Clusters

The Utopia system [109] supports load sharing among large heterogeneous workstation networks (as large as several thousand machines). No operating system changes are made. The workstations are grouped as clusters. For most remote executions the initiating machine and the remote host will be in the same cluster. Specially written "load sharing" applications exploit remote execution. A load sharing application can be written to establish a remote context for another application, thus allowing existing applications to be executed remotely without modification. Users supply *remote execution lists* which describe the attributes and requirements of their load sharing applications. A relatively complex scheduling algorithm uses the attributes to find suitable hosts for applications. Task migration is not supported. The system designers do not consider this a problem because users of busy machines hosting foreign tasks can execute their own jobs remotely on lightly loaded computers.

2.6.3 Workstation Cluster Based Parallel Processing

Three similar systems which use idle workstations to execute parallel applications are Piranha [24], Remora [87] and Phish [20]. Piranha and Remora are implemented using the Linda distributed shared memory model [23]. Piranha requires applications to be structured as three components: a *feeder()*, a *piranha()* and a *retreat()* function. The feeder() divides the problem into tasks and builds a data structure (called a *tuple*) that describes each task. The piranha() is initiated on idle workstations and consumes tuples and returns results to the feeder(). If a workstation is reclaimed by its owner while it is running a pirahna() the retreat() function is called to return the tuple it was working on to the pool of available tuples where it can be claimed by another

workstation and restarted from the beginning. The Piranha design is sensitive to the granularity of the applications being performed [87]. If the decomposition is too small system overhead will impede performance. If the granularity is too large the proportion of tasks which are preempted and must be restarted will become high and prevent the application from making progress. Remora is less sensitive to application granularity because a checkpointing mechanism is provided so that preempted tuples do not have to restart from scratch. Both Piranha and Remora implement "adaptive parallelism", meaning that the amount of parallelism within an application will change as idle workstations join or leave the computation.

Phish provides capabilities similar to Remora. Additionally, it has a more sophisticated scheduler. When there are several Phish applications running and many workstations available the system will attempt to optimize the number of workstations assigned to each application. One PhishJobQ is maintained in a cluster to track the Phish applications. A PhishJobManager runs on each host and requests work from the PhishJobQ if its host is idle. Phish jobs can request or release machines dynamically.

Figure 6 shows a Phish cluster. Shaded nodes indicate that the node is not idle and not eligible to host a portion of a Phish application. Each idle node in the figure is assigned to one of the three available Phish jobs.

A different approach to cluster based parallel processing is taken by Global Layer Unix (GLUnix) [6]. GLUnix uses kernel modifications to enable idle machines in a homogeneous workstation cluster to be used to simulate a MIMD supercomputer. The memory contents of an idle machine are swapped out and the node is made available for parallel processing. A user can submit a parallel application that requires up to the number of available nodes. When a workstation user reclaims a node any parallel process using the machine is swapped out and the original memory contents are returned. The parallel application will be relocated to a different idle node if one is available. Otherwise a node failure exception is reflected to the application.

Finally, the Sidle system [60] provides a special Prolog interpreter which automatically spawns tasks on idle or nearly idle workstations. Tasks are killed and restarted

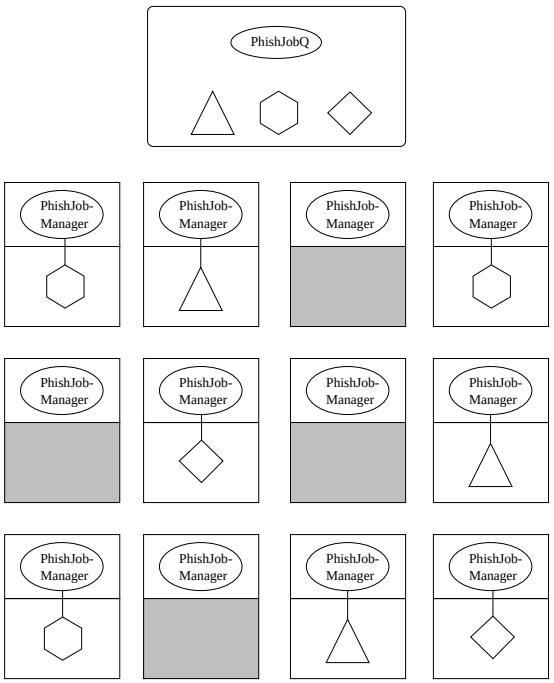


Figure 6: Components of the Phish System [20]

elsewhere if a workstation owner reclaims the machine.

2.7 Parallel Program Development Tools

As discussed in Section 1.1.1, scientists and engineers not only need computing environments that run fast, they also need environments that make it easy to construct high performance applications. The most common approach to building parallel programs involves linking a conventional programming language with one of the message passing or distributed shared memory systems mentioned earlier. These systems have the drawback that they require the programmer to figure out an efficient way to parallelize his or her application.

Parallel languages like High Performance Fortran [46] or the Mentat Programming Language [53] (which is an extension of C++ [97]) provide more intuitive means for expressing application parallelism. However, it is still necessary for the programmer to structure the application with considerable care in order to achieve good results [7] [52].

A more reasonable approach to high performance programming is taken by the POET system [7]. POET is intended for use by chemists and other physical scientists. Parallel programming experts develop program templates for common application. Scientists can then customize the templates to match the problem of interest to them. As long as a scientist can find a template which is reasonably close to his or her application, a sound, efficient parallel program can be built quite easily. Over time, as more templates are developed, the system will become more useful.

Another class of tools which are popular for developing scientific applications are *modular application builders* [73, 103]. An example of such a tool is the Advanced Visualization System (AVS) [2]. AVS, which will be discussed in more detail in Section 3.2, is a software engineering environment which supports the construction of scientific applications from AVS supplied or user written modules using a graphical user interface. The user selects modules from a menu, places them on a palette, and draws connections indicating how the output of one module is to be supplied

as input for another. Users can indicate which host each module is to run on. The AVS environment is intuitive, powerful and user friendly. The current design of the VCE software development suite envisions a front end similar to AVS (see Chapter 3). The primary limitation of AVS is that it is workstation oriented and provides only limited support for parallel computing platforms [103]. Further limitations are that it does not provide dynamic scheduling or support for an object oriented programming model [103].

2.8 Distributed Application Description Languages

The systems discussed thus far have been oriented either towards easing the development and execution of high performance applications or making more efficient use of distributed systems through load sharing. Another area of active research in distributed computing involves technologies to create and manage real time, embedded or fault tolerant systems. High performance computing environments stress effeciency and ease of use. Systems that support real time applications are oriented more towards providing expressive and intuitive methods for developers to specify complex application timing behavior and recovery scenarios. These are the objectives of distributed application description languages and their associated run time environments. In Chapter 4 it will be seen that the VCE application description framework is similiar to these languages but its syntax is simplified to support the more straight forward requirements of high performance distributed applications.

Application description languages [35, 86] have long been used as a tool in engineering large software systems. Individual modules of such a system are implemented using standard programming languages. The application description language specifies how the modules are linked and how data flows between them. Use of an application description language facilitates the use of a variety of sound software engineering practices such as stepwise refinement, function encapsulation and component reuse.

The goals of distributed application description languages are similar, but in addition to describing application components and their associated data flows, a syntax

for specifying how software components are to be assigned to processors must also be provided. Two research projects which have saught to implement such languages are Conic [63, 64, 65] and Durra [12, 13]. Conic provides two tools used to build distributed systems [64]: the Conic Programming Language (CPL), which is a superset of Pascal with message passing support, and the Conic Configuration Language, which is used to specify how a group of CPL programs are interconnected in an application. Rather than implement an extended programming language to facilitate interprocess communication, Durra uses a library of routines to allow programs written in C or Ada to communicate with each other and with the Durra run time system. the Durra application description language is similar in syntax, scope and function to the Conic Configuration Language.. In order to illustrate the functionality of a distributed application description language the next section outlines the Durra language and run time environment. Durra and Conic are (for the purposes of this discussion) quite similar, so describing both is not necessary. Durra was chosen because it is slightly more flexible than Conic, it was developed later, and it is closer in goals and implementation to the VCE application description framework.

2.8.1 Durra System Overview

Durra is a system for developing, implementing and executing distributed applications, particularly real-time embedded applications for areas such as robotics and avionics [12]. Durra applications are executed in a loosely-coupled network of special- and general-purpose processors. In Durra such a hardware configuration is referred to as a *heterogeneous machine*. Durra applications are constructed from task implementations (programs). Each task implementation has a corresponding task description written in the Durra distributed application description language. The Durra language is also used in an application description file to indicate how the tasks are to be dispatched and interconnected. Durra uses the application and task descriptions to build an *executive* for the application. One component of the executive runs on each machine participating in the application to load task executables and support

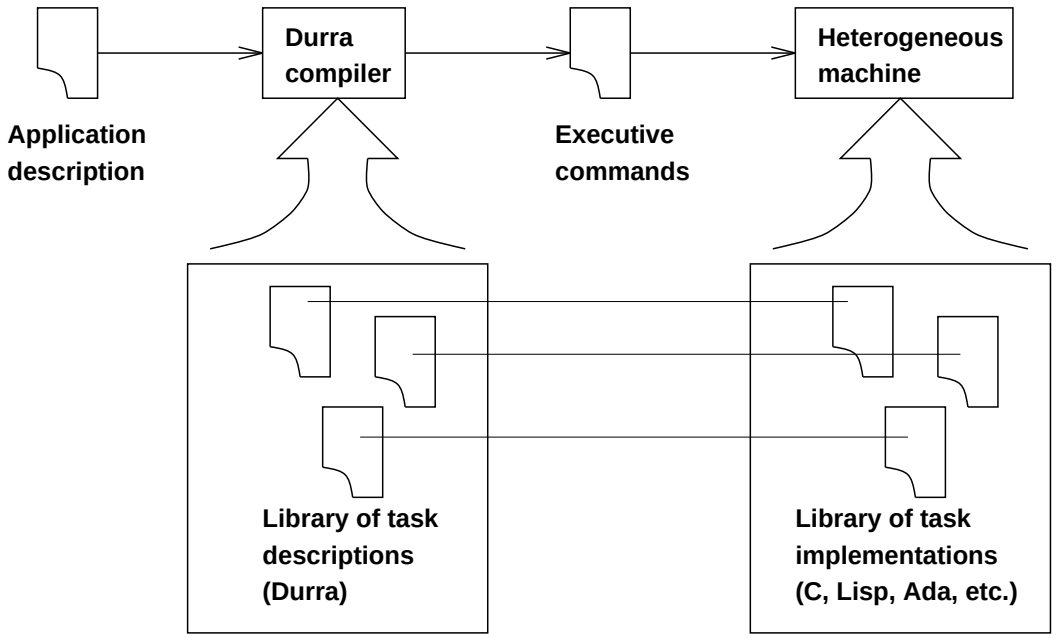


Figure 7: Elements of the Durra System [12]

communication and dynamic reconfiguration. This process is illustrated in Figure 7.

2.8.2 Durra Tasks and Applications

Figure 8 shows examples of two simple Durra task descriptions. Durra includes library functions which allow tasks to write data to, and read data from, Durra **ports**. The task description is used to declare to the Durra executive the types and direction of data that will flow through a task's ports. In addition to declaring ports, the task description also declares **attributes**, which include the file name of a corresponding executable file, the library where the executable resides and the type of processor the executable runs on.

Communication functions in Durra applications are performed by **channels**. A channel description, which is also illustrated in Figure 8, uses the same syntax as a task description to describe the properties of an Ada package implementing a communication facility. The channel description of Figure 8 also demonstrates that the

```
task  producer
  ports
     output:  out message;
  attributes
      processor = "sun4";
      procedure_name = "producer";
      library = "/usr/durra/srclib";
end  producer;

task  consumer
  ports
     input:  in  message;
  attributes
      processor = "sun4";
      procedure_name = "consumer";
      library = "/usr/durra/srclib";
end  consumer;

channel  fifo(msg_type: identifier, buffer_size: integer)
  ports
     input:  in  message;
     output:  out message;
  attributes
      processor = "sun4";
      bound = buffer_size;
      package_name = "fifo_channel";
      library = "/usr/durra/channels";
end  fifo;
```

Figure 8: Durra Simple Task and Channel Descriptions [13]

```
task dynamic_producer_consumer
  components
    p: task producer;
    c[1..2]: task consumer;
    buffer: channel fifo(message, 10);
  structurs
     L1: begin
         baseline p, c[1], buffer;
         buffer: p.output >> c[1].input;
       end L1;
     L2: begin
         baseline p, c[1], buffer;
         buffer: p.output >> c[2].input;
       end L2;
  reconfigurations
    enter -> L1;
    L1 -> L2 when signal(c[1],1);
  clusters
    cl1: p, buffer;
    cl2: c[1], c[2];
end dynamic_producer_consumer;
```

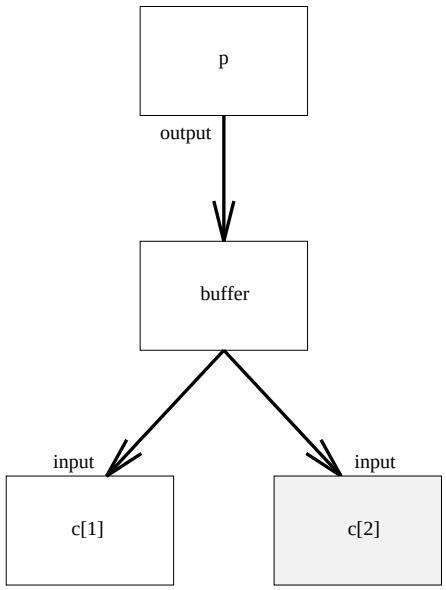


Figure 9: Durra Producer-Consumer Application with Dynamic Reconfiguration [13]

Durra language supports parameterized entities. When the `fifo` channel is used in an application description the *message_type* and *buffer_size* parameters must be specified. The Durra function library provides mechanisms for the task to obtain parameter values from the executive at run time.

Use of these task and channel descriptions to construct a producer-consumer application is shown in Figure 9. A Durra application is expressed as a *compound task* which permits hierarchical application development. The **components** section lists all the tasks and channels which will be included in the application. In this case there is one instance of a `producer` task, two instances of the `consumer` task and a `fifo` channel. The **structures** section lists all the ways in which data may flow among the application components. Within a **structure** the **baseline** statement lists the tasks and channels which constitute the structure. The other component of a **structure** specifies, for each channel, which task ports are connected to each channel port. Durra, which supports real-time fault tolerant systems, supports dynamic reconfiguration. The **reconfigurations** section of a compound task description indicates

when reconfigurations are to be performed. The **enter** clause identifies the initial configuration. Each subsequent line in the **reconfigruations** section gives a current structure, a next structure, and an event which is to trigger the reconfiguration. Finally, the **clusters** section details how tasks are to be assigned to processors. Each **clusters** statement includes a cluster label and a list of tasks to run on that cluster.

These examples illustrate the Durra application development methodology [13]. Tasks are designed to provide medium grain functions with well defined interfaces. An application is constructed by selecting or developing appropriate tasks, determining how they should interact, constructing reconfiguration procedures, and grouping tasks into course grain clusters and assigning them to hardware. Tasks provide encapsulation and reusability. However, developing an application is still quite complex. Partioning tasks into efficient clusters and enumerating reconfiguration scenarios requires considerable skill on the part of the application developer.

2.9 Distributed Object Systems

Most of the technologies discussed in this chapter are procedurally oreieted. For instance, the RPC approach described in Section 2.2 allows a program to call a procedure that is executed on a different machine; the message passing systems of Section 2.3 provide functions to send and receive data between machines; the distributed shared memory systems described in Section 2.4 include procedural interfaces to insert data in and remove data from shared storage. As object oriented methodologies have become common in all areas of computer engineering, efforts have begun to incorporate them into distributed computing environments. Current efforts in this area are analogous to the RPC systems mentioned earlier. That is, RPC environments allow programmers to invoke procedures on remote machines as if they were local. Similarly, distributed object systems seek to allow programs to use conventional object oriented semantics to transparently invoke operations on objects whose data and methods actually reside on other computers.

Because object oriented techniques for developing software are preferred by many

practioners, the VCE supports object oriented applications using interfaces based on the CORBA [82] standard for distributed object management systems. Currently most CORBA compliant systems are intended for commercial, rather than scientific, computing. An example of such a system, IBM's DSOM, is presented below.

2.9.1 DSOM System Overview

IBM's System Object Model/Distributed System Object Model (SOM/DSOM) [59] is an implementation of the CORBA standard for shared object systems [82]. Compliance with such a standard is intended to allow for interoperability between distributed object systems.

A simplified representation of the main components of DSOM are shown in Figure 10. For the sake of clarity, the nomenclature of the DSOM documentation [16] [59] is modified in this discussion. Applications which utilize distributed objects are referred to as *client applications*. Remote objects run within a *server process*. The server process maintains the object's persistent state information and performs operations on the object on behalf of the client. Client applications perform operations on an *object proxy* which has the same interface as the remote object. The object proxy forwards the parameters provided by the client to the server process, receives results back and returns them to the client. Each client application has an associated *object manager*. The object manager provides a variety of services to the client including creation and destruction of object proxies and location of server processes. Each machine which hosts server processes runs a *DSOM daemon* which collaborates with object managers to load and execute server processes as needed.

DSOM (and CORBA), use two files to hold information on the interfaces and implementations of distributed objects (Figure 11). The *interface repository* contains interface information for all distributed objects in the system. This file is not only used by the system but can also be used by application developers in selecting distributed objects to participate in an application. The *implementation repository* is used by object managers to locate server processes and by DSOM daemons to dispatch them.

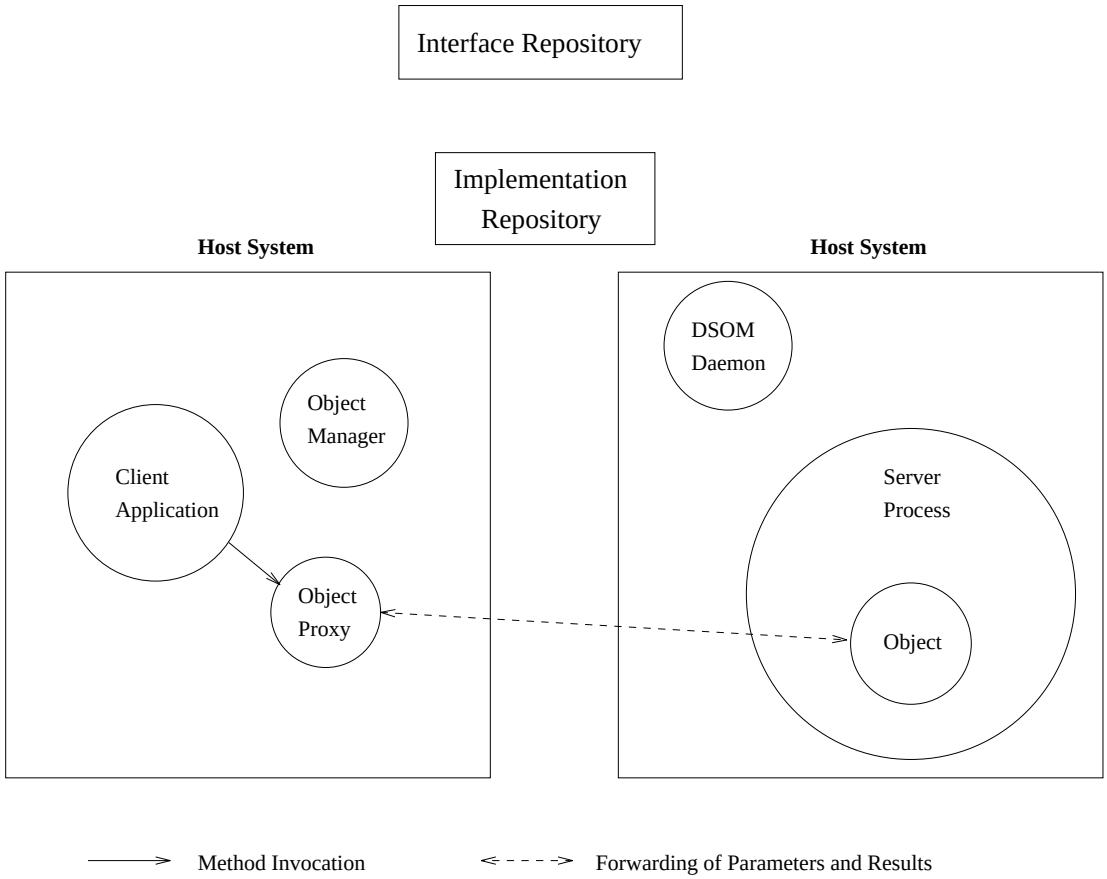
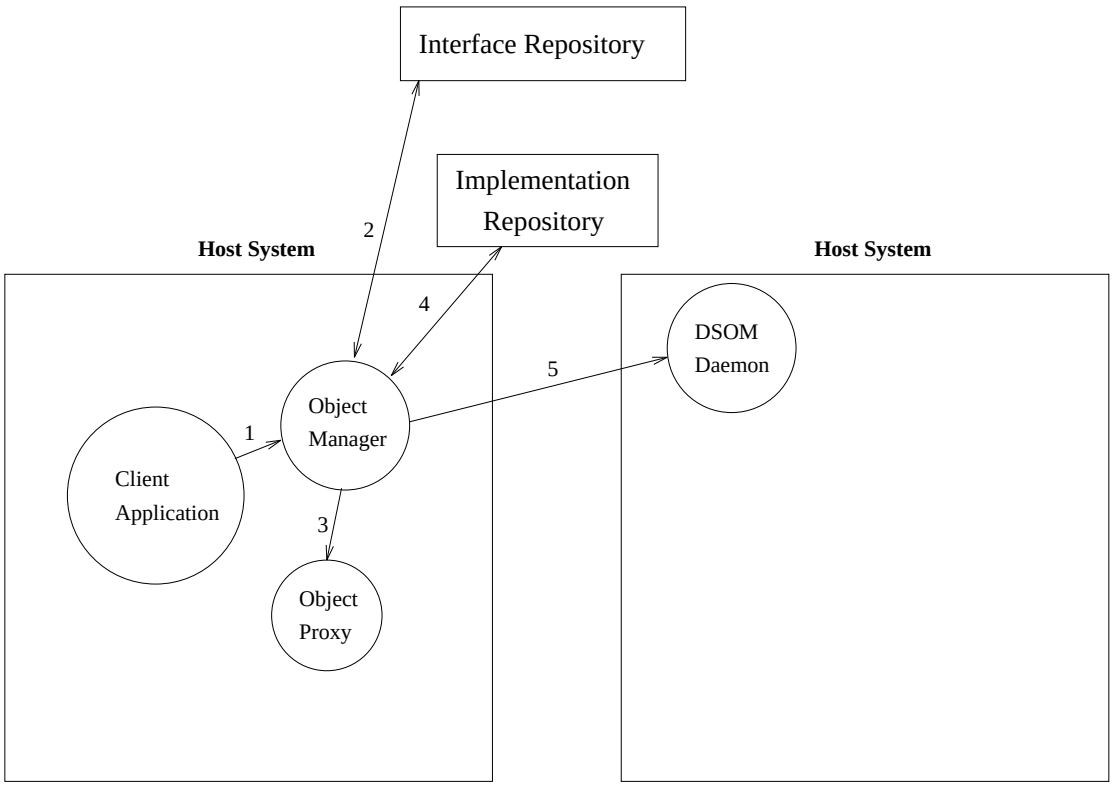


Figure 10: Components of the DSOM Distributed Object System

Interface Repository
One entry for each distributed class - Describes the interface for the class

Implementation Repository
One entry for each server process - Lists the classes implemented by the server - Indicates which host the server runs on - Gives the name and location of the executable for the server

Figure 11: DSOM Repository Files



1 - The client application requests access to a remote object

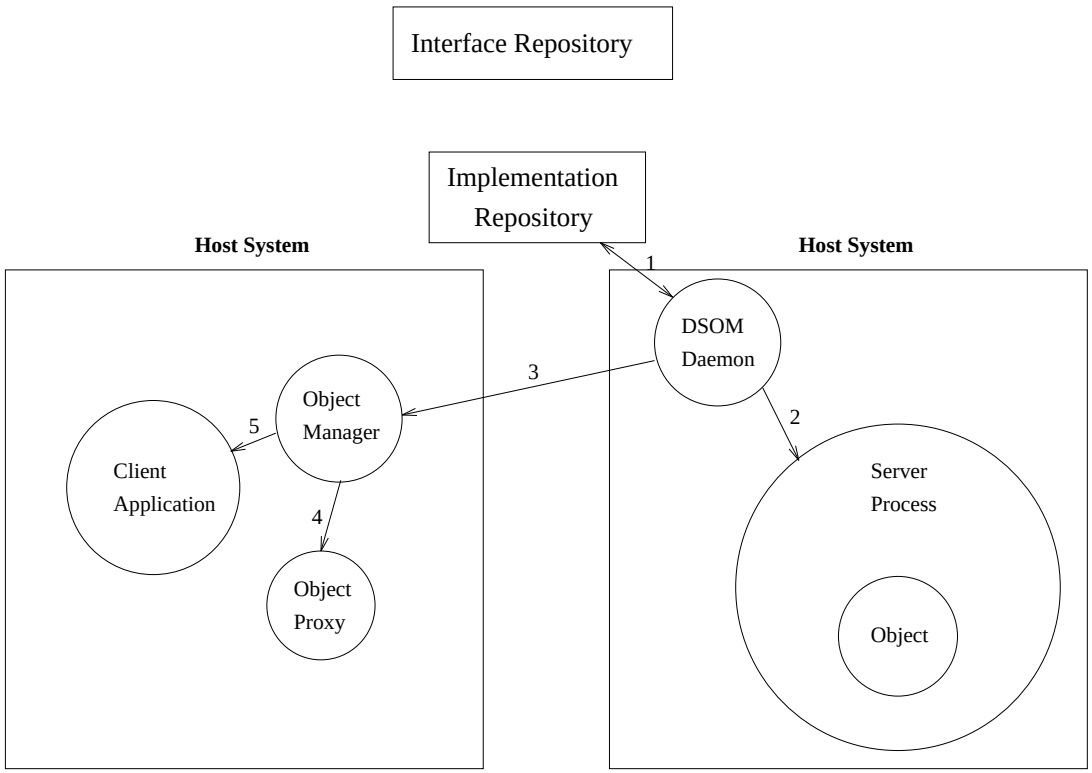
2 - The object manager fetches the object's interface

3 - The object manager constructs a proxy for the object

4 - The object manager locates the server process's host system

5 - The object manager contacts the server process's DSOM daemon

Figure 12: Initiating a DSOM Remote Object Access



1 - The DSOM daemon fetches the name of the executable

2 - The DSOM daemon starts the server process

3 - The DSOM daemon returns the address of the server process to the object manager

4 - The object manager give the address of the server process to the object proxy

5 - The object manager returns the address of the object proxy to the client application

Figure 13: Completing a DSOM Remote Object Access

Figures 12 and 13 show how all the DSOM components interact when a client program requests access to a distributed object for which no server process is running.

2.9.2 DSOM Application Development

As with RPC systems, the most common use for DSOM is to ease the development of programs which use remote shared resources. For instance, consider the problem described in Section 2.2 of sharing a single database management system among multiple machines. A data base server process could be used to provide remote data base objects. The data base object's interface would be registered in the interface repository, the server process would be included in the implementation repository, and a SOMD daemon would run on the database machine. Once this infrastructure was implemented, writing applications that run on any machine and use the database would be only slightly more complex than if the database where available on each machine.

2.10 High Performance Computing Over the World Wide Web

The World Wide Web is becoming a popular technology for providing access to remote data [51]. Users employ a *browser* to download and view documents from servers that contain text, images and multimedia. A set of special protocols have been developed to govern the interactions of the browsers (clients) and data servers. Through the use of hypertext links, complex documents can be constructed which incorporate existing data from any number of hosts.

Proposals are being developed to extend the World Wide Web model so it can be used to facilitate high performance distributed computing [44, 45]. The web is currently oriented to allowing users to access data on multiple remote servers. The proposed extensions involve a new set of protocols which will support web based compute servers as well as data servers. For example, a single high performance

parallel computer might be designated to perform a service like matrix multiplication. Applications would then be constructed by selecting the best remote server for each job step and linking them dynamically in much the same way data servers can now be linked to construct a complex document. These projects are currently in their very early stages. New protocols must be developed to support them.

The VCE will also exploit World Wide Web technology (Section 5.2) to dynamically locate hosts. Unlike the systems discussed in the previous paragraph, the VCE will use excess capacity on participating machines and redundant scheduling to achieve high performance. It will not require machines to be dedicated to providing specialized functions. In the future it is likely that VCE research and World Wid Web based high performance computing will benifit from and exploit results realized in both efforts.

2.11 Separating Coordination from Computation

Many of the systems discussed in this section have a common shortcoming. They require the application structure to be expressed by the same programs that perform the application's computations. From a software engineering perspective this is a step backwards. Large systems designers have long made it a practice to separate programming-in-the-large from programming-in-the-small [35]. A variety of methodologies and module interconnection languages [86] have been developed based on this practice. In scientific programming the problem can be framed in terms of a need to separate coordination from computation [49].

2.12 The VCE Approach

This chapter has described a wide variety of distributed computing environments. Each has different objectives. These objectives have evolved over time. As the capabilities of distributed computing hardware have increased, more sophisticated environments to exploit those capabilities have been introduced. Currently, more capable

network hardware is being deployed which will allow large parallel computing systems to exchange data at very high rates. This hardware capability will make it feasible to utilize multiple high performance computing platforms in a single application. The VCE constitutes one of the first distributed computing environments which will ease the development of these large scale high performance applications.

Many of the remote execution environments discussed in Section 2.6 seek to harness the unused capacity in idle workstations and make it available to parallel or long running applications. The goals of the VCE are similar but more ambitious. None of the systems mentioned in Section 2.6 can accommodate multiple supercomputers. Most are workstation based systems. The VCE will provide tools to manage execution of application tasks on remote supercomputers and workstation clusters. The redundant scheduling methods discussed in Section 5.1 and demonstrated experimentally through the prototyping and simulation discussed in Chapter 6 give the VCE the ability to provide high performance with a wide variety of hardware configurations and load conditions.

The VCE provides an application description framework which allows the details of an application's structure to be specified separately from its computations [49]. That is, computing procedures or objects can be implemented on whatever hardware at whatever granularity is appropriate. The VCE allows applications to be constructed from these procedures or objects without requiring the user to modify or even understand their implementation. Further, decisions about what hardware to use at run time are made by the VCE's scheduling system (Section 5.1) using hints and other information associated with each computing task. While some of these goals are shared with the application description languages described in Section 2.8, the emphasis on high performance computing and accommodation of object oriented methods is unique to the VCE.

Table 2 shows how the VCE compares to many of the high performance distributed computing and load sharing tools described in this chapter. The VCE is implemented completely above the operating system level. Operating system modifications add complexity and limit portability and heterogeneity. These drawbacks overshadow the

Table 2: Capabilities of HPC Environments

	CONDOR	MOSIX	GLUNIX	UTOPIA	MENTAT	STEALTH	SIDLE	PIRANHA	POET	VCE
USES KERNEL MODS	NO	YES	YES	NO	NO	YES	NO	NO	NO	NO
SUPPORTS PARALLEL APPLICATIONS	NO	YES	YES	NO	YES	NO	YES	YES	YES	YES
MINIMIZES IMPACT ON CONVENTIONAL WORKSTATION USERS	YES	NO	YES	NO	NA	NO	YES	YES	NA	YES
SUPPORTS HETEROGENEOUS ENVIRONMENTS	NO	NO	NO	YES	NO	NO	NO	NO	NO	YES
SUPPORTS SUPERCOMPUTERS	NO	NO	NO	NO	YES	NO	NO	YES	YES	YES
SUPPORTS MULTIPLE SUPERCOMPUTERS	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES
SEPARATES COORDINATION FROM COMPUTATION	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES

gains in performance and transparency which they allow.

Some remote execution systems simply place a job on an idle workstation and migrate it if the workstation owner reclaims the machine. These systems don't permit the remotely executing job to communicate with other tasks and therefore cannot support parallel applications. The utility of such systems is limited to providing a means for the user of a long running sequential job to run it without impeding the performance of other tasks on his or her workstation. The VCE provides benifits to a potentially larger class of users. Namely, those scientific and engineering users who desire to run parallel applications. When a configuration has multiple idle machines, and a user has an application that can be efficiently partitioned into multiple communicating tasks, the ability to dispatch each task on an idle host, as can be done by the VCE, is attractive.

Table 2 shows that many distributed computing environments are strictly oriented towards *distributed workstation* environments and do not allow users to utilize parallel computing systems. Other tools can only be used to implement or execute applications on a single supercomputer. The VCE is uniquely oriented towards selecting the optimal host for each portion of an application. The VCE choses hosts based not only on the properties of the task or user supplied specification, but also considers dynamic load conditions.

Finally, most other distributed environments do not provide a software engineering environment that makes it easy for scientists and engineers to design and implement large applications. Most require special programming techniques (such as language extensions) to achieve high performance. The VCE separates task design and construction form application design. Only the task implementer, presumably a high performance programming specialist, has to be concerned with the complexities of parallel programming. The application designer, often a scientist or engineer, simply uses a graphical interface to express how an application can be constructed from existing tasks. The application designer is not concerned with the details of how tasks are implemented or how machines are selected to run them.

Chapter 3

Overview of the VCE Architecture

The VCE is a suite of tools and technologies which support the development and execution of large distributed applications on networked high performance computing platforms [91]. This chapter will present a high level description of the VCE components and discuss their interfaces. The VCE design is modular and thus new tools and methodologies can be incorporated into the design over time. Some individual components of the system may have considerable value to users apart from their role in the larger VCE scheme. The modular design of the VCE allows subsets of its functionality to be deployed if other portions are not implemented or not needed in a particular situation.

3.1 VCE Building Blocks

The VCE design can be represented hierarchically by the components (building blocks) shown in Figure 14. At the highest level, the components of the VCE are divided into a *software development suite* and an *execution suite*. Application development and execution are largely independent activities except for areas such as problem determination and performance monitoring. The VCE design seeks to minimize the dependencies between the tools in the software development and execution suites. In the future new hardware architectures and development methodologies will

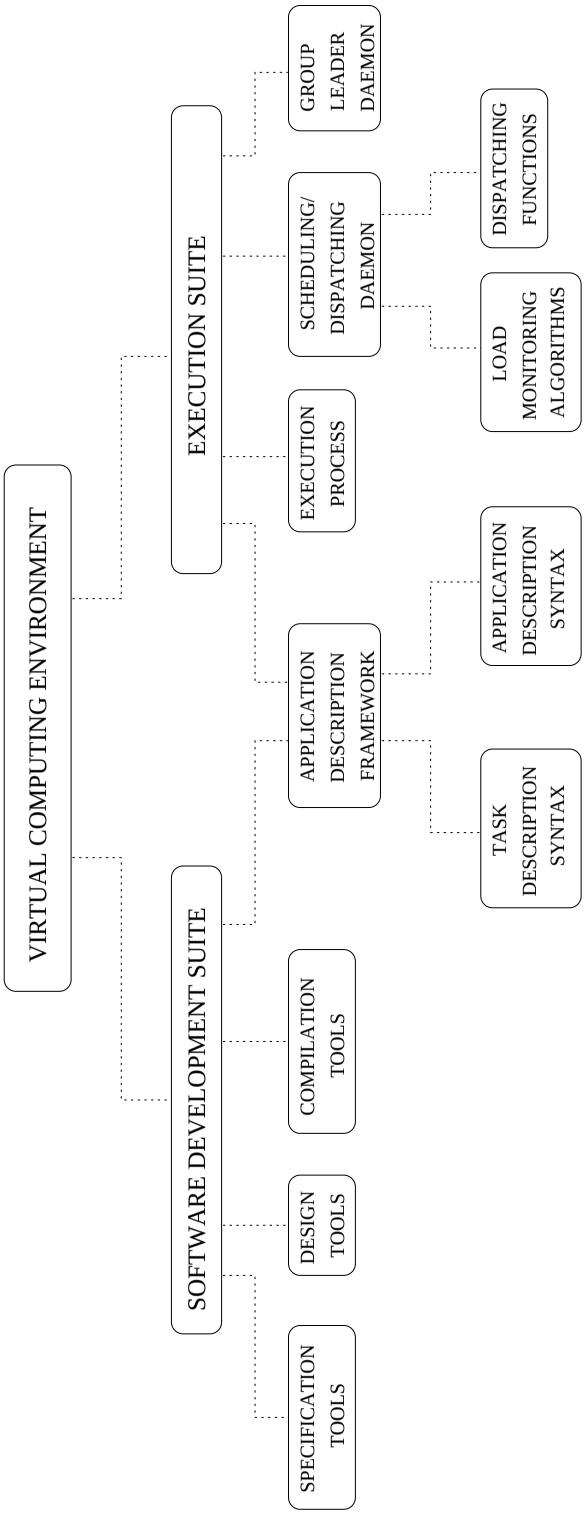


Figure 14: Structure of the VCE

inevitably arise. By constructing a general, extensible interface between the VCE development and execution environments, support for new software engineering tools and hardware platforms can be provided quickly with minimal effort and user disruption.

Figure 14 shows a software development suite that contains *specification tools*, *design tools* and *compilation tools*. This particular set of tools reflects a stepwise refinement approach to procedurally oriented application development. There are currently many mature procedurally oriented development tools (such as AVS [2]) which can easily be adopted to fill these roles in the VCE. These procedural tools can be replaced or augmented by object oriented tools which support object class development or class hierarchy development. Design tools specifically geared towards implementing high performance applications across multiple World Wide Web attached high performance computers [45] can be accommodated in the VCE software development suite as well. Section 3.2 will describe a VCE software development suite which uses an AVS-like graphical user interface. Chapters 4 and 5 discuss how object oriented and Web based functions can be supported by the VCE execution suite. The design of the software development suite is flexible and can be implemented using procedural, object oriented or Web techniques.

The *VCE Application Description Framework* is the primary interface between the VCE development environment and the execution environment. The tools of the software development suite produce application and task description files (Figure 2). These description files, along with executable and data files they refer to, are input to the execution suite. The description files, which use the syntax of the application description framework (Chapter 4), contain all the information the execution suite needs to dispatch the application's executables and manage its data files and intertask communications. Figures 2 and 14 both convey that the definition of the application description framework is an integral part of both the software development suite and the execution suite. The characteristics of an application are captured by the software development suite and communicated to the execution suite using the application description framework syntax.

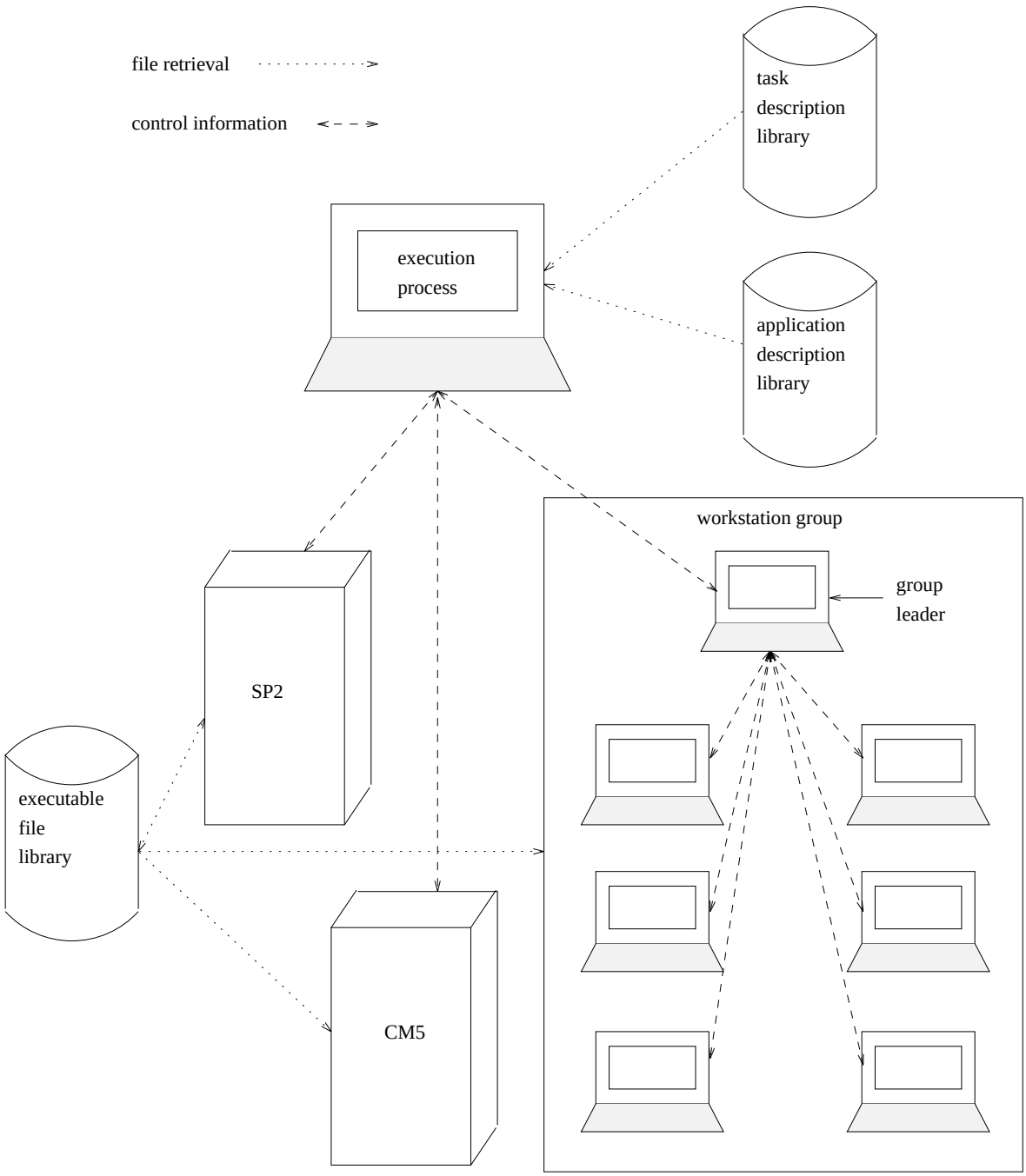


Figure 15: The VCE Execution Environment

The execution suite components shown in Figure 14 represent the components of the VCE's unique automated distributed application control system. Research on managing the execution of distributed applications over multiple high performance computers (as opposed to multiple workstations) is a new but active area of inquiry. The VCE execution suite is an early attempt to provide such capability. Figure 15 shows how control information flows in the VCE execution suite. Task and application descriptions are only used by the execution process. It makes task placement decisions based on load information provided by individual machines and group leaders. Task implementations are loaded by individual machines under the control of the execution process.

The function of the execution process is to manage the execution of a single application. From the information in task and application description files it determines the precedence constraints between the application tasks. When a task is ready to be dispatched, the execution process solicits load information from group leaders whose groups contain machines which are capable of running the task (based on the architecture of the machines, the availability of executable files, and the placement of data files and communicating tasks). The scheduling and dispatching algorithms used by execution processes are outlined in Section 5.1.

The implementation of an execution process is straight forward. It reads an application description and builds data structures that indicate the synchronization and communication requirements of each task. The application description will indicate that some tasks cannot be run until others have completed (i.e. a precedence constraint exists). When all such constraints are satisfied the execution process determines (from a task description) what executables are available for the task. It then contacts eligible group leaders to determine how many machines are available to run the task. Finally, it sends dispatching commands to the scheduling/dispatching daemons on the available machines. Once all eligible tasks are dispatched the execution process waits for notification from a scheduling/dispatching daemon that a task instance has completed. When such a notification arrives, the execution process cancels any redundant instances of the task and checks to see if the completion satisfies the

precedence constraint of any other task or completes the application. The execution process also copies data files needed or produced by task instances and instantiates intertask communication facilities as needed (Section 5.2).

The execution process is the most complex element of the execution suite. While the execution process runs on the workstation of the application user, group leaders and scheduling/dispatching daemons run on widely dispersed host machines. They must not place a significant load on these machines. A scheduling/dispatching daemon periodically samples the load on its host, notifies its group leader if the load changes significantly, and executes dispatching instructions sent by execution processes. Group leaders maintain a list of lightly loaded group members and furnish it to execution processes upon request.

Modularity is a goal of the execution suite as well as the software development suite. Among execution suite components, only execution processes are effected by changes to the application description framework. Refinements to the scheduling/dispatching daemon's load monitoring algorithms, which determine when a machine is eligible to accept VCE work, can be made without impacting other components.

3.2 The Software Development Suite

The decomposition of the software development suite shown in Figure 14 can be achieved by adapting a modular application builder [103] such as AVS [2] for use in the VCE.

AVS is a tool which is used to construct scientific applications from vendor supplied or user written modules [73]. AVS provides a set of data types and functions for receiving or transmitting data between modules. Well designed AVS modules accept an input data type, perform some operation on it and produce output data (such modules are often called filters).

Applications are constructed with AVS using a graphical user interface. The user can select modules from a menu, place them on a palette and draw connections

between them to indicate how the output of one module is to be supplied as input to others. This forms a dataflow pipeline. In most AVS applications the last module in such a pipeline renders the data using AVS supplied graphic functions.

AVS provides some support for distributed systems. Specifically, in a workstation cluster environment the user can indicate that different modules are to be placed on different machines within the cluster. However, no load balancing capabilities are supplied.

AVS support for supercomputing platforms is also rather limited [103]. The most common way to enable a parallel computer to participate in an AVS pipeline is to write a special module that runs on the user's workstation and uses sockets connections to communicate with a remote supercomputer and run a program there. Recent releases of AVS have included some facilities to allow AVS modules to be implemented on some types of parallel machines (such as the CM5). Because the number of parallel architectures supported is small, and only a limited number of AVS data types can be used on them, AVS is still not suitable for constructing applications which involve processing of large data sets on multiple, geographically dispersed supercomputers [103].

The goal of the VCE software development suite is to provide a software engineering environment which is as intuitive and usable as AVS but which serves as a front end to a much more robust execution environment. AVS's emphasis on visual user interaction, modular application development, function encapsulation and program reuse are retained in the VCE. In fact, it is likely that AVS will be used to prototype the VCE software development suite. AVS captures the application specification supplied by the user in a data structure called an *AVS network* [2]. Implementing an AVS based front end to the VCE software development suite simply entails constructing a translation program to convert the AVS network syntax into VCE application description framework syntax. Similar translators can be used to include other modular application builders, or development tools based on other paradigms, into the software development suite.

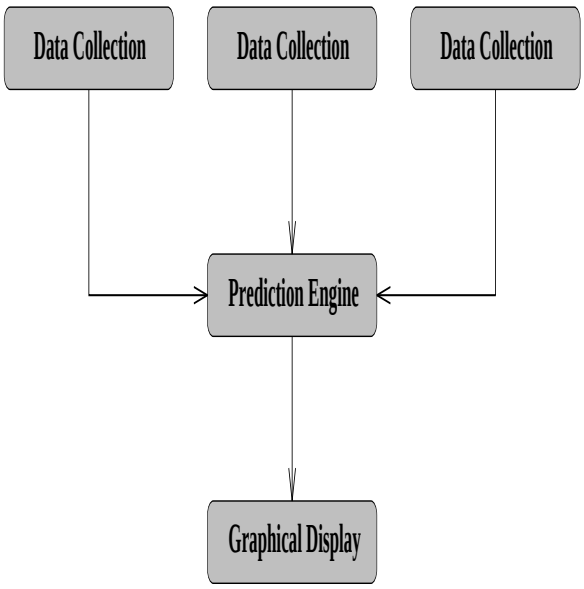


Figure 16: Task Graph of a Meteorological Application

3.2.1 Example Application Development Activities

To demonstrate the capabilities of the software development suite an example is presented showing how a meteorological application which processes observation data to predict storm activity is developed using one particular development approach.

The application is represented in the software development suite as a *task graph*. Problem specification tools are used to formalize the functional flow of the application and create the initial task graph. Figure 16 shows the preliminary task graph.

Once the initial task graph is complete, design tools are used to identify the temporal (time or synchronization) characteristics of the tasks within the application. For this example the we consider design tools based on Fox's [43] architecture of problems and portable parallel software. The information supplied at this stage helps the execution suite to select suitable hardware architectures to host each task. Figure 17 shows the task graph of the weather forecasting application after the design stage analysis is complete.

Finally, in the coding stage of development, the user supplies an architecture

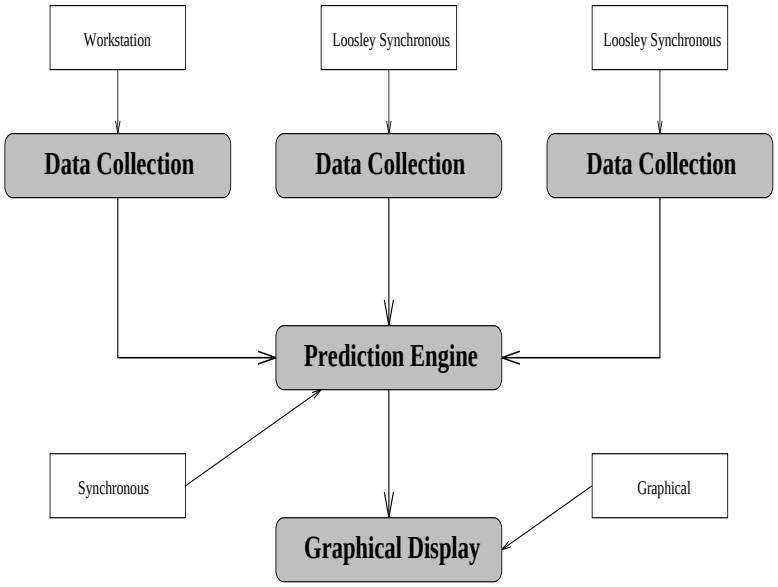


Figure 17: Annotated Design Stage Task Graph

independent source module to implement each task. Tools based on emerging standards such as High Performance Fortran [46] and High Performance C++ [29] will be included among the software design suite's compilation tools to express intratask parallelism. Support for communication between tasks will be provided by the VCE communication system which is discussed in Sections 3.6 and 5.2. Specifying the coding phase information for the weather forecasting application results in the task graph shown in Figure 18.

The completed task graph contains the input, output, and synchronization requirements of each task and its implementation language. As the task graph is annotated in the various stages of the software development suite, the user is able to record information about the nature of the application that may be useful to the execution suite. These *hints* will allow the execution module to do additional optimization. For instance, suppose a particular application has three functionally parallel modules and the user expects one to run much longer than the combined running times of the other two. If the system is aware of this, dispatching of the longer task can be given higher priority and opportunities for parallelism will be maximized.

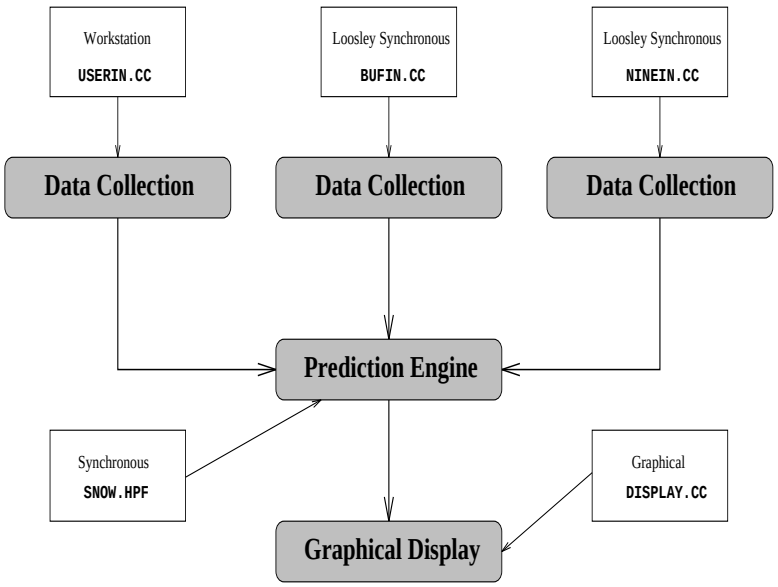


Figure 18: Coding Level Task Graph

3.3 The Application Description Framework

The application description framework provides a textual methodology for encoding all information about a distributed application that the VCE execution suite will need to efficiently dispatch it and manage its communications. The framework is described in detail in Chapter 4.

Since compilers are a component of the software development suite, executable files are one of its output products. A single program produced by the software development suite may be compiled on several different hardware platforms, resulting in a set of executable files. Similarly, a problem might be attacked using different algorithms on different machines, resulting in a set of programs that yield functionally equivalent executable files for a variety of platforms. Managing these programs and executable files is the responsibility of the tools in the software development suite. However, the execution suite needs to be aware that a set of executables can be used interchangeably when dispatching an application. Therefore, the software development module constructs a *task description file* which describes such a set of

executable files. The task description file includes an indication of which executable file is to be used on which platform. It also indicates which platforms are preferable to others and documents the type of input data consumed by the task and the type of output data produced. Other dispatching hints can be supplied by the user and encoded in the task description file.

Each task constitutes one application component available to developers. As described in the previous section, a user can employ an AVS-like tool to construct an application by selecting tasks and graphically indicating their communication and synchronization requirements. The user can also use the application construction tool to specify the source of application input and the destination of its output. The result of this activity is an *application description file* which captures the structure of the application.

The application description framework specifies the syntax for task and application description files which, along with executable files, are produced by the software development suite and used by the execution suite to realize applications.

3.3.1 Supporting Object Oriented and Other Development Methods

The specification, design and development tools described in Section 3.2 are procedurally oriented. That is because most current AVS-like modular application builder tools, which are the motivation for the current software development suite design, are procedurally or dataflow oriented [103]. However, the application description framework contains capabilities to manage object oriented applications as well. Tasks can be specified as providing method implementations and an application description can lay out the flow of data and control within a class hierarchy. In the future, as visual tools for designing and developing object oriented applications become more common, they can be integrated into the software development suite. The application description framework and the execution suite already have capabilities to describe

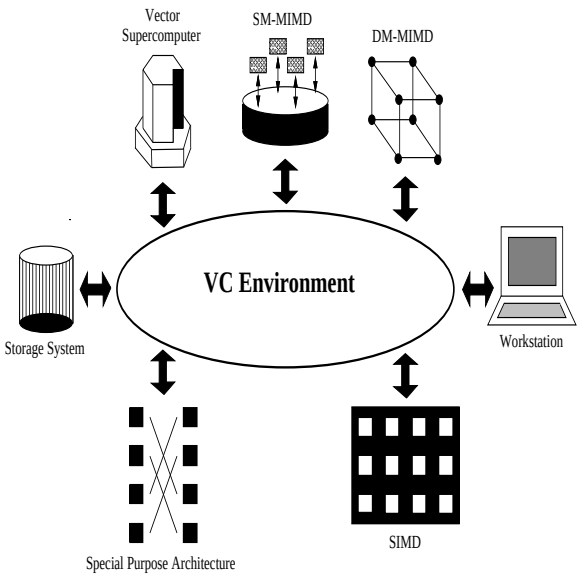


Figure 19: The Virtual Computing Environment

and execute object oriented applications, so the addition of an object oriented development environment will be straight forward. Also, the syntax of the application description framework is simple, intuitive and human-readable. This will give users the option of preparing task and application description files manually if an application is very simple or if it is being developed using a methodology (such as object oriented) which is not supported by a visual programming environment.

3.4 The Execution Suite

The VCE execution suite coordinates execution of user applications on a collection of heterogeneous network attached computers as illustrated in Figure 19. Participating computers may be widely geographically separated. All computers used by the VCE are shared with conventional users. The VCE seeks to use only excess capacity on participating machines and minimize any possible impact to non-VCE users.

Most of the user's interaction with the VCE will be with the software development

suite. Once a user is satisfied that an application is correctly specified and implemented he or she will invoke the execution suite to run it. Therefore, the requirements of the execution suite can be stated succinctly. They are to take an application, which is defined by the information contained in a fully specified application description (as shown in Figure 18) and the modules referenced by the description, and manage the execution of the tasks so as to allow the efficient realization of the application in a heterogeneous distributed environment exemplified by Figure 19.

Some of the functions which must be performed by the execution suite include:

- Managing the intertask communications of VCE applications.
- Locating hosts which are architecturally suitable for application tasks and dispatching the work.
- Monitoring the load on participating machines to prevent the scheduling of VCE work on machines that are already busy.

While all of these activities are discussed in detail in subsequent chapters, the remainder of this section provides an overview of the techniques used in the VCE for these functions.

3.4.1 Components of the Execution Suite

Machines with a wide variety of capabilities participate in the Virtual Computing Environment. A *VCE daemon* running on each machine contributes to global scheduling and remote execution functions. The machines participating in the VCE are organized as groups, where the members of a group share similar architectural features. In a particular configuration there might be an Alpha workstation group, a Sun workstation group, and an SP2 parallel computer group. One machine within each group acts as group leader and coordinates the activities of the group.

The group concept lends considerable flexibility to the VCE and will be discussed in more detail in Section 5.1. Individual machines send load information to the group

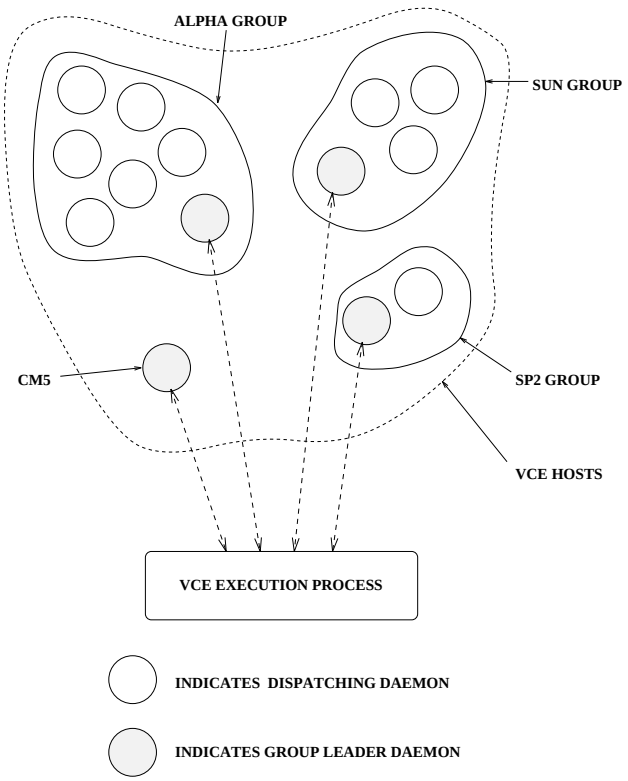


Figure 20: Organization of Machines in the VCE

leader. In current prototypes this information is simply a message indicating that the load on the machine has changed and now is or is not low enough that the machine can host VCE tasks. The group leader is therefore a repository of load information for the entire group and can be consulted when placement decisions need to be made rather than communicating with all the group members.

Execution of a VCE application utilizes an *execution process* running on the user's workstation. The execution process uses information produced by the software development suite. The application and task description files are parsed to determine the types of machines required to run the application. The execution process then sends request messages to leaders of groups that are candidates to host portions of the application (see Figure 20). Groups are chosen based on the availability of executable files and the preferences and hints supplied by the user. Each group leader returns a list of lightly loaded group members. The execution process then sends task dispatching instructions to the machines identified by the group leader. On some types of machines VCE work is dispatched at low priority to minimize interference with non-VCE processing. Internet and World Wide Web technology is used to allow the execution process to locate group leaders.

3.5 Load Balancing

Much of the literature on load balancing is not useful in designing a load balancing methodology for a system like the VCE. Load balancing algorithms are often validated on homogeneous systems using tasks that are easily migrated (like parallel Monte Carlo simulations [104] or big batch jobs [21]). Load balancing algorithms are not tested on large networks composed of multiple supercomputers.

To provide the most robust possible execution environment and provide the greatest possible flexibility, the execution suite may implement a variety of process migration schemes. The following list outlines four process migration approaches that may be useful the VCE.

- **Process migration through redundant execution.** Dispatch the same task on several idle machines. If one of those machines gets busy with other work the other instances of the redundant task will still perform well.

- **Process migration through checkpointing.** Migratable jobs checkpoint regularly. To migrate a job kill it and start it somewhere else by instantiating the new instance from the checkpoint record. This is expensive and may require the cooperation of the task involved (the task may need to be coded to write and read checkpoint data [63]).

- **Process migration the old-fashioned way.** To migrate a job dump the contents of the address space, copy it to a new machine and restart it. This has many drawbacks, one being that it requires homogeneity [71].

- **Process migration through recompilation.** This is very expensive but may be very robust. It is likely to be difficult to implement [102].

Ideally, The VCE execution suite would include implementations of all these algorithms. When a migration is needed, the most efficient technique would be chosen. For instance, if a task was to be migrated across compatible hosts, dumping and migrating its execution image would be feasible. For migrations between heterogeneous machines, checkpointing or recompilation methods would be needed.

So far, VCE research has focused on the use of redundant execution as the most general load balancing technology. It permits the use of heterogeneous hosts and does not require any special processing on the part of the task being run. Redundant execution does require special support from the run time system. Communication among redundant instances of application tasks must be managed carefully to insure that the redundancy is transparent to the application developer and communication coherence is maintained even if some task instances progress considerably faster than others. Details of how communication is handled in the face of redundancy is discussed in the next section and in Section 5.2.

A central contribution of the current work is to demonstrate the value of redundant execution as a load balancing tool in high performance computing environments. Techniques for scheduling redundant execution are outlined in Section 5.1. In the absence of redundant scheduling or the more complex strategies mentioned above, the only method for scheduling task executions involves *selective scheduling*. That is, placing tasks on a single lightly loaded host. Chapter 6 will describe a series of prototype and simulation experiments which were performed to gain insight into the characteristics of both redundant and selective scheduling. These experiments showed that relatively low levels of redundancy (three instances of each task) dramatically improved performance and were sufficient to overcome the performance penalty of low priority dispatching.

3.6 Communication

The VCE communication system is discussed in detail in Section 5.2. Many distributed computing environments use some variation of a proxy based system (Figure 21) for communication among distributed application components [59] [89]. That is, if two portions of an application are to run on different machines, the distributed computing system supplies proxy processes which present the same interface as the remote component but actually communicate with their counterparts to allow remote communication using the semantics of local communication. This isolates the distributed application from the complexities of network communication and greatly simplifies distributed application development.

In the VCE these communication proxy functions are performed by *communication processes*. Once placement decisions are made by the execution process, it starts a communication to support each task instance. The communication process runs on the same machine as the task it supports.

The communication pattern of the application is described in the application description file and is conveyed to the communication process by the execution process. This enables the communication process to know the types and destinations of the

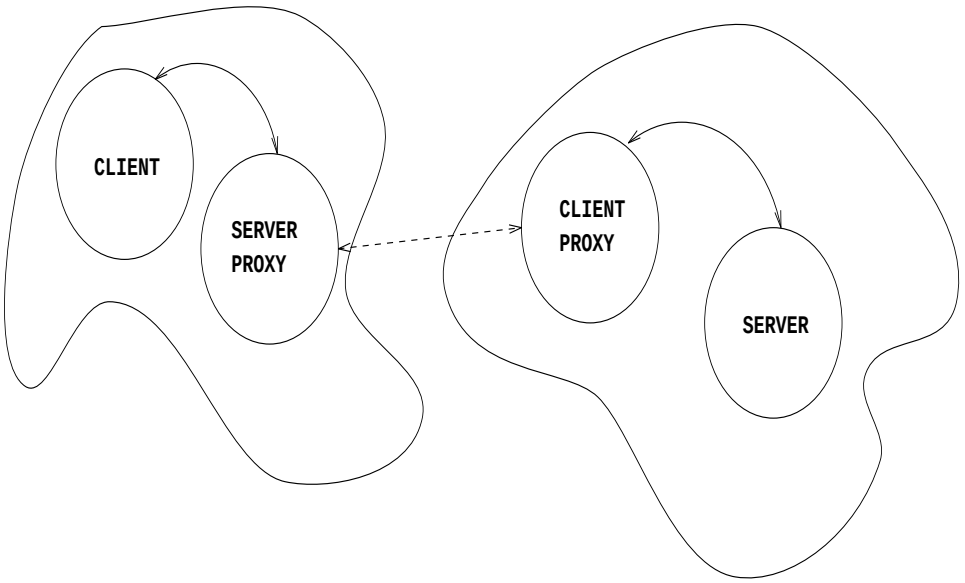


Figure 21: Communication via Proxies

data it will be responsible for. The communication process insulates the user task from the management of redundancy by automatically duplicating output data for all instances of a replicated destination task and discarding redundant input received from replicated source tasks.

3.7 Design Attributes

The VCE design is more robust than related systems described in Chapter 2. The software design suite allows a user to design an application from existing components without requiring any special knowledge of the programming languages, algorithms or hardware they employ. All these issues are addressed by task implementers, who similarly need not be knowledgeable in the particulars of the applications in which the tasks will be used. This enables scientists and engineers to exploit high performance computing facilities more easily than systems which focus on parallel programming

language extensions (Sections 2.7 and 2.9). The modularity of the software development suite will permit the use of procedural, object oriented or other (perhaps as yet undiscovered) software engineering technologies.

The VCE's execution suite can coordinate application tasks executing simultaneously on multiple high performance computers and heterogeneous workstations. This will accommodate the execution of Grand Challenge applications which require multiple heterogeneous high performance computers to acheive their best performance [74]. Previous remote execution systems were often limited to coordinating application execution across multiple workstations (Section 2.6). The VCE retains the ability of these systems to detect idle hosts and use the resources of these machines. These benefits are obtained without operating system modifications.

Chapter 4

The VCE Application Development Model

The previous chapter introduced the two major components of the VCE. They are, the *software development suite*, which provides software engineering tools for designing and developing large applications, and the *execution suite* which manages dispatching, scheduling and communication among application components across a distributed heterogeneous hardware configuration. A central design element of both these components is the *VCE application description framework*. Each VCE application has a corresponding description expressed in the semantics of the application description framework. The description captures all the design information produced by the specification, design and coding tools which make up the software development suite. The execution suite is supplied an application description as input. It uses the description, and the executable and data files the description refers to, to execute the application. The application description framework is the principle interface between the two major components of the VCE. Tools in the software development suite are geared towards producing and refining an application description while the technology in the execution suite interprets and executes it.

The structure of the application description framework dictates the capabilities which are required of tools included in the software development suite. That is,

any tool, or set of tools, which can produce output which can be translated into VCE application description framework syntax can be included in the software development suite.

4.1 The VCE Application Construction Approach

The design of the software development suite, as laid out in Section 3.2, includes tools that assist the user in all phases of application design, development and implementation. Specification tools help the user to express the characteristics of the problem. Design tools are used to specify the major components of the application, their hardware, synchronization and communication requirements. If an application can not be completely implemented using existing components, compilation tools are available to assist the user in implementing new executable tasks. Tools supporting any development methodology (procedural, object oriented, etc.) or multiple methodologies can be included in the development suite. The only requirements is that they produce application descriptions expressed in, or translatable to, the syntax of the VCE application description framework.

Section 2.11 discussed the differentiation between programming-in-the-small and programming-in-the-large [35]. Unlike most of the parallel/distributed environments discussed in Chapter 2, the VCE encourages a strong separation between high performance programs and high performance applications. In building a Grand Challenge application it may be necessary to exploit a variety of hardware architectures, programming languages and programming tools in a single application. Rather than providing more sophisticated programming paradigms for expressing complex cross-platform communication and synchronization operations, the VCE application description framework allows programs written in any language on a variety of architectures to be incorporated into an application. By isolating dispatching, synchronization and communication specifications in a separate description file, the source code for individual computing tasks can be simplified. It is also easier to reuse programs if they are not dependent on the structure of the applications they are used in [86].

The VCE application description framework is a synthesis of, and extension to, two existing technologies: distributed application description languages (Section 2.8) and distributed object management systems (Section 2.9). Both of these approaches provide mechanisms for expressing how a large software system is decomposed into smaller components and how they are to be dispatched on a distributed hardware configuration to form an application. The VCE application description framework builds on the most powerful characteristics of both these technologies. Unlike current distributed application description languages, the VCE application description framework provides facilities to describe the structure of object oriented applications. It provides a syntax that is simple and flexible enough to support a variety of software engineering approaches including procedural, object oriented and data flow. It can easily be extended to accommodate development models that will evolve in the future. It is also intended for more general purpose uses than distributed application description languages that are specifically designed for fault tolerance and dynamic reconfiguration. Unlike distributed object systems, the application description framework allows application structure to be expressed outside of, and therefore independently of, the source code of the programs that make up the application. This eases program reuse and gives the run time system the greatest possible flexibility in dispatching.

The remainder of this chapter will focus on the structure of the VCE application description framework. The framework defines what is required of tools in the software development suite. It illustrates the VCE application model where *applications* are constructed from smaller computational units such as *tasks* or *method implementations* which are performed by *executable files* running on hardware *platforms*. Tools in the VCE software development suite help users to construct applications in these terms. The application description framework is the bridge between the tools in the software development suite and the distributed application scheduling, dispatching, control and communication tools in the VCE execution suite.

4.2 The VCE Application Description Framework

Durra (Section 2.8) and DSOM (Section 2.9) are two very different approaches to distributed application development environments. The differences between them are largely due to their different objectives. Durra is oriented towards creation of fault tolerant embedded systems while DSOM is meant to ease development of object oriented systems which access remote shared resource.

The objectives of the VCE are also unique. These objectives are to support development of Grand Challenge applications which exploit multiple supercomputers and workstation clusters, and better utilize these hardware facilities. As in Durra, The VCE uses task descriptions, which correspond to programs, and application descriptions which specify the communication and synchronization patterns among a group of tasks. However, while there is a one-to-one correspondence between task descriptions and executable files in Durra (Figure 7), a single VCE task description may point to executable files on several different architectures. This allows the VCE run time system to dynamically choose which machine to run the task on, or to run it on several machines redundantly. Another unique feature of the VCE is that application specification is kept as simple as possible. Both Durra and DSOM consider the application implementer to be a programmer. In the VCE, it is expected that applications will be constructed by scientists and engineers. Therefore, VCE application descriptions only describe the communication and synchronization requirements of a group of tasks or objects. Decisions about how to allocate tasks to processors are made by task implementers or the run time system.

The remainder of this section introduces the elements of the VCE application description framework and illustrates its use to construct an example application.

4.2.1 VCE Platforms

A VCE *platform* is any computing system capable of performing computations for a VCE application. Parallel supercomputers, clusters of workstations and individual high performance workstations can all be regarded as potential VCE platforms.

4.2.2 VCE Tasks

Within the VCE, a *task* refers to a dispatchable entity. A task may be a parallel program and may, therefore, execute on several nodes of a single platform. A task may not simultaneously execute on nodes of more than one platform. For example, a task might require four nodes on an IBM SP2 supercomputer or four nodes from a DEC Alpha workstation cluster. However, a computation that required two nodes of an IBM SP2 and two nodes of a DEC Alpha workstation cluster could not be regarded as a single task.

A single task may have implementations on many platforms. That is, there may be executable files corresponding to a task on multiple platforms which perform the same computations and have the same interfaces.

4.2.3 Intertask vs. Intratask Communication

If a task is implemented as a parallel program on some platform, the communication among the participating nodes is managed by the task. The task could use any parallel programming mechanism such as a message passing tool [99] or parallel programming language [46] to perform intratask communication. Intratask communication is distinct from the task's interfaces, which define how it can interact with other tasks and the VCE.

Intertask communications are managed by the VCE. The application description framework includes a syntax for specifying the intertask communication requirements of application components. The task description syntax (Section 4.2.4) allows the task developer to specify the input consumed and output produced by a task. The application description syntax (Section 4.2.6) allows the application developer (or application development tool) to indicate how outputs of one task or group of tasks are to be routed as input to another. Section 5.2 will discuss how these specifications are used by the execution suite to construct *communication processes* which send output and receive input on behalf of application tasks. These communication processes use the information captured in the application description framework to provide location

[**task:** task_name]
[**executables:** architecture: path_name
<architecture2: path_name2>
< ... >]
<**nodes:** architecture: number_of_nodes
<architecture2: number_of_nodes2>
< ... >>
<**command:** comand_line_parameter <command_line_parameter2> < ... >>
<**input:** input_object_name.input_object_type
<input_object_name2.input_object_type2>
< ... >>
<**output:** output_object_name.output_object_type
<output_object_name2.output_object_type2>
< ... >>
<**hints:** <replicateable | non-replicateable>
<duration: short | medium | long>
<method-implementation>>

[ ] indicates required parameters
< > indicates optional parameters
bold face indicates keywords

Figure 22: VCE Task Description Syntax

transparent intertask communication functions among application components. They also allow redundantly executing tasks to communicate transparently.

4.2.4 VCE Task Description Syntax

Each task used in a VCE application has a corresponding *task description*. The task description documents various characteristics of the task which enable the VCE run time system to dispatch it and manage its communications.

The syntax of a VCE task description is shown in Figure 22. The parameters of the description have the following meanings:

task_name Is a mnemonic name for the task which will be used to refer to the task in application descriptions. In most cases the name of the task description file should match the task_name.

architecture: path_name The *architecture* refers to a platform architecture. For example, in one VCE prototype implementation (to be discussed in Chapter 6) valid architecture specifications were **SP2** to indicate an IBM SP2, **ALPHA** to indicate a DEC Alpha cluster, and **SUN** for a Sun workstation cluster. The *path_name* identifies an executable file which implements the task on the given architecture.

architecture: number_of_nodes For each target platform this parameter indicates how many nodes the task is to be allotted.

command_line_parameter Is a string to be passed to the task as a command line parameter.

input_object_name.input_object_type Defines the name and type of an input object to be supplied to the task. Details of how input and output parameters are supplied to tasks will be discussed in Section 4.2.5.

output_object_name.output_object_type Defines the name and type of an output object produced the task (see Section 4.2.5).

replicateable‖non-replicateable The run time system uses redundant execution to achieve load balancing and improved performance (Section 5.1). A task is *replicateable* if it does not read any file or shared data object which it also modifies. If this is true, and file updates are made by writing rather than appending, then the task will function correctly even if multiple instances are executing concurrently.

duration: short‖medium‖long This information helps the scheduler determine when redundant scheduling will be beneficial.

method-implementation The task is an implementation of a CORBA compliant [82] object method. The first object listed in the input specification is taken to be the object which the method is to be applied to. The type of this object

task: LU_decomposition
executables: SUN: LU_decomposition.sun
SP2: LU_decomposition.sp2
ALPHA: LU_decomposition.alpha
nodes: SUN: 4
SP2: 4
ALPHA: 4
input: rectangular_matrix.file
output: lower_triangle.matrix
upper_trangle.matrix
hints: replicateable

task: inverse_matrix
executables: SUN: inverse_matrix.sun
SP2: inverse_matrix.sp2
ALPHA: inverse_matrix.alpha
nodes: SUN: 4
SP2: 4
ALPHA: 4
input: input.matrix
output: inverted.matrix
hints: replicateable

task: multiply_matrix_by_vector_file
executables: SUN: matrix_multiply_1.sun
SP2: matrix_multiply_1.sp2
ALPHA: matrix_multiply_1.alpha
input: input.matrix
input_vector.file
output: output.vector
nodes: SUN: 4
SP2: 4
ALPHA: 4
hints: replicateable

task: multiply_matrix_by_vector_giving_file
executables: SUN: matrix_multiply_2.sun
SP2: matrix_multiply_2.sp2
ALPHA: matrix_multiply_2.alpha
input: input.matrix
vector_input.vector
output: output_vector.file
nodes: SUN: 4
SP2: 4
ALPHA: 4
hints: replicateable

Figure 23: VCE Task Descriptions for Linear Algebra Operations

indicates the class which the method is a member of. The *task_name* indicates the name of the method. More details of the use of object implementations in VCE applications are given in Section 4.2.7.

An example of task descriptions for a set of common linear algebra operations is given in Figure 23.

4.2.5 Task Input/Output Semantics

Every distributed system must provide facilities for communication between different components of a distributed application. In Durra this is done with the **port** construct. Durra task programs use funtions provided by the Durra library to write data to, and read data from, **ports**. Application descriptions specify how the **ports** of various tasks are interconnected via **channels**. DSOM programs communicate with remote objects by invoking methods on their proxies. The interfaces and remote locations of the distributed objects are maintained in the interface and implementation repositories.

VCE_send(output_object_name <, output_object_namr2> <...> **)**

VCE_receive(input_object_name <, input_object_name2> <...> **)**

VCE_test(input_object_name <, input_object_name2> <...> **)**

Figure 24: VCE Communication Operations

The VCE provides three functions which are used for intertask communications (Figure 24). The **VCE_send()** function transmits data to other tasks. The `object_name` specified, which identifies a program variable, must correspond with an `object_name` listed as an **output** object in the task description. The type of the output variable must match the `object_type` indicated in the task description. For instance, the `inverse_matrix` task of Figure 23 must perform the call `VCE_send(inverted)`, where `inverted` is an object of type *matrix*, in order to make its results available to other tasks. Similarly, **VCE_receive** is a blocking receive which retrieves data supplied by other tasks. Again, the variable received must correspond to an **input** parameter in the task description. **VCE_test** returns *true* if new input is available for all the variables listed.

Object types are defined using IDL [82] in a VCE interface repository.

The `file` object type has a special connotation. Performing a **VCE_receive()** on an object of type `file` returns a string which is the name of a file in the task implementation's execution context which holds the actual data. Therefore, it is permitted to do a **VCE_receive()** on a `file` object even if it is specified as an **output** parameter. Such a call would return the name of a file to be used by the task to hold its output. Details on the use of `file` objects are included in Section 4.2.6.

4.2.6 VCE Application Description Syntax

A VCE application description contains instructions on how the run time system is to dispatch and interconnect tasks to realize an application. The application description syntax is shown in Figure 25. Applications are divided into *stages*. Each stage

[**application:** application_name]

[**stage:** stage_name

[**task:** task_name]
<**nodes:** architecture: number_of_nodes
<architecture2: number_of_nodes2>
< ... >>
<**command:** comand_line_parameter <command_line_parameter2> < ... >>
<**input:** input_object_name
<input_object_name2>
< ... >>
<**output:** output_object_name
<output_object_name2>
< ... >>
<**hints:** <duration: short | medium | long>>
<**synchronization:** after other_stage_name begins | completes>
<after other_stage_name2 begins | completes>
< ... >>

<**stage:** stage_name2

[**task:** task_name]
<**nodes:** architecture: number_of_nodes
<architecture2: number_of_nodes2>
< ... >>
<**command:** comand_line_parameter <command_line_parameter2> < ... >>
<**input:** input_object_name
<input_object_name2>
< ... >>
<**output:** output_object_name
<output_object_name2>
< ... >>
<**hints:** <duration: short | medium | long>>
<**synchronization:** after other_stage_name begins | completes>
<after other_stage_name2 begins | completes>
< ... >>

< ... >

Figure 25: VCE Application Description Syntax

employs a task to perform some computation. Many of the specifications of a task can be overridden in an application description. The only values which cannot be changed are the **executables**, the *types* of input and output objects, and the *replicateable* and *method-implementation* hints. The values specified for parameters in a task description which can be overridden later constitute the default values for those parameters.

Each application stage can include **synchronization** specifications which indicate the order in which tasks are to be performed or method implementations instantiated. The **synchronization** parameters are a list of conditions which must be satisfied before the task can be dispatched. Obviously, if one stage performs operations on the output of an earlier stage, then the later stage would list completion of the earlier stage as a synchronization constraint. Special synchronization specifications are also needed if one stage sends data to a concurrently executing stage and the waits to receive results back. In this case synchronization directives are used to indicate that the former task is to be dispatched after the latter task is started.

Interconnections between tasks are achieved by manipulating the object names on the **input** and **output** directives. If an output `object_name` of one task and a compatible input `object_name` of another task are overridden with the same value, it will indicate to the VCE execution process that the output of the one task is to be routed as input to the other.

Figure 26 shows an example of how the tasks of Figure 23 can be used to construct a linear equation solver application. The example illustrates several features of how the VCE run time system interprets an application description. The application accepts as input the files `A_matrix_file` and `B_vector_file` and produces `X_vector_file` as output. All the input and output default `object_name`s established in the task descriptions are overridden to achieve the desired data flow. The **synchronization** parameters are used to impose the correct order of operations. The execution process can determine, through analysis of the application description, that no output of any task will be used as input by a concurrently executing task. That is, all intertask communication is performed by buffering the outputs of earlier tasks

Application Description

application: solver

stage: LU_decompose
task: LU_decomposition
input: A_matrix_file
output: lower, upper

stage: inverse_matrix_1
task: inverse_matrix
input: lower
output: lower_inverse
synchronization: after LU_decompse completes

stage: inverse_matrix_2
task: inverse_matrix
input: upper
output: upper_inverse
synchronization: after LU_decompse completes

stage: first_matrix_multiply
task: multiply_matrix_by_vcetor_file
input: lower_inverse, B_vector_file
output: temp_vector
synchronization: after inverse_matrix_1 completes

stage: second_matrix_multiply
task: multiply_matrix_by_vector_giving_file
input: upper_inverse, temp_vector
output: X_vector_file
synchronization: after first_matrix_multiply completes
after inverse_matrix_2 completes

Application Graph

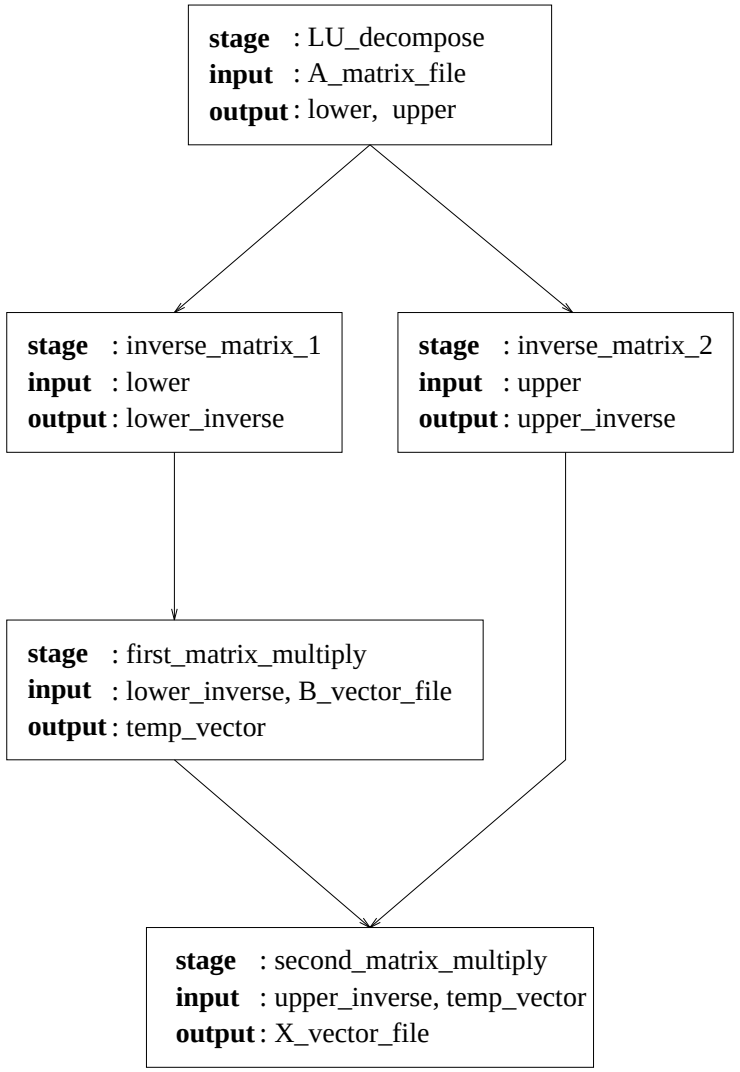


Figure 26: A VCE Application

and providing them as inputs to later tasks.

Figure 26 also illustrates the use of `file` objects in an application description. The task description of `LU_decomposition` (Figure 23) includes the specification "**input:** `rectangular_matrix.file`". The **input** specification of the linear equation solver's `LU_decompose` stage overrides `rectangular_matrix` with `A_matrix_file`. When an implementation of the `LU_decomposition` task issues `VCE_receive(rectangular_matrix)` it will be given the string "`A_matrix_file`" so it can open the file and read its input. The VCE execution process insures that each machine which runs the `LU_decomposition` has access to the `A_matrix_file`.

The three VCE communication operations and the application description framework can be used to construct a variety of application data flow patterns. For instance, if the output of one task is specified as input for several others, then a **VCE_send()** of that output has the effect of a broadcast operation. Similarly, **VCE_receive** can be used to collect output objects from multiple tasks. These communication patterns are established by the way the tasks are used in application description. The same task which feeds input to a single successor task in one application can broadcast its results to many concurrent tasks in another application. This difference in behavior can be accomplished without changing the task implementation or description or recompiling the task.

The application description framework accomplishes several VCE objectives. The syntax is simple, so with just a few pages of documentation it should be easy for a non-programmer to construct an application from a task library. One reason the description syntax is simple is that decisions about how work is assigned to particular machines are made by the run time system rather than the application designer. Facilities for intertask communication are also constructed dynamically at run time rather than being specified in the application description. This simplicity and emphasis on allowing the run time system to deal with as many application execution issues as possible will make it easy to construct intuitive design tools in the Software Development Module.

4.2.7 Object Oriented Input/Output Semantics

In addition to the message oriented communication semantics described in Section 4.2.5, the VCE also uses the syntax of Figure 22 to support object oriented communication semantics. If the hint `method-implementation` is included in a task description, it signals the VCE to treat the executable as a CORBA compliant [82] method implementation. An object method packaged as a VCE task is called a *VCE remote method.* A task can only contain a single method definition. In object oriented environments, CORBA style interface and implementation repositories would be produced by the software development suite as well as the task and application description libraries. The first object listed on the **input** specification of an `object-implementation` task is considered to be the object the method is applied to and identifies the class the method is a member of. The *task_name* must match the name of the remote method. At run time the VCE execution suite checks that the input and output specifications of the task description are a superset of the method signature in the interface repository. Additional input and output objects may be included in the task description to allow the method to invoke other VCE tasks or methods. That is, the **input** objects include parameters received on invocation as well as objects returned by called tasks. Similarly, **output** objects may be passed to called tasks or returned to the method's invoker. Both the task description and the interface repository entries are produced by the tools in the software development suite and the tools insure that the specifications are compatible.

There are two ways VCE remote methods can be invoked, *implicitly* and *explicitly.* Implicit invocations occur when the application description shows that all a method's inputs are supplied by tasks that are predecessors and all the method's return values are used by successor tasks. For example, if all the sample task descriptions shown in Figure 23 included the `method-implementation` hint, the application description of Figure 26 would still be valid. For instance, the `inverse_matrix_1` stage would invoke the method defined by the `inverse_matrix` task on the `matrix` object `upper` and the returned object, `upper_inverse`, would be supplied as one of the inputs to

the subsequent `first_matrix_multiply` stage.

Explicit VCE remote method invocations would be indicated by placing an asterisk after the name of a method being invoked in a source program. This would cause the compilation tools to produce code to alert the run time system to the remote method invocation. The execution suite would use the input and output specifications of simultaneously executing tasks, and its knowledge of the methods they implement, to execute the correct remote method. Again, it is the responsibility of tools in the software development suite to insure that ambiguities do not arise.

4.2.8 Other Communication Semantics

Sections 4.2.5 and 4.2.7 showed how the syntax of the VCE application description framework could be used to support two different communication semantics (massage passing and object oriented). Further extensions to support other paradigms such as distributed shared memory or data flow are also possible. The task description syntax is a language independent and paradigm independent method of indicating the input and output characteristics of a course grain application component, the executable files which implement it, and the hardware requirements of those executables.

4.2.9 Designing VCE Tasks and Applications

The VCE application and task concepts reinforce the notion that task design (program design) and application design are separate activities. This will make the development of grand challenge applications easier by allowing software engineering activities to be more specialized. Different members of a development team can focus on areas where they have particular expertise. For instance, one possible to organize members of a VCE development team would be to divide it into *task designers*, *task developers* and *application designers*.

Task designers can focus on providing general purpose software components with well defined interfaces and high granularity. The goal of the task designer is to anticipate the needs of the application designer.

The *task implementer*, who is presumed to be knowledgeable in parallel programming techniques, can select the best available tool for implementing each task on each platform where it is sensible to do so. Task implementers are the only team members who must be skilled programmers. They can focus on optimizing task implementations on the widest possible variety of high performance architectures.

Application designers should be experts in the application area. They do not need to be concerned with the details of how tasks are implemented or what particular machines are available. Their only concern is to determine how to express an application in terms of the task implementations included by the team's task designers in the VCE repertoire.

The ability of the VCE to enable experts in application areas to construct and execute very large applications that exploit multiple high performance computing systems, without having to master programming languages and other tools, represents a significant advantage over the systems described in Chapter 2.

4.2.10 Extensions to the Application Description Framework

A variety of possible extensions to the application description framework which will make it more flexible, robust and expressive will be discussed in the "Conclusions and Future Work" chapter of this dissertation (Section 7.2.1). Extensions to the syntax can be implemented easily because of the flexibility of the underlying design. One design philosophy that permeates VCE research is that programs (task or object implementations) should be developed independently to encapsulate a general purpose coarse grain computational function. Applications can be developed by expressing how a group of tasks are to be dispatched and interconnected. The application description framework is the VCE approach to communicating the charastics of tasks and applications to the run time execution tools. While the framework may be made more useful by the addition of syntactical elements to express conditional or iterative execution, or other computational characteristics, these can be built upon the current framework of a straight forward task and application description syntax specifically

oriented towards high performance distributed heterogeneous computing.

4.3 Application Execution

The application description framework provides a mechanism for capturing information about the structure of a large scale application. The major phases (tasks) of an application can be represented along with their hardware, file, synchronization and communication requirements. The syntax is easy to parse. In a single pass it is possible for the VCE execution process to build data structures that represent each phase in an application. Tasks that have no synchronization constraints are dispatched initially. Each time a task completes other tasks are checked to determine if they are now ready to begin.

The execution suite judges the run time conditions and maps application tasks to available hardware resources. It determines if redundant task scheduling is appropriate and dynamically allocates intertask communication facilities. The scheduling, dispatching and communication facilities provided by the execution suite are the subject of Chapter 5.

Chapter 5

The VCE Execution Environment

The tools in the VCE software development suite help users to construct high performance applications. They allow an individual task to be implemented on various platforms with a set of executable files that all have the same input requirements and output characteristics. Applications can be constructed by indicating how tasks are to be interconnected and dispatched. The VCE run time system (the *execution suite*) manages the execution of these applications. It is responsible for determining what hardware resources are available and which of the available platforms should be used to host task executions. The execution suite is also responsible for providing communication facilities between application components and efficiently transporting intermediate data between tasks.

Executing large multistage applications on multiple high performance computers presents unique scheduling and load balancing challenges. Each application task may run for minutes or hours. Transient load conditions can change drastically during task execution and impede application performance. Migrating a task from one parallel computer to another during its execution is problematic. In order to address the difficulty of providing acceptable performance in an environment where process migration is not feasible, the VCE execution suite employs redundant scheduling techniques. If multiple suitable hosts are available to run a particular task, the VCE execution suite will dispatch the task on several host simultaneously. If the load on

some hosts increases, the task will complete in the time achieved by the least loaded host.

Of course, redundant scheduling requires specialized communication support. New Algorithms have been devised to provide flexible and efficient communication functions among redundantly executing task instances.

This chapter will provide a detailed description of the scheduling, dispatching and load balancing methods used to manage redundant execution (Section 5.1). It will also outline the methods used to provide transparent communication in the presence of redundant execution (Section 5.2).

5.1 VCE Scheduling Techniques

A VCE application consists of a set of tasks with associated synchronization and communication requirements. The tasks can be dispatched independently, as long as all synchronization constraints are observed and appropriate intertask communication mechanisms are provided. This section will explore techniques for scheduling and dispatching VCE tasks.

The goal of the VCE's scheduling techniques are to maximize application performance, improve hardware utilization and minimize interference with non-VCE workload (particularly on privately owned workstations).

Some of the factors which have been considered in the design of the VCE scheduling system include:

- The existence of many hosts to run each task.
- Preemptive process migration in a heterogeneous environment is problematic.
- Precise load measurement across a wide area network is difficult.
- The number and types of host systems participating in the VCE may change from one application run to the next and even during a single application execution.

In order to accommodate these factors a scheduling/dispatching protocol has been designed for the VCE. It involves scheduling redundant execution of task instances. A conservative load measurement policy is used to identify lightly loaded hosts for initial task placements. When task instances are placed on privately owned workstations they are run at low priority to minimize their impact on locally originated work.

While the VCE may benefit in the future from a large repertoire of process migration mechanisms (Section 3.5), current technology makes preemptive process migration between incompatible machines, and even between architecturally compatible parallel computers, impractical. Experimental and simulation studies discussed in Chapter 6 have shown that the redundant scheduling policy described here is suited to a broad range of VCE configurations.

The VCE has several characteristics which make it significantly different from more conventional distributed systems. These differences make it difficult to use an existing scheduling discipline. Instead, careful study of typical VCE applications and hardware configurations, along with analysis of simulated and prototype systems, was used to develop a scheduling protocol appropriate for use in the VCE.

While scheduling and load balancing in distributed systems have been studied thoroughly [26], some key assumptions of those research efforts do not hold for the Virtual Computing Environment. For instance, load balancing studies have traditionally assumed that the number of tasks to be scheduled is much larger than the number of processors available [32]. However, in the VCE this situation may be reversed. At any given time, there may be only a few Grand Challenge VCE applications running, each with a small number of tasks executing concurrently in any one phase of the application. Further, if a large number of computer centers, connected by a wide area network, are participating in the VCE, the number of possible hosts for each task can be quite large.

The scheduling mechanisms of the VCE have less control over the environment than other remote schedulers. Most scheduling schemes assume that all work on all processors is under the control of a single scheduling algorithm [15, 28, 32, 34, 36, 37, 84], which is not the case in the VCE. Computers participating in a VCE

configuration will remain available to accept tasks from other sources (usually local users). Most load fluctuations will be caused by increased or decreased demands of local users rather than by VCE activities. The VCE's goals of providing the best possible performance of VCE applications with the least impact to non-VCE users is less ambitious than the cited projects which manage all scheduling on (generally smaller) distributed systems and seek to approximate globally optimal scheduling.

Intelligent scheduling techniques [28, 84, 106] may be useful in the VCE. An algorithm based on neural networks has been proposed [106] which learns the characteristics of jobs after repeated runs and can use this knowledge to predict the effect of various scheduling options. Implementation of these approaches in the VCE is difficult. The large number of hosts in a VCE configuration makes collection of accurate state information problematic. Because the available hardware may be completely different each time an application is launched, inferences based on passed application behavior may be unreliable. Currently VCE research is concentrated on the use of redundant scheduling techniques, which have proven very successful (Chapter 6).

5.1.1 The VCE Execution Environment Components

The components of the VCE run time environment were discussed in Section 3.4.1. They include a scheduling/dispatching daemon running on each host node, group leaders which collect load statistics from the scheduling/dispatching daemons on architecturally compatible hosts, and an execution process which runs on a user's workstation. The execution process interprets application descriptions and collaborates with other components to schedule and dispatch tasks.

5.1.2 Elements of the Load Balancing Strategy

Dynamic load balancing and scheduling strategies can be characterized by the following four components [54]:

1. **The processor load measurement technique** - determines what quantities will be measured to gauge system load conditions.

2. **The information exchange policy** - indicates when machines exchange load information to detect imbalances.

3. **The transfer policy** - the criteria which governs when and what work should be moved from one machine to another.

4. **The location policy** - selects a machine where migrated work is to be placed.

This section will discuss how redundant task scheduling allows a simple and efficient implementation of each of these four components.

Processor Load Measurement

Objective

In any scheduling system, load measurement is needed to detect load imbalances. Systems that employ process migrations must insure that both the migrating task and the system overall will benefit from a migration. Timely and precise load measurements are often needed to achieve this objective. However, excessive overhead due to load measurement must be avoided.

Common Approaches

A survey of load measurement methods is included in [69]. Some possible load measures include: the number of tasks in a processor's run queue; the amount of available memory; the rate of CPU context switches; average run queue length over some time quantum; or the amount of idle CPU time during a time quantum. In practice, it is common to use some combination of these methods [9, 109].

Many systems which schedule remote executions on privately owned workstations use load measurement techniques geared towards distinguishing whether or not a machine is idle [6, 20, 21, 87]. "Idle" is usually defined as a very small average run queue length and no keyboard or mouse activity over some period of time [101].

Redundant Task Scheduling Approach

The main goal of load measurement in redundancy based scheduling is to identify lightly loaded machines where task instances should be placed. An extremely precise

load measurement technique is not needed for redundant task scheduling. If one instance of a task is placed on a heavily loaded machine, as a result of imprecise load measurement or bursty changes in the load on that computer, the result produced by another instance of the task running on a lightly loaded machine will be accepted. The overall performance of the application will be good in spite of poor allocation of one instance of a task. The scheduling approach that places tasks on completely idle machines [21] is overly conservative. Large parallel computers are rarely idle. A workstation being used to view e-mail or perform other casual processing can host background tasks, especially when they are run at low priority, without impacting the foreground user.

The scheduling goal of the VCE is to harness the excess capacity of lightly loaded machines without placing additional burdens on heavily loaded machines. A more formal definition of heavily and lightly loaded is needed. A machine is heavily loaded if its users notice poor performance. The load of such a machine is referred to as being over its *nuisance threshold*. A machine that would be below its nuisance threshold even after the addition of one more task or process is below its *adequate utilization threshold* and is considered lightly loaded (see Figure 27).

The load measurement routines for machines in the VCE employ an algorithm described in Figure 28. The algorithm computes a smoothed load measure ($Smoothed_load(t_i)$) as well as an instantaneous load measure ($Load(t_i)$). The instantaneous load measure is based on frequent snapshots of the local load conditions. Both the instantaneous and smoothed load measures can take any of the three values low, medium or high as indicated in Figure 27. When the smoothed load measure becomes low the group leader is notified that the machine is available to host remote tasks. When the smoothed load measure goes from low to medium the group leader must be informed that the machine is no longer in need of more work. A smoothed measure is used because the instantaneous measure may fluctuate due to the transient effects such as process initiation. It will also fluctuate when the load is very close to the threshold. The smoothed measure is not changed until several consecutive changes in the

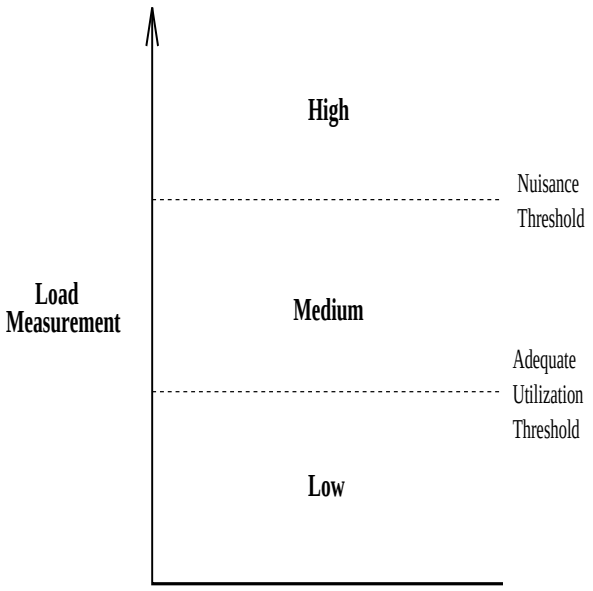


Figure 27: VCE Load Thresholds

instantaneous measure in the same direction are observed. Smoothing also allows for the introduction of a bias. That is, the criterion for decreasing the smoothed measure can be constructed to be more stringent then the criteria for raising it. This further decreases the probability that work will be sent to a machine whose load is approaching the nuisance threshold. The sampling interval and the criterion for translating between an instantaneous and smoothed load measure must be customized for different types of machines. Examples of values used in a workstation cluster are discussed in Chapter 6.

The VCE load measurement algorithm (Figure 28) can be customized for particular platforms. For instance, the measurement interval can be modified based on factors such as the desired precision and the frequency which load statistics are updated by the operating system. Also, the definition of low, medium, and high load levels can be adjusted by system administrators. Lowering these thresholds reduces the chance that non-VCE users will experience performance degradations and reduces the processing capacity available to VCE applications. Chapter 6 will discuss possible settings of the algorithm parameters for various platforms.

```
do forever

   sleep 1 Sample_Interval

   compute Load(t_i)

   if Load(t_i) = Smoothed_load(t_{i-1})
      then continue

   if Load(t_i) > Smoothed_Load(t_{i-k}) for k = 1...N
      then Smoothed_Load(t_i) = Load(t_i)

   if Load(t_i) < Smoothed_Load(t_{i-k}) for k = 1...M
      then Smoothed_Load(t_i) = Load(t_i)

   if Smoothed_Load(t_i) ≠ Smoothed_Load(t_{i-1})
      then notify group leader if a period
      of low load has begun or ended

end
```

Figure 28: Load Measurement Algorithm

Time t	Instantaneous Load Measure $Load(t_i)$	Smoothed Load Measure $Smoothed_load(t_i)$
0	Medium	Medium
6	Medium	Medium
12	High	Medium
18	Medium	Medium
24	Low	Medium
30	Medium	Medium
36	Low	Medium
42	Low	Medium
48	Low	Medium
54	Low	Medium
60	Low	Medium
66	Low	Medium
72	Low	Medium
78	Low	Medium
84	Low	Medium
90	Low	Low
96	Low	Low
102	Low	Low
108	Medium	Low
114	Low	Low
120	Medium	Low
126	Medium	Medium
132	Medium	Medium

Figure 29: Load Measurement Algorithm Example

An example of how the algorithm behaves is shown in Figure 29. The example uses a sampling interval of six seconds, an upward bias, N, of two, and a downward bias, M, of ten. These values are appropriate for a Unix workstation (see Section 6.1.3). At time 0 both the instantaneous and smoothed load measures indicate the load on the workstation is medium. The changes in the instantaneous load measure that occur between time 12 and time 30 are considered noise and are not reflected in the smoothed load measure. The ten consecutive low instantaneous readings beginning at time 36 cause the smoothed load measure to become low at time 90. The two consecutive instantaneous readings of medium at time 120 and 126 cause the smoothed load measure to return to medium.

The instantaneous load measure is derived from measurements of the amount of contention for system processor [69] and I/O facilities. Users will perceive poor performance if there is high contention for either the processor or I/O facilities. The instantaneous load measure reflects the higher of the two factors (processor and I/O).

Information Exchange Policy

Objective

The frequency of load information exchange is a central design parameter in dynamic task scheduling. Excessive exchanges can contribute to network congestion. Infrequent exchanges lead to decisions based on inaccurate data. Information exchange is especially important in systems that use process migration. Migrating a running process from one host to another requires significant processing from both the source and destination system. It can also generate a large amount of network traffic. In such an environment extensive information exchanges may be required to insure that costly process migrations will actually result in better overall system performance.

Common Approaches

A wide variety of information exchange policies have been discussed in the literature. Information exchanges can be performed periodically [9, 21] or only when a machine detects that its local load is extremely high or low [34]. Information may be

exchanged with neighboring nodes [84], randomly selected nodes [9, 34] or a central manager [21]. These parameters tend to be established experimentally [9] and are influenced by the amount of scalability desired (the neighbor based approach of [84] is extremely scalable, the centralized system of [21] is intended for use only in local area networks).

Redundant Task Scheduling Approach

The VCE information exchange policy is incorporated into the load measurement algorithm (Figure 28). Machines disclose their loads to their group leader only when a change is detected and persists for a significant period of time. A method for recovering from group leader failures will be presented in Section 5.2.7. Redundant scheduling reduces the need for frequent information exchanges because the penalty for a poor task placement decision is small.

Transfer Policy

Objective

In most distributed systems a transfer policy determines which processes are migrated and when they are migrated. The policy seeks to identify which migrations are most likely to improve workload balance. For instance, migrating a task that is about to complete is wasteful. Determining which migrations will be beneficial is difficult in a network based computing environment.

Common Approaches

Transfer policies are often the most complex part of a load balancing algorithm. Careful design of a transfer policy is essential because an incorrect transfer policy can degrade both system and application performance. For instance, transferring a task just before it completes is counterproductive. Thrashing behavior, where the same job is constantly transferred and is prevented from making progress, must also be prevented [9].

Many ways of characterizing transfer policies are possible. One of the most useful delineations is between sender-initiated and receiver-initiated policies [36]. With a sender-initiated policy a machine which, based on its estimation of the global system

state gleaned from the information exchange policy, determines that it is overloaded, will select a less loaded host to transfer work to. Receiver-initiated policies require lightly loaded nodes to seek out a heavily loaded machine to transfer work from. Because receiver-initiated policies are more efficient in heavily loaded systems and sender-initiated policies are better otherwise [36], research continues into hybrid systems which will provide good overall performance [15, 77].

Another important element of a transfer policy is whether it supports preemptive or non-preemptive transfers. In non-preemptive systems only newly arriving jobs or task queued for execution are candidates for transfer. Non-preemptive systems have less flexibility because fewer transfer options are available [109]. However, preemptive transfers are usually not possible in heterogeneous systems [10, 21].

Various mechanisms have been designed to insure stable transfer policies and prevent thrashing. Commonly a limit is placed on the number of times any single process can be transferred. In preemptive systems a a minimum time can be established for which a process must reside on a particular machine before it can be transferred [9]. This insures that a job will make some progress before each transfer and also prevents the transfer of small tasks.

Redundant Task Scheduling Approach

The VCE employs a non-premptive sender-initiated transfer policy. When a stage of an application becomes dispatchable the execution process obtains from the group leaders lists of lightly loaded machines, Task instances are then transferred to those machines for execution. A sender-initiated approach is used because the number of execution processes, and therefore the number of available tasks, is expected to be much smaller than the number of lightly loaded machines in a wide area network. It would not be practical for lightly loaded machines to attempt to see out execution processes.

The Virtual Computing Environment is inherently heterogeneous, so preemptive process transfers are often infeasable. Some of the characteristics of preemptive process transfer are retained nonetheless. If a task is placed on a machine and the load on the machine increases, the local operating system will reduce the resources allotted

to the task. That is, as the load on a time-shared system increases, the resources available to each task decrease. Since the slower instances of a task are cancelled when the fastest instance completes, the impact of VCE task instances on heavily loaded machines are further reduced.

Running VCE tasks at low priority on privately owned workstations further mitigates the need for a preemptive transfer mechanism. The resources consumed by a low priority task on a heavily loaded machine are minimal [8], so the behavior of the local operating system in starving a low priority VCE tasks as the local load increases has an effect similar to preemption.

Location Policy

Objective

Nodes in a distributed system must cooperate to determine the best location for each task. Once a task is selected for migration by the transfer policy, the location policy selects a target computer for the migration. To maximize the chances that a process will benefit from a migration, the receiving machine must have significantly less load than the sending machine.

Common Approaches

Location policies usually involve some type of threshold value for the difference between the load of the sending and receiving machines. The method for selecting candidate machines depends on the information exchange policy in force. In systems where machines communicate only with a fixed set of neighbors, a heavily loaded machine would transfer a job to its least loaded neighbor if the difference in their loads is significant [84]. Other systems employ random communication patterns [9]. In these systems an overloaded or underloaded machine would randomly probe other machines until a suitable transfer target was found [15].

Redundant Task Scheduling Approach

The location policies needed for redundant task management are simplified. They focus on selecting lightly loaded machines to host the initial placements of redundant task instances. The overhead associated with determining the optimal targets for

migrating running processes of an overloaded host is eliminated.

An execution process uses hints to estimate the computational requirements of the tasks to be placed and determines which group it would prefer to host each task. This decision is based on several factors such as what executables are available and each group's communication distance from the originating computer.

The execution process informs each group leader how many tasks it wishes to place in the group. For supercomputer or mainframe groups, which will rarely contain more than two or three members, the group leader will send to the execution process the address of each lightly loaded machine in the group. For workstation groups the execution process tells the group leader the maximum number of machines it can use. The workstation group leader provides the addresses of the requested number of lightly loaded machines, or the addresses of all lightly loaded machines if less than the maximum number are available.

If the execution process can achieve adequate redundancy within a single group, it will do so. If intergroup redundancy is needed the execution process will employ it.

To run a task on a machine provided by a group leader, the execution process contacts the daemon on the target machine directly and sends it the information necessary to establish an execution context. On privately owned workstations, the daemon's dispatcher starts the task with the lowest possible priority to insure it will only consume excess resources on the machine.

Once a task is placed, no further communication or cooperation is needed. Redundant task scheduling algorithms only need to identify a set of computers which are likely to perform well. By running instances of the task on several computers in this set the performance of the best machine is received without the burden of selecting one individual machine.

5.1.3 Overview of Scheduling Activities

Figures 30, 31 and 32 give an overview of the scheduling activities performed by each type of VCE component shown in Figure 20 (execution processes, group leaders and

group members).

5.1.4 Rationale

Most scheduling systems have one of two goals. They either: (1) manage all the tasks in a distributed system to improve overall performance [9, 67]; or (2) detect idle computers and make their resources available for running batch jobs or parallel programs [6, 21, 60]. The VCE is in the latter category, but its objectives are significantly different from past systems. The VCE seeks to harness available resources scattered across a wide area network and use them as a platform for running very large applications. Most existing systems do not scale well beyond a local area network and have little or no support for heterogeneous systems. Since wide area networks will contain an enormous variety of heterogeneous machines, the VCE supports an application model which allows the user to select a different set of candidate machines for each stage of an application. At run time the VCE attempts to locate machines suitable for each portion of the application. This approach may be a very attractive option for scientists and engineers whose applications require multiple high performance computers with disparate architectures. Accessing such machines remotely is the only way many users can exploit them, so a scheduling system which automatically locates underutilized system and manges the scheduling and dispatching of tasks on them may be valuable.

Because the goals of the VCE are distinctive, its scheduling approach has unique requirements. The scheduling system must avoid creating excessive traffic of scheduling information between widely separated machines. This is accomplished by having machines only disclose load information to group leaders. When A VCE application starts, the execution process contacts only those group leaders whose groups are architecturally suitable for the tasks being dispatched. Geographically near by groups are queried before more distant machines are considered. These measures further reduce network traffic.

```
for each dispatchable task

   determine preferred group

   obtain list of lightly loaded machines from group leader

   if not enough machines were provided
      then select another group

end

for each selected computer

   send dispatching information

end

for each completed task instance

   cancel redundant instances

   determine if any other task has become dispatchable

end
```

Figure 30: Execution Process Scheduling Activities

```
for each load update message received
   update list of lightly loaded machines
end
for each execution process placement request
   determine group capacity
   send list of available hosts to execution process
end
```

Figure 31: Group Leader Scheduling Activities

```
do forever

  sleep 1 Sample_Interval

  compute current Smoothed_Load (Figure 28)

  If Smoothed_Load changed from low to medium
    then notify group leader

  If Smoothed_Load changed from medium to low
    then notify group leader

end

dispatch tasks as directed by execution processes

terminate tasks as directed by execution processes

notify execution processes of task completions when they occur
```

Figure 32: Group Member Scheduling Activites

Redundant scheduling is the simplest way to achieve high performance in a heterogeneous environment. Process migration of jobs between parallel computers is not feasible. Canceling and restarting tasks is costly and can result in poor application performance. This is especially true when an application employs multiple communicating tasks in a single stage. Cancelling one task may require rollback of an entire set of tasks. By placing redundant instances of tasks on machines which are all lightly loaded when the tasks are dispatched, the chances of high performance are maximized.

High levels of redundancy provide only minimal performance improvement beyond more reasonable levels of three or four instances per task. This will be seen in the experimental and simulation data of Chapter 6. The reasons for this are straight forward. Consider an execution process with one task to dispatch and a large group of identical initially idle hosts it can choose from to place redundant task instances. Given the assumption that all the hosts are identical, the performance of the task will only be influenced by differences in load conditions among the hosts. Since all the hosts are assumed to be idle when the placement decision is to be made, the execution process has no basis to expect better performance from any particular host, so the first task instance can be placed at random. Assuming that load fluctuations on the machines are independent, there is a 50% chance that a second host, randomly selected from the remaining machines, will perform better than the first. There is a only a 25% chance that a third machine will be superior to both of the first two. The probability of realizing a performance improvement continues to drop exponentially as the number of machines chosen increases. Of course, real environments are more complex and more variables, such as the amount of speedup achieved when a superior host is selected, must be considered. However, the general principle that the performance returns of redundant scheduling decline rapidly as redundancy levels increase, still holds true.

The largest shortcoming of redundant scheduling is its cost. Running several instances of a task increases processing requirements because all task instances will consume some CPU resources. Only the results of the fastest task instance are made

available to the user or used for further processing, so resources used by slower task instances do not benefit the application. Communication costs also are incurred when inputs must be duplicated and sent to each instance of the target task (Section 5.2). These costs are a further argument against high redundancy levels. For low redundancy levels, these costs are acceptable. Alternative methods for dealing with load fluctuations such as migrating an entire execution image [21] or disassembling, migrating and recompiling a task [102] also introduce considerable overheads.

5.2 Managing Communications Within the VCE

The VCE communication system provides a medium for tasks of VCE applications to transmit and receive partial results to and from other tasks participating in the same application. VCE tasks are expected to be large grain computational components. Even though the running times of such tasks are not dominated by continuous communication, the tasks may have to communicate large amounts of data from time to time (for example, when one stage of a large application completes and its successor begins). Managing these communications are complicated by two factors: (1) application tasks may be redundantly dispatched, so a single message could have multiple senders and multiple receivers, and (2) tasks of a single application could be running on systems which are geographically dispersed. The VCE manages these issues so as to achieve high transparency for both application designers and task implementers. As long as the task implementer realizes high granularity and (whenever possible) overlaps communication with computation operations, and the application designer correctly specifies the application, the VCE delivers high performance on the platforms available at run time.

In addition to supporting intertask communication, a variety of other communication activities are managed by the VCE. Scheduling/dispatching daemons disclose load information to their group leaders. Execution processes locate group leaders

and collaborate with them on task placement decisions. Once task placement decisions are made, the execution processes send dispatching information to the scheduling/dispatching daemons and receive completion notices and other information from them. All these communication operations have different characteristics. They differ on the basis of the proximity of senders and receivers, and the frequency and amount of data to be exchanged. In order to achieve high overall performance, different communication technologies are used for different communication activities. The design of the communication system provides high modularity, so that as new communication technologies are introduced they can be incorporated into the VCE with minimal effort.

5.2.1 VCE Application Communication Issues

The requirements of VCE application intertask communication differ from those of other distributed environments. For each application task the VCE may dispatch several redundant instances. This means that when a task produces an output data object it must be duplicated and sent to all instances of each task designated in the application description as a recipient of the object. When a slower instance of the sending task produces the same data object it must not be presented to the receiving task instances a second time. Further, the system must insure a proper level of consistency among intertask communications. If an application consists of two tasks which exchange data periodically, it is possible for the fastest instances of both tasks to run several exchanges ahead of slower instances. In such a situation the VCE's communication protocols must insure that all task instances receive their input in a coherent and transparent manner.

Before considering the methods used to support intertask communications it is important to examine the characteristics of such communications. Each VCE task represents a major, course grain, high performance component of a large scale (grand challenge) application. Communication among these tasks commonly take one of two

possible forms. Often intermediate results of an earlier application phase are communicated to later tasks. This was illustrated by the linear equation solver application described in Section 4.2.6. Alternatively, application components with different computational requirements may need to run on different machines and exchange data from time to time. The global climate simulation application described in [74], which is structured as separate global atmospheric and ocean models that communicate intermittently, employs this style of communication.

Most of the research on communication support for redundant processes concentrates on supporting broadcast communication [18, 27, 47, 107]. The broadcast model allows individual processes to join a *broadcast group*. Messages addressed to the group are received by all member processes. Different systems provide different levels of support [47]. Some simply guarantee that all messages sent to a group will be received by all group members. Others stipulate that messages sent to the group from a single source will be received by all members in the same order. The strongest level of support involves an assurance that all group members see all messages in the same order. Of course, systems that support stronger consistency criteria require greater complexity and suffer reduced performance [18]. Several techniques are commonly employed by these protocols. Time stamps or sequence numbers can be appended to messages by the sender so the receiving node can process them in the correct order. All broadcasts to a group can be routed through a leader node which forwards them to the member nodes. Or precedence histories can be included with messages to indicate the correct delivery order. Most broadcast protocols use some combination of these techniques.

The requirements of the VCE communication system are somewhat beyond the scope of broadcast protocols. These protocols are concerned with systems where message senders need to reach multiple receivers with a single message. That is, each message has a single source and multiple destinations. The challenge faced by the VCE is to manage communication among redundant task instances where a single message can have multiple senders as well as multiple receivers. This situation has been researched in the past as it relates to process replication in the fault tolerance

domain. In some fault tolerant systems process replication is used as an alternative to checkpointing and roll-back/recovery schemes [5, 31]. In such systems, as in the VCE, each application process can have multiple replicated copies running simultaneously. Of course, input to such a process presents similar issues to those encountered by the broadcast environments described above. However, output handling is more complex because each replicated process will generate identical output, only one copy of which can be presented to each instance of the receiving process. The most common approach to handling this situation is to designate one instance of each process as a leader or coordinator. Output can be routed through the leader which would discard redundant output [38] or the leader can be consulted for permission to send output, which would be denied if the output has already been sent [31]. In either case, performance is constrained by the dependence of all process instances on services provided by only one. These schemes also necessitate special procedures to allow the process instances to recover from the failure of the leader process.

The VCE uses a specially designed protocol to support communication among redundant tasks. The protocol, which is outlined in Section 5.2.3, exploits the unique properties of VCE applications and the VCE run time environment to allow efficient communication with a simpler protocol than would be required for a more general-purpose use.

5.2.2 Components of the VCE Intertask Communication System

The semantics of VCE application construction were outlined in Section 4.2. Task implementations use three simple functions (Section 4.2.5) to perform intertask communications. The execution process uses information in the task and application descriptions (Sections 4.2.4 and 4.2.6) to determine how these communications are to be routed. Each VCE task instance is supported by a VCE communication daemon. The communication daemon is dispatched by the scheduling/dispatching daemon before the task instance is dispatched (Figure 33). At compile time the VCE communication

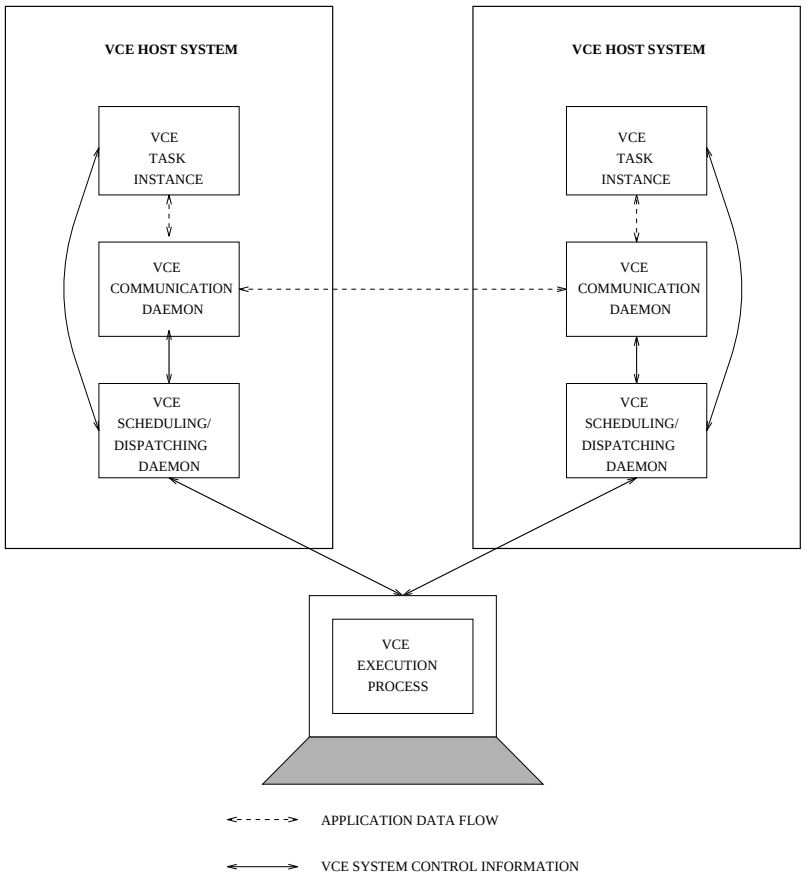


Figure 33: The Function of VCE Communication Daemons

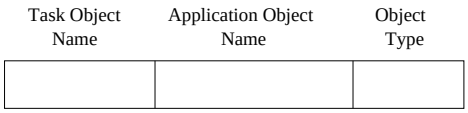


Figure 34: Input Data Table Entry Format

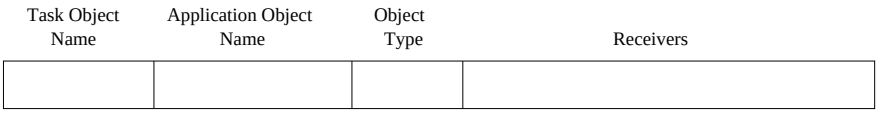


Figure 35: Output Data Table Entry Format

functions are replaced by routines which perform efficient interprocess communication, such as Unix shared segments [96], to exchange data with the communication daemon. The communication daemons do not run in the same address space as the task instance because the communication daemon, in order to ease overlapping of communication with computation, requires an independent thread of control. The communication daemon also needs to exercise memory management independent of the task instance.

When a communication daemon is instantiated it receives from the execution process (by way of the scheduling/dispatching daemon) an *input data table* and an *output data table*. The input data table indicates, for each object that the task instance will receive as input, the name and type that the object was declared with in the task description and the name which it was overridden with in the application description (Figure 34). The output data table contains equivalent information. Additionally, an output data table entry also contains the addresses of all the communication daemons supporting tasks which the output object is to be forwarded to (Figure 35).

5.2.3 Managing Communication Among Concurrent Redundant Tasks

Section 5.1 described how the VCE uses redundant task scheduling to improve application performance. In an environment where the same output object could be

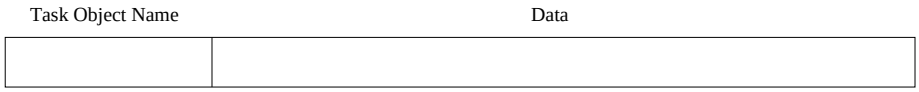


Figure 36: Task Instance Output Message Format

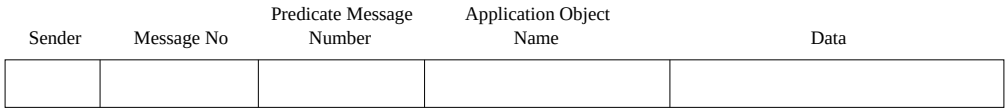


Figure 37: Format of Data Sent Between Communication Daemons

produced by several redundant instances of a single task, the processing required of communication daemons are more complex than simply forwarding output and receiving input on behalf of a task instance. For instance, it is the communication daemon's responsibility to insure that input is presented to each task instance in a coherent manner. Not all the instances of a single task will progress at the same rate. When there are two or more communicating tasks, and each task may have multiple instances, intertask communication must be managed so as to avoid *presentation of duplicate input* and *presentation of input which violates coherence.*

Detecting Duplicate Input

Presentation of duplicate input is a straight forward problem. When a task instance uses the **VCE_send()** function to make a data object available to other tasks, the communication daemon forwards copies of the object to all task instances which require the object as input (as indicated in the application description). At some later time another redundant instance of the same task running on a slower machine (due either to load conditions or processor limitations) may produce the same output object. In the VCE communication system such duplicate messages are detected and discarded by the receiving communication daemon. The algorithm for detecting duplicate data is described below.

When the **VCE_send()** function is executed it builds a data structure which contains the name and contents of the object being output (Figure 36). This data

structure is presented to the communication daemon through shared memory [96] or some other interprocess communication method. The communication daemon uses the output data table to determine what the object name has been overridden with in the current application and where task instances which will receive this object are located. The communication daemon also keeps a count of how many objects it has sent to each receiver. This count enables receivers to detect duplicate messages. The format of a message used to exchange a data object between instances of different tasks is shown in Figure 37. The *Sender* field is filled in with the name from the application description of the application stage producing the output. The *Message No.* field holds the total count of messages sent by this application stage (task) to the receiver. When two instances of the same task produce the same output, the receiving communication daemon will see the same *Sender* and *Message No.* on each. By logging the *Sender*s and *Message No.*'s on input objects as they arrive, duplicate input can be detected and discarded. This methodology implies that extraneous messages are going to be sent across platforms only to be discarded as they arrive. A method for limiting such extraneous traffic will be presented below.

Maintaining Communication Coherence

Redundant communication can lead to incoherence in the following situation. Suppose an application has two concurrently executing tasks, A and B, where A sends an object (a request) to B and receives another object (a result) in return. When the fastest instance of A sends its first request and the fastest instance of B returns a result, the result may arrive at a slower instance of task A before it has made its first request. The communication daemon for the slower instance of task A must not present the result object from task B until the task instance has sent the corresponding request. This is resolved using the *Predicate Message Number* field shown in Figure 37. In this example, the communication daemon for the slower instance of task A would find a 1 in this field of the message sent from task B. The 1 indicates that task B received one object from some instance of task A before sending the message. Since the communication daemon knows that the instance of task A it is

supporting has not sent any objects to task B, it will buffer the input until it can be presented in a coherent fashion. This also enables the communication daemon to discard extraneous output rather than sending it. Since the communication daemon for the slower instance of task A knows that task B has already received one object from a faster instance of task A, when its task instance trys to send an object to task B the communication daemon can discard the duplicate output rather than sending it.

The Communication Daemon Algorithm

When a communication daemon receives input it must determine if the input can be presented to the task instance immediately or must be buffered. Additionally, if the communication daemon can determine that an output object being produced by the task instance it is supporting has already been sent by a faster instance of the same task, then the output can be discarded rather than being sent. The processing required of a communication daemon in managing the communications of the task instance it is supporting (called the *local task instance* for clarity) is summarized in Figure 38 and discussed below.

The communication daemon must maintain one (possibly empty) input buffer for each task sending input to the local task instance. The processing shown in Figure 38 pertains to the maintainence of one such buffer. When input is received by a communication daemon, the *Predicate Message Number* field of the message is checked to determine whether the input may be dependant on any output not yet generated by the local task instance. If not, then the input can be presented to the local task instance or discarded if another copy of the input was previously received from a faster task instance. When a newly received input object is dependant on output data not yet produced by the local task instance, it is added to the appropriate input buffer. Whenever output is produced by the local task instance, the input buffers associated with each task receiving the output are examined. If any predicates associated with buffered inputs are satisfied by the output object, those inputs can be presented. When an output object satisfies the predicate of a queued input message, the output

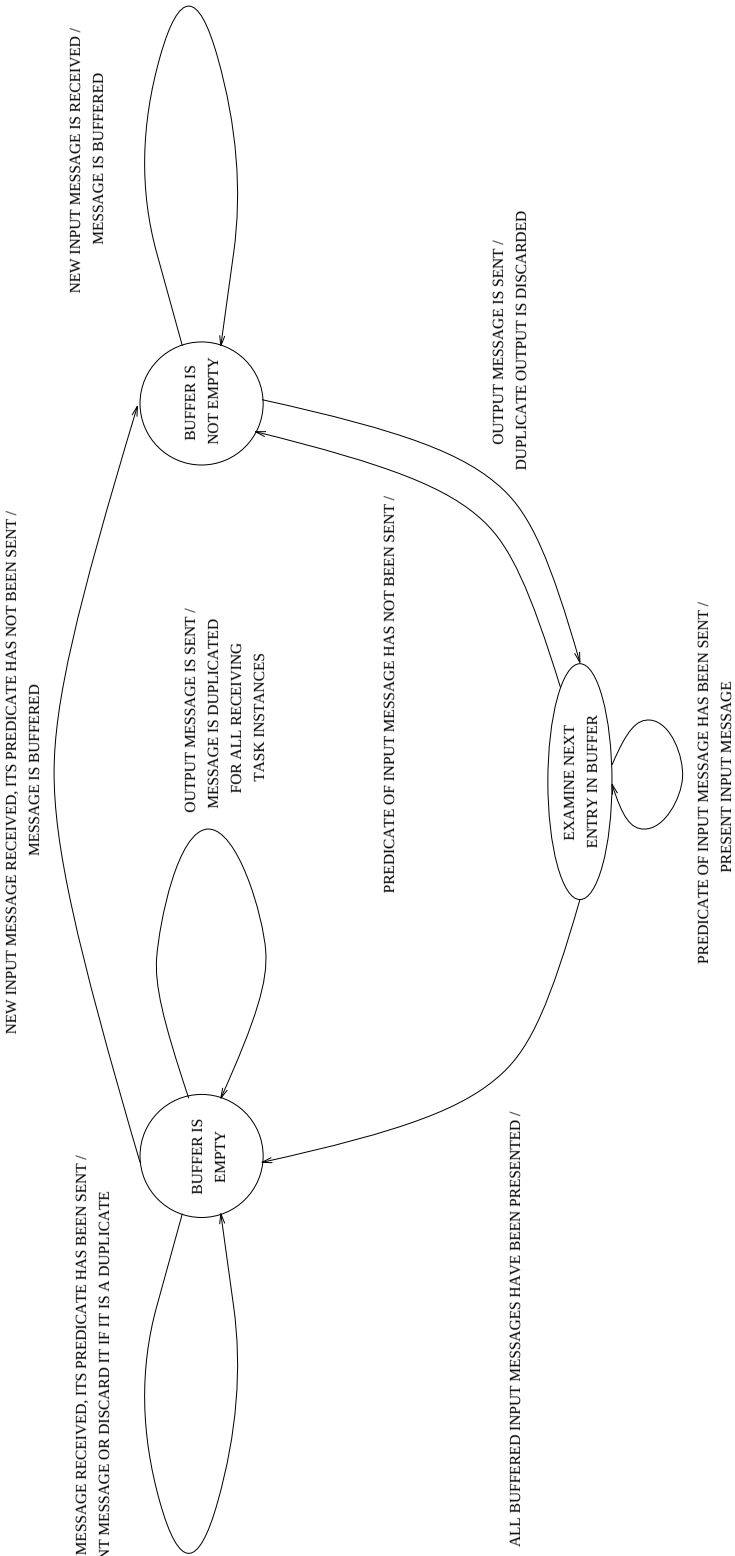


Figure 38: An Algorithm for Managing Communication Among Concurrent Tasks

must be a duplicate of an object produced by a faster instance of the local task. In this case the output object is discarded rather than being sent.

Limitations of the Communication Daemon Algorithm

The algorithm discussed above is not a general purpose algorithm for communication among replicated processes. It is specifically intended to meet the most common needs of VCE applications with maximum efficiency and minimal complexity. The purpose of intertask communication is to allow application components to transmit intermediate results to each other. The VCE application model (Chapter 4) supposes that tasks will use the **VCE_receive()** function to wait for a set of inputs to become available. Then they will use these inputs to perform a computation and invoke the **VCE_send()** function to return the results. In such an environment, the order in which input objects are received is not consequential. That is, if a task needs object A and object B to perform its computation, the application will function correctly even if some instances of the task receive A first while others receive B first. Therefore, the algorithm does not include complex processing to impose such consistency [18].

The algorithm does not support all possible intertask communication patterns. Figure 39 shows five possible patterns. In Figures 39a, 39b, 39c and 39d the VCE communication algorithm does guarantee that no instance of any task will receive input that may be dependent on output it has not yet generated. However, this assurance does not hold for the communication pattern of Figure 39e. That is, the algorithm cannot insure coherent communication in the presence of indirect recursive communication. In Figure 39e TASK A may send data to TASK B which would generate a data object to be consumed by TASK C. If, as a result of this, TASK C returns data to TASK A, some instances of TASK A might see the final result before they have produced the output that generated it. This occurs because the message sent by the communication daemons for TASK C to the communication daemons for TASK A would contain a zero in the *Predicate Message Number* field (Figure 37). So it would appear (incorrectly) to the communication daemons supporting TASK A that the input from TASK C is not related to any prior output of TASK A.

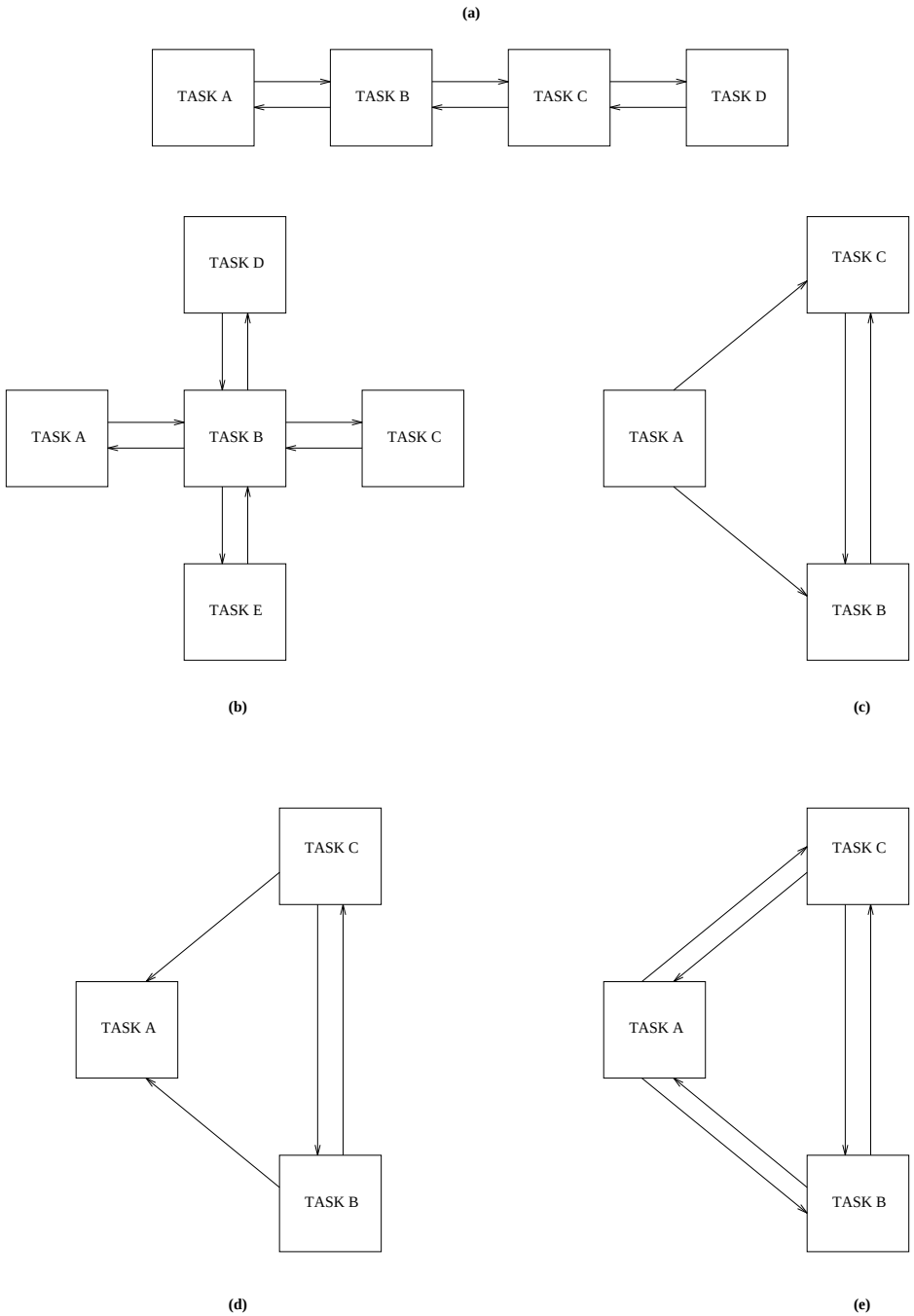


Figure 39: Intertask Communication Pattern Examples

This limitation of the algorithm could be overcome by extending the *Predicate Message Number* field shown in Figure 37 to contain a precedence graph [47]. This data structure would identify all the input received by a task before it generated a data object. Further, the precedence graph would incorporate information included in the precedence graphs of all those prior inputs. Essentially, it would identify all the data that a particular object could have any dependence on. On receiving an input object, the communication daemon would parse the precedence graph and determine if its task instance had not yet created some object that the newly received object might have been derived from. If so, the input object would be buffered until its precedents were satisfied.

It is unlikely that such an elaborate communication algorithm will be warranted. An application which employs indirect recursive communication is likely to be poorly structured and error prone. The software engineering tools in the VCE's software development suite, which are used to formulate the application description, will detect such a situation, alert the user to its presence, and help him or her to replace it with a more well structured application design.

Capabilities of the Algorithm

In spite of the limitations mentioned above, the communication algorithm of Figure 38 is well suited for managing communication among concurrent VCE tasks. One advantage of the algorithm is that a communication daemon supporting a particular task instance does not need to communicate with any communication daemon supporting instances of the same task, as would be required by the more robust algorithms of [31] and [38]. A communication daemon does not need to have any information about the number of redundant task instances dispatched or their locations. Since task instances may be dispatched on geographically dispersed platforms, requiring communication daemons to consult with each other before broadcasting their output could be costly. Also, the chief objective of redundant scheduling is to enable the application to progress at the rate of the fastest machine available. Making one communication daemon the "leader" and requiring it to coordinate all the input and

output for a set of task instances [31] is likely to introduce a costly bottleneck.

5.2.4 Managing Communication Among Predecessor and Successor Tasks

The previous section discussed the methods used by VCE communication daemons to manage communications among concurrently executing application components (tasks). A different protocol must be used to accommodate the exchange of objects between tasks in one application stage and tasks in later stages (Figure 26). This situation is evident in an application description when the **synchronization** specifications indicate that one stage cannot begin until another completes, and the output of the earlier stage serves as input to the later stage. When such a situation is detected by an execution process, it informs the communication daemons supporting the earlier stage that the output objects are to be buffered. This is indicated by placing a special value in the *Receivers* field of the output table entry (Figure 37) corresponding to each output object to be buffered.

Whenever any instance of a VCE task completes, all other redundant instances of the task are terminated. The communication daemons supporting those task instances are also terminated and any (possibly incomplete) data maintained in their buffers is discarded. This means that the output objects needed by the later application stages are maintained only by the communication daemon of the task instance that ran to completion. When all the synchronization constraints of the successor stage are satisfied, the execution process determines where its task instances are to be placed. After the successor task instances and their communication daemons are instantiated, the execution process sends the communication daemon holding the buffered object(s) an updated output table indicating where they are to be sent. Once all the buffered objects have been delivered to the successor tasks the surviving communication daemon of the predecessor task instance is also terminated.

This procedure further minimizes the amount of extraneous message traffic produced by redundant dispatching. In the case of predecessor to successor communications only one communication daemon transmits output to the successor task instances, regardless of the number of predecessor task instances that were dispatched.

5.2.5 Managing Communication Among VCE System Components

Sections 5.2.1, 5.2.2, 5.2.3 and 5.2.4 dealt with techniques for managing the communication of data objects among the various tasks that make up a VCE application. Execution of VCE applications also requires exchange of various types of control information among the components of the VCE itself. Providing efficient mechanisms to manage this communication also contributes to the robustness of the VCE design.

Figures 40 and 41 show schematicly how different VCE communication activities exploit different technologies. Table 3 summarizes the characteristics of these communication activities. The gist of the information in Table 3 is that communication among communication daemons is the situation that most requires the use of high performance communication technology.

The scheduling methods discussed in Section 5.1 minimize the amount of load information that must be transmitted across wide area networks. This information is gathered by an execution process only when it is needed to make a task placement decision. Once the execution process has selected machines to host instances of a task, it relays dispatching information to the scheduling/dispatching daemons on those machines. The dispatching information includes the location of the executables that correspond to the tasks to be run. Task executables may be copied to remote machines ahead of time by the software development suite to avoid a delay at run time. The execution process may also look ahead in an application description to determine where candidate machines to host subsequent application stages are available. If some of these machines do not have local copies of the requisite executables, the execution process can transmit them as earlier application stages are running. This

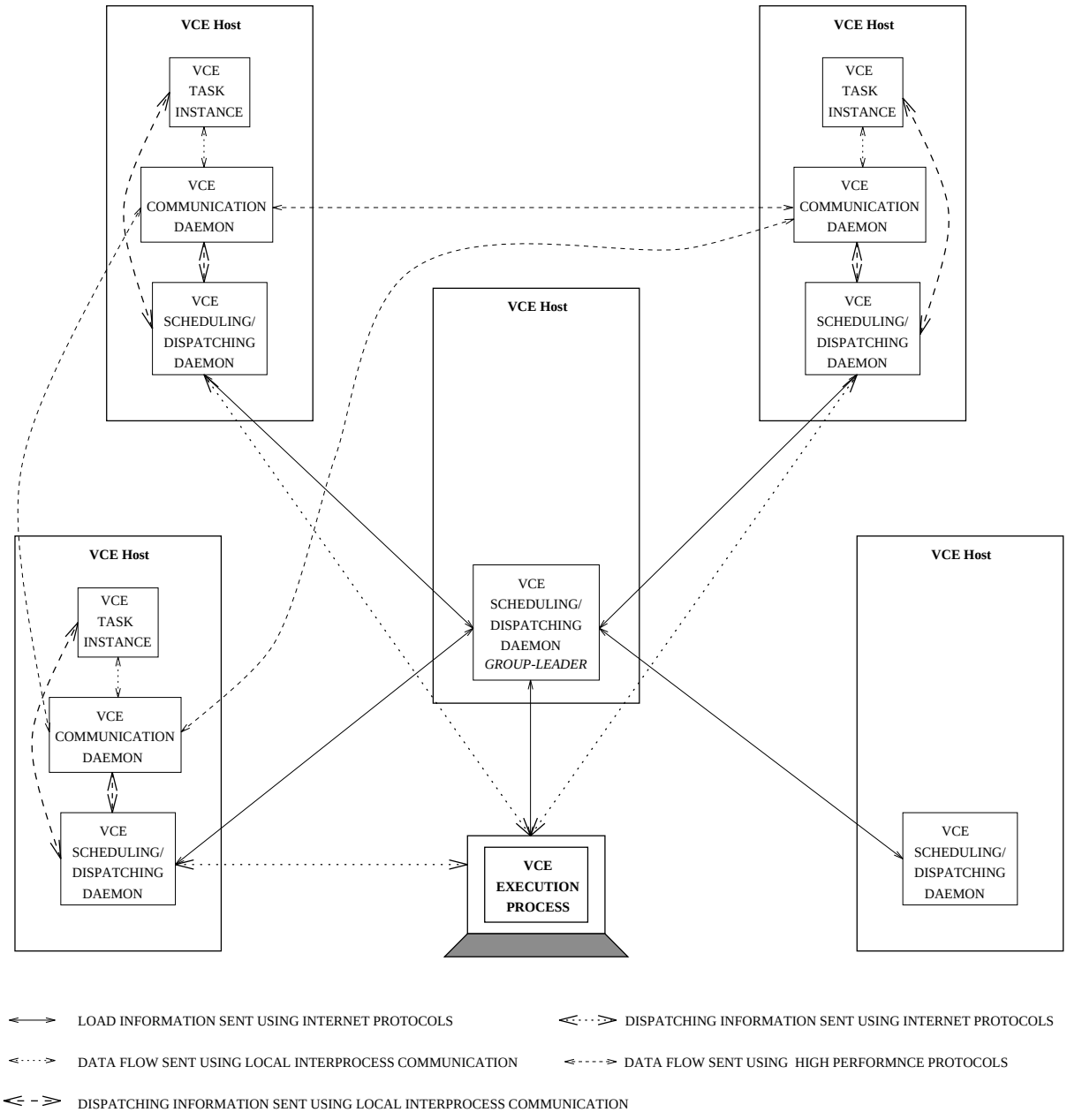


Figure 40: Components of the VCE Communication System

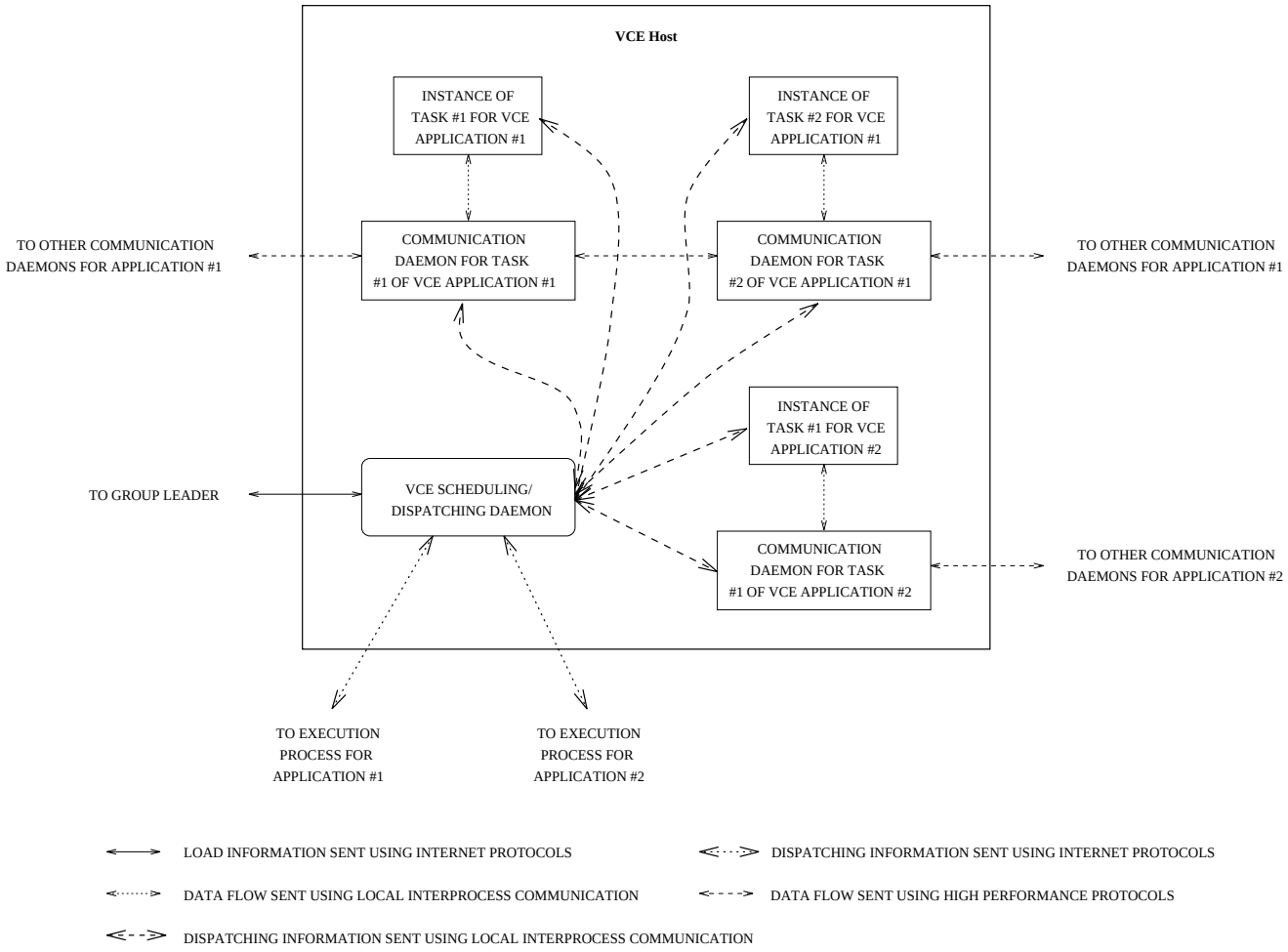


Figure 41: Communication to Support Multiple VCE Tasks on a Single Host

Table 3: Communication Characteristics of VCE Components

Parties	Geographical Seperation	Frequency	Size
Group Leaders and Scheduling Daemons	Small	Less than twice a minute	10's of bytes
Execution Processes and Group Leaders	Great	At the beginning of each application stage	10's of bytes
Execution Processes and Scheduling Daemons	Great	At the beginning and end of each application stage	100's of bytes
Scheduling Daemons and Communication Daemons	None	At the beginning and end of each application stage	100's of bytes
Communication Daemons with Each Other	Great	Intermittently or continuously	Terabytes

affords an opportunity for overlapping VCE communication activities with application processing.

Executables for several different types of communication daemons reside on the local file systems of each platform. These communication daemons differ in the types of protocols they employ. The execution process has considerable information about the properties of the communications that will be carried out by each communication daemon. For instance, it knows the number and location of task instances that will be exchanging data and the size of the objects being communicated. The synchronization specifications of the application stages also indicate whether peer-to-peer or predecessor-to-successor communications will be performed. The execution process can use all this information to select which of the available communication daemon implementations will give the best results for a particular task instance.

The VCE simplifies the process of enabling applications to exploit advances in communication technology. If a new high performance protocol or channel becomes available, VCE applications can be enabled to use it by implementing a new communication daemon and updating the execution processes with information on when use of the new daemon would be beneficial. Modifications to task implementations, task descriptions, and application descriptions would not be required.

The VCE design is not inordinately sensitive to the performance of communication protocols used to carry control information between group leaders, scheduling/execution daemons and execution processes. Each stage of a VCE application is a course grain long running computation. Delays on the order of a few seconds for the execution process to gather load information, select task hosts and disperse dispatching directives will not represent a critical performance issue as long as the dispatched tasks have running time on the order of minutes of hours.

5.2.6 Managing Location Information in the VCE

In all distributed systems, except those that are so static that machine addresses can be hard coded into system components, a mechanism must be provided for system

components to locate each other. In the VCE, in spite of the fact that participating machines may be spread across a wide area network, efficient mechanisms for managing location information have been established.

Section 5.2.2 discussed *Output Data Tables* (Figure 35) which are supplied to communication daemons and indicate how the locations of other communication daemons which are to receive their output. Communications daemons also communicate with a scheduling daemon and the task instance they are supporting. Both these components reside on the same machine as the communication daemon so dynamic location mechanisms are not needed.

Scheduling/dispatching daemons must divulge their load status to their group leaders. When a VCE scheduling/dispatching daemon is installed on a machine a file is placed on the local file system which includes a list of possible group leaders. The scheduling/dispatching daemon attempts to contact these machines starting at the top of the list. When one is found to be hosting the group leader this fact is noted for future reference. In many cases group members will have access to a common file system which simplifies administration of the group leader list. Methods for recovering from failures or withdrawal of scheduling/dispatching daemons or group leaders will be discussed in Section 5.2.7.

The most difficult location problem involves how execution processes can locate group leaders at run time. For each task in a VCE application, the execution process must select a group, or set of groups, to host it. To make this decision the execution process must have access to information about the location and architectural capabilities of each group participating in the VCE. This information is maintained in a series of World Wide Web servers geographically dispersed across the network. When an execution process begins, it contacts the nearest such server and down loads the information. When a task placement decision is to be made, the execution process consults this list and determines which groups would be most suitable for each task. Group leaders of suitable groups are contacted, starting with those that are geographically closest, to determine their capacity in light of their current transient load conditions. The execution process continues contacting group leaders until the

desired redundancy level is attained.

This method for locating group leaders requires that they register with the World Wide Web servers, and provide the required architectural information, when they are ready to begin fielding requests from execution processes.

5.2.7 Managing Dynamic Changes in the VCE

A Virtual Computing Environment is inherently dynamic. Machines may become unavailable because of hardware failures, network failures or decisions by local system administrators to temporarily remove resources from the VCE. Several strategies are employed by the VCE to cope with this dynamism.

While redundant scheduling is used in the VCE primarily as a load balancing methodology, it also imparts certain fault tolerant characteristics on VCE applications. When a machine hosting a VCE task instance fails, it is equivalent to the machine becoming infinitely slow. That is, instances of the same task running on other machines will complete and furnish results to concurrent tasks, subsequent tasks, or the user. Therefore, no special actions are needed on the part of an execution process to recover from a node failure.

Group leaders can minimize the impact of node failures with some rudimentary processing. The scheduling and dispatching algorithms discussed in Section 5.1 stated that when the load status of a machine changes, that is, when the load on a lightly machine increases or the load on a heavily loaded machine decreases, it notifies its group leader of the change. The group leader maintains a list of lightly loaded machines. If a heavily loaded machine fails, no action on the part of the group leader is needed. Since the machine was not on the group leader's list of available machines, no attempt will be made to dispatch VCE work on it. When a group leader is contacted by an execution process, the group leader provides a list of machines that VCE tasks can be dispatched on. Before the group leader places a machine on the list it contacts the scheduling/dispatching daemon on that machine to verify that it is still operating. If it is not, it is not included in the list returned to the execution process and it is

removed from the list of lightly loaded (available) machines.

Lightly loaded machines send test messages to the group leader at random intervals to verify that it has not failed or withdrawn from the VCE. If the group leader does not respond, the scheduling/dispatching daemon that detects it consults the group leader list and begins contacting these machines. The top most machine which has a functioning scheduling/dispatching daemon becomes group leader and contacts the World Wide Web servers to correct their group leader lists.

Chapter 6

VCE Design Validation

In order to demonstrate that the VCE design is implementable and demonstrate and quantify its value, two prototype systems and a simulation model have been constructed. The focus of these experimental efforts has been on studying the scheduling, dispatching and load balancing methodologies described in Section 5.1. An interpreter for the *application description framework* syntax described in Chapter 4 has also been implemented and used as a component in one of the prototypes and in the simulator.

All three of these systems serve different purposes in validating the VCE design. The first prototyping effort involved building a small version of the VCE Execution Suite on a workstation cluster. This prototype included implementations of all the major Execution Suite components described in Section 3.4.1 (scheduling/dispatching daemons, execution processes, group leaders, etc.). The goal of this first system was to gain experience with the construction of these components and refine their interfaces. It also was used for experimentation with the VCE load measurement algorithm (Section 5.1.2) and to experiment with redundant scheduling (Section 5.1.2) and low priority scheduling (Section 5.1.2).

The second prototype demonstrates the value of the VCE scheduling and application description techniques in managing the execution of multi-stage applications on multiple high performance parallel computers. This prototype includes a more robust execution process which can parse application and task description files.

A simulation model was built to allow for more controlled load balancing experiments than could be performed on the prototype systems using shared hardware. The simulation system was developed using an object oriented approach and utilizes a processor simulation algorithm which more closely matches actual processor behavior than many commonly used techniques (see Section 6.3.2).

This chapter will describe the prototypes and simulation system. Preliminary performance results gathered from experiments with all three systems will be discussed.

6.1 A Workstation Cluster Based Prototype

The simplest possible VCE configuration involves a single cluster of homogeneous workstations. Such a configuration was used to construct the first VCE prototype [90]. While this configuration does not include some features of the VCE, such as exploitation of multiple high-performance parallel machines, it was useful to establish designs for the Execution Suite components and implement and exercise their interfaces. Further, it may be beneficial to demonstrate that the Execution Suite is capable of "scaling down". That is, that it can be useful in very small systems as well as very large ones. Finally, since networked workstations are common and widely underutilized (see Section 1.1.3), demonstrating that the VCE can effectively use their excess capacity will make it attractive to a larger user community.

6.1.1 Scope of the Prototype

A schematic showing all the components of the VCE was given in Figure 2. Figure 20 showed just the components that constitute the VCE execution suite. The goal of the VCE workstation cluster prototype was to implement a single group and execution process. These components were used to validate the scheduling algorithms discussed in Section 5.1 and demonstrate their applicability to a workstation cluster environment. The application description framework discussed in Chapter 4 was not implemented for this prototype. Instead, the dispatching sequences of three test

applications (Section 6.1.5) were hard coded into the prototype execution processes.

To allow for rapid prototyping, the system uses the Isis Distributed Toolkit [19] for communication among the scheduling/dispatching daemons, group leader and execution process rather than the World Wide Web technology mentioned in Section 5.2. Isis is particularly well suited to a workstation based VCE implementation. It supports the concept of a *process group*. When processes join an Isis process group they are supplied with information about other members of the group. It is natural to implement a VCE scheduling/dispatching daemon group as an Isis group. When a process joins the group it can use the information provided by Isis to determine which scheduling/dispatching daemon was initialized first. This daemon serves as the group leader. Isis also allows non-group members to broadcast messages to a group. This function is used by the execution process to send its machine requests to the group leader.

Because this prototyping effort was focused on studying scheduling and dispatching issues, the inter-task communication capabilities described in Section 5.2 were not implemented. Application components exchanged data using Isis functions or shared files.

6.1.2 Low Priority Dispatching

The VCE design seeks to insure that conventional (non-VCE) users of workstations do not experience poor performance because their machines are being used to host VCE tasks. VCE tasks should only consume excess workstation capacity. This is accomplished by dispatching VCE tasks on shared workstations at very low priority (Section 5.1.2). VCE tasks are only placed on hosts that are lightly loaded when the task is initiated. If other workstation users increases their use of the machine, the operating system will reduce the amount of resources made available to the low priority VCE task [8] so that its imact on the workstation users will be negligable. Of course, this causes starvation of the VCE task. Redundant scheduling is employed so that the VCE application user will experience performance equivalent to that provided

by the least loaded host.

6.1.3 Workstation Load Measurement

Section 5.1.2 introduced the general VCE load measurement algorithms and their underlying philosophy. These algorithms involve taking periodic snap shots of the processor load state, using several consecutive snap shots to derive a "smoothed load average" and determining how heavily the machine is loaded relative to its *nuisance threshold* (the load level at which users notice poor performance). This section will discuss those algorithms were implemented for the VCE workstation cluster prototype.

A workstation's run queue length is generally a useful measure of system load [69]. Experience with the VCE has confirmed the utility of monitoring the workstation run queue length, however, by itself, it does not adequately characterize the state of a workstation. Specifically, a machine executing several I/O bound jobs may experience performance degradations. These degradations are caused by competition for I/O facilities rather than competition for the CPU and are not detected by monitoring the length of the run queue. A load measurement limited to examination of the run queue might incorrectly indicate that a workstation being used for local program compilation or document processing had excess capacity. To compensate for this deficiency, the VCE monitors both the amount of interrupt activity (which is indicative of the I/O load) and the run queue length.

Before a machine is added to the VCE workstation cluster prototype, a one time experiment is performed. A large number of I/O bound jobs are run concurrently on the machine. The number of system interrupts resulting from this load is noted and establishes the I/O capacity of the machine. This experiment is performed only once for each machine. The results are recorded and used subsequently by the scheduling algorithms. Because the experiment measures a hardware characteristic (I/O capacity) it does not need to be repeated.

After a machine is added to the VCE the *rstat()* function is called every six

seconds to sample the average run queue length and number of system interrupts which occurred during the preceding interval. If the average length of the run queue during the interval was greater than two (indicating significant CPU contention) or I/O activity was greater than two-thirds of capacity (the capacity was estimated experimentally as described above) then the load on the machine is considered high. At this load level performance degradations are noticeable. Conversely, if the average run queue length is less than one (indicating the CPU was idle for some period) and I/O activity was less than one-third of capacity then the load is considered low. When load conditions are between these two extremes the load is considered to be adequate.

Figure 28 showed the general algorithm for processor load measurement in the VCE. The algorithm involves taking periodic snapshots of the load on the machine (*instantaneous* load measurements). A *smoothed* load measure is adjusted any time several consecutive instantaneous load readings are higher or lower than the current smoothed load measure. For workstations the algorithm is customized as follows:

- The sampling interval is six seconds.
- The smoothed load measure is raised if the instantaneous load measure is higher for two consecutive intervals.
- the smoothed load measure is lowered if the instantaneous load measure is lower for 10 consecutive intervals.

Six second sampling intervals are used for the following reasons:

- Some Unix systems only update load measurements every five seconds, so more frequent sampling may not be useful.
- Run queue lengths are reported in terms of a one minute average. Values for shorter periods are estimated by extrapolating between consecutive samples. Sampling every tenth of a minute simplifies this computation.
- Longer sampling intervals increase the chance of placement errors because it takes the system longer to notice and react to changing conditions.

As described in Section 5.1.2, The algorithm is designed to react to increases in local load faster than decreases. Increases in load cause performance degradations for local tasks. The VCE must act quickly to avoid starting remote tasks on a machine where the local load is increasing. However, after a period of high or medium load conditions a machine should not be considered eligible to host remote tasks until a sustained period of low load conditions is observed. The criterion for raising the smoothed measure is observation of two consecutive intervals of higher load. This was selected because it insures that, if a large job is started on a workstation, the workstation will be removed from the group leader's list of eligible machines within 12 seconds. The machine will accept tasks again after 10 consecutive intervals of low load.

6.1.4 Task Placement

In a workstation group the group leader maintains a list of machines with low load. When an execution process wishes to place tasks in a group it sends a message to the group leader with the number of machines needed. The group leader responds with a list of available machines.

6.1.5 Experimentation

Three test scenarios have been constructed to exercise various capabilities of the prototype. One is a numerical quadrature system. This application was used mainly to test the dispatching and communication facilities in the VCE. The execution times of the constituent tasks were only a few seconds, so this application did not provide much insight into the performance characteristics of the scheduling system.

In order to gauge the performance impact of redundant scheduling, a benchmark application was constructed. The application is made of two tasks. Both tasks execute an identical number of integer arithmetic operations and exit. The sizes of the tasks were chosen so they would be course grain and run on the order of hours on workstations. The performance data given in Table 4 summarizes data gathered

Table 4: Performance Benchmark

	Average Run Time (Minutes)	% Improvement
Random Placement	150	0
Selective Placement	108	28
2-Way Redundancy	102	32
3-Way Redundancy	85	43
4-Way Redundancy	82	45
5-Way Redundancy	82	45

from more than 30 runs of the benchmark with various levels of redundancy. The random placement entry in Table 4 corresponds to a policy of running one instance of each task on a randomly selected workstation. This policy occasionally results in the selection of a very heavily loaded machine and results in poor performance. Placing tasks on machines that are initially lightly loaded ("Selective Placement" in Table 4) improves performance considerably. Redundant task placements improve performance even more. The data shows that raising redundancy from two machines per task to three yields a substantial performance improvement. This is because, with three machines per task, the probability that at least one instance of each task will be running on a lightly loaded machine for a substantial proportion of its execution time is high. The performance benefits of using more than three machines per task are minimal.

The third application used to test the prototype is a text analysis system. Each task in the application examines a different text database and determines how many occurrences of a target string it contains. The tasks are of varying sizes. Two process databases containing about 240K bytes of text. The third searches 160K bytes of text.

This text processing application is representative of a class of applications which will be common in network based computing environments. The expansion of the

Table 5: Text Processing Application Performance

	Average Run Time (Minutes)	% Improvement
Random Placement	81	0
Selective Placement	57	30
3-Way Redundancy	48	41

internet will make vast text databases widely available. The VCE, with its redundant task scheduling capabilities, may provide an attractive platform for applications that process or analyze such data.

This application functions well in a redundant execution environment. All tasks accept their input from read-only files. Contention for the input files does not degrade performance because the distributed file system caches the data efficiently and the ratio of processing time to I/O time is high. When each task completes, it writes a single record of results. If one task writes its output and a second instance of the same task writes its output before the system can terminate it, this does not cause an error. The second write will overlay the first with identical correct data.

Table 5 shows average performance results when the application was run with random task placement, selective task placement and three-way redundancy.

6.2 A Multiple Parallel Computer Based Prototype

A somewhat more relevant VCE configuration is one that includes the participation of multiple parallel supercomputers. A VCE prototype has been developed to manage execution of multi-stage VCE applications in such an environment [92]. As in the workstation cluster based prototype, the focus is on studying the load balancing approaches layed out in Section 5.1. This prototype also implements the *application description framework* syntax of Chapter 4. The application description interpreter

makes it easier to experiment with more complex applications and gives a basis to judge the expressiveness and implementability of the syntax.

6.2.1 Scope of the Prototype

Because Isis is not available on some of the platforms used in this prototype, a different message passing tool, p4 [22], was used for communication among the execution suite components. P4 is not as conducive to the VCE structure as Isis. Isis supports a group based communication model which lends itself naturally to the VCE design's process group orientation. The communication paradigm supported by p4 is less dynamic. P4 applications are structured as follows: a master program is started which then starts a group of slaves that can communicate with each other and with the master. The number of slaves is constant and must be established when the master program is initialized. In the VCE the scheduling/dispatching daemons run continuously and the execution process communicates with them when a user requests application execution and as the application executes. The daemons are not slaves to the execution process nor is the execution process a slave to the daemons.

For the purposes of this prototype study, the p4 communication functions were supplemented by Unix sockets [96] calls to overcome the limitations of p4's master/slave communication model. Specifically, the load monitoring routines are incorporated into separate daemons. These daemons perform the load monitoring algorithm and provide the current load state to client programs using a sockets interface. When a VCE application is to be run, the execution process starts a scheduling/dispatching routine on each machine. The scheduling/dispatching routine obtains its host's current load conditions from the load monitoring daemon and notifies the execution process if the load conditions will permit it to host a VCE task. This structure is inferior to the VCE design specified in Chapter 3 because excessive overhead is incurred when the execution process is initiated. Also, if several users were executing VCE applications simultaneously multiple scheduling/dispatching routines would be running on each host.

Of course, in production VCE systems VCE components will communicate using more robust distributed object technologies [1, 17, 59, 82]. Since the primary objective of both prototypes was to study scheduling and load balancing issues, fully implementing the communication design of Section 5.2 was not necessary.

While the communication daemons that will support intertask communication in the VCE are not implemented in this prototype, some communication facilities beyond those present in the workstation cluster based prototype are included. Specifically, this prototype manages data files on behalf of application tasks. When a task is redundantly dispatched, any files that are indicated by the application description as being input to that task, are automatically replicated in the local file systems of each host. Similarly, when a task instance completes, its output files are made available to subsequent application stages. Partially completed files of cancelled task instances are discarded. This makes redundant dispatching transparent as regards file I/O operations.

6.2.2 Hardware Platforms

Several supercomputers and high performance workstation clusters at the Northeast Parallel Architectures Center (NPAC) have been included in this prototype system. The NPAC environment consists of a wide set of state-of-the-art multi-computer systems (see Figure 42). The platforms used are briefly described below:

IBM SP-2: The SP-2 consists of a cluster of 16 RISC/6000 370 nodes interconnected by a crossbar switch (Allnode). Each node runs at a clock rate of 62.5 MHz.

ALPHA/FDDI: The ALPHA/FDDI configuration consists of 8 DEC ALPHA workstations interconnected by a high performance (100 Mbps) backbone composed of dedicated, switched FDDI segments. The ALPHA nodes have a clock rate of 150 MHz.

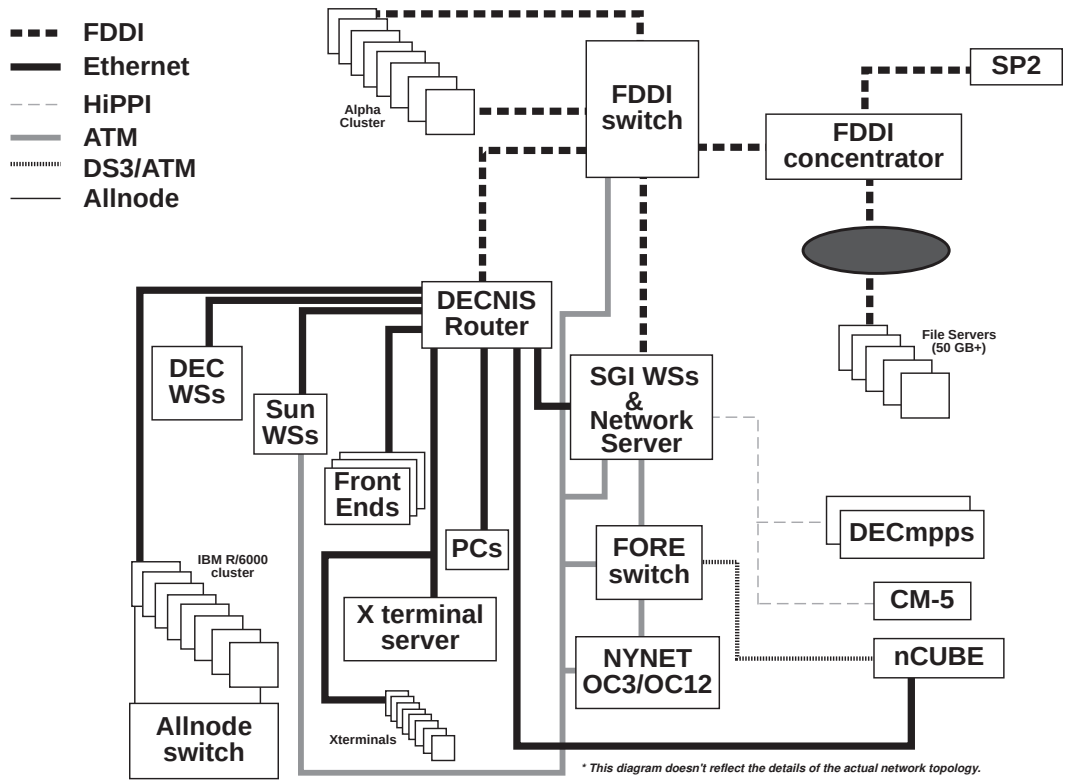


Figure 42: Computing Environment at NPAC

ATM LAN: This configuration consists of 2 SUN SPARCstation IPXs and 3 SGI workstation communicating over an ATM Local Area Network using ATM FORE ASX-100 switch. SUN IPX nodes operate on an approximately 40 MHz clock. Host computers are connected to the ATM switch through Fore's 200 series adaptors. Each adaptor has a dedicated Intel i960 processor (running at 25 MHz) to support segmentation and reassemble functions and to manage data transfer between the adaptor and the host computer. The 200 series adaptors also have special hardware for AAL CRC and special-purpose DMA hardware.

6.2.3 Applications

Three distributed applications have been implemented on the multiple parallel computer based prototype and have been used to test the runtime support.

Linear Equation Solver

A linear equation solver constructed from a set of general purpose linear algebra tasks (shown in Figure 26) was discussed in Chapter 4.

Multi-target Tracking System

The Multi-target tracking system demonstrates the multi target tracking capabilities that is required by a Battle Management Command Control and Communication System [56]. The multi target tracker, shown in Figure 43, is designed to provide an estimation of launch vehicle parameters for individual targets/missiles in multi-target scenarios. The tracker receives input in terms of launch sites for missiles or targets. The launch sites are specified in terms of latitudes and longitudes. This information is fed to two focal plane tracking (FPT) modules (2 dimensional tracking) at 5 second intervals. The focal plane tracking modules process this data using kinematic filtering algorithms and track pruning and prediction algorithms. The output of this module is an initial prediction of trajectories of launched missiles. This data is then fed to a three dimensional tracking system which uses the data from the two focal plane tracking modules to prune duplicate tracks (if any), extend existing tracks, prune bad tracks and initiate new tracks. The output of the system is a list of target trajectories.

JPEG Compression/Decompression

JPEG (Joint Photographic Experts Group) is an emerging standard for image compression. JPEG standard aims to be generic and can support a wide variety of applications for continuous-tone images. We have used data parallel programming model to implement a distributed JPEG algorithm on a cluster of workstations. In this implementation half of the computers participate in compression of an image file

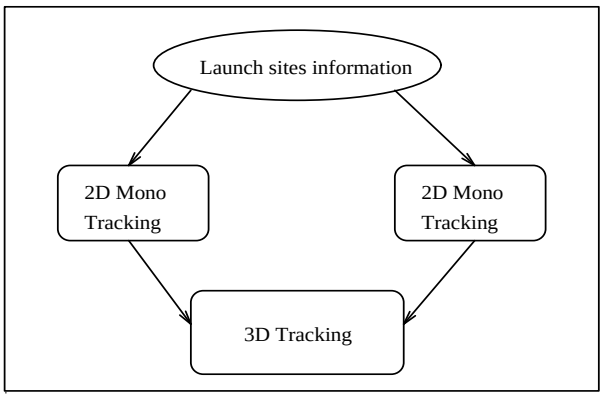


Figure 43: Multi-Target Tracker System

while the second half reconstruct the compressed image. The image to be compressed is divided into N/2 equal parts (where N denotes the number of processors) by the master process and then shipped to one half of the processors. Each processor performs the sequential JPEG compression algorithm on its portion of the image. After compression the processors send the compressed image to another set of N/2 processors which perform the decompression. Once decompression is done, the results are sent back to the master process which combines them into one image. Consequently, this algorithm involves five stages *viz.* distribution of uncompressed image , compression of the image, transmission of compressed image, decompression of the image, and finally combining the decompressed images.

6.2.4 Performance

Of the three VCE applications described in in Section 6.2.3 the linear equation solver is most representative of the kind of application that will benefit from development and execution in the VCE. It is constructed from tasks which could be reused in other applications and for large data sets it has a long running time. So the developer benefits from the application description framework and the user benefits from the automatic runtime scheduling and dispatching capabilities.

A series of performance tests using the linear equation solver application were

performed to determine under what conditions running an application in the VCE with the scheduling protocols described in Section 5.1 would be better than running it in a single computer. The IBM SP2 and the DEC ALPHA workstation cluster have similar performance characteristics [55], so often a user who has an application that could run on either platform would simply choose one at random. Observations in [55] suggest that even a moderate difference in load conditions at runtime can effect performance enough to impose a considerable penalty for choosing the wrong machine. Performance experiments done with the VCE support these findings and show that redundant scheduling will reduce turnaround times in many situations.

In order to obtain controlled testing conditions, testing was performed on the DEC ALPHA cluster and the IBM SP2 when transient load conditions were low. A load generating utility was used to slightly elevate the load on one or the other as the tests were performed. The load generator allows load conditions to be varied between *low* and *moderate*. It places two dummy processes on each node in the configuration for a randomly determined period to simulate an interval of high activity. These *busy* periods are alternated with randomly determined *idle* periods where no artificial load is placed on the machine. When *low* load is specified, the busy periods vary from four to eight minutes while the idle periods are between 20 and 30 minutes. To simulate *moderate* load, idle periods are between four and eight minutes and busy periods are from 20 to 30 minutes.

The results of performance tests run using the linear equation solver application to solve a system of 1024 equations are shown in Table 6. The table shows application performance on each platform under low and moderate load conditions. It also shows performance where instances of the application tasks are placed on both machines ("Redundant Scheduling") or placed only on the machine that is less loaded at the time the tasks start ("Selective Scheduling"). The table shows the degradation which results when just two other processes per node are competing for computing resources. Neither redundant nor selective scheduling will have a significant impact on performance when the loads on all the candidate machines are roughly equivalent. That is, if all machines are heavily loaded, or all machines are lightly loaded, there

Table 6: Performance of the Linear Equation Solver Application

	Load	Execution time in seconds
SP2	Moderate	1153
	Low	849
ALPHA	Moderate	1440
	Low	732
Reduntant Scheduling	SP2-Moderate ALPHA-Low	741
	SP2-Low ALPHA-Moderate	868
Selective Scheduling	SP2-Moderate ALPHA-Low	744
	SP2-Low ALPHA-Moderate	886

is little a scheduling system can do to improve performance. However, when some candidate machines are less active than others, both the scheduling approaches studied tend to prevent the performance loss that could be encountered with a random placement.

Redundant scheduling yields slightly better performance than selective scheduling. If there is a significant load change after a task is placed with selective scheduling performance will degrade. Redundant scheduling insures that the best possible performance will be obtained because all candidate machines run the tasks and the result produced by the fastest machine is returned to the user or made available for subsequent computations. Of course, redundant scheduling incurs higher costs for the application because the computations performed by some machines are discarded. These costs are somewhat offset by eliminating the overhead required for gathering load measurements.

Both of the scheduling protocols insulate the user from the effects of poor random placements without imposing the considerable overheads associated with process migration or checkpointing techniques.

In production Virtual Computing Environments it is expected that the choice between redundant and selective task scheduling would be based on the nature and frequency of VCE jobs to be executed. If VCE applications are submitted only occasionally, or tend to be of short duration then the use of redundant scheduling would be preferred. If many large VCE jobs are to be executed then selective scheduling would provide lower costs and preserve improved performance.

6.3 A VCE Simulation Model

The VCE design incorporates many features which are not commonly found in distributed computing environments. These features include a simplified load measurement and information exchange policy, group oriented load balancing and redundant scheduling techniques. Because the approaches and technologies employed in the VCE are novel, a variety of experiments have been performed to understand the characteristics of the system and quantitatively measure its utility. Two different avenues have been used to gather data on the system's performance. A series of experimental prototype systems have been implemented and a simulation model has been developed. This section describes the simulation model which has been constructed to study VCE scheduling and load balancing policies. The simulation is constructed to model each processing element of parallel computers and each workstation in workstation clusters. Multiple supercomputers and workstation clusters can be modeled in each simulation run. The model employs an object oriented approach where parallel computers and workstation clusters are formed as aggregations of simple processor objects. Similarly, parallel applications are modeled as aggregations of sequential tasks and VCE applications are constructed as aggregations of parallel applications. Simulated processors use a round robin scheduling policy which is more realistic than the first-come first-served approach used in many other simulation studies. The model

of a parallel application relaxes the common assumption that task arrivals at each node are independent.

6.3.1 Evaluating Distributed System Designs

There are three techniques which are commonly used to evaluate distributed system designs: analysis, simulation and prototyping [57]. Both simulation and prototyping have been used to explore various aspects of the VCE design. Mathematical (queuing) studies have not been performed. The limitations of mathematical modeling make it particularly difficult to use in studying VCE configurations. For instance, analytical models characterize processors as servicing tasks on a first-come first-served (FCFS) basis [57, 66, 77, 93]. Assuming FCFS scheduling protocols allows a distributed system to be characterized as a network of M/M/1 queues. The queuing theoretic properties of such systems are well understood [57]. Unfortunately, the FCFS assumption usually does not hold because most operating systems use round robin or prioritized scheduling [100]. Inappropriately assuming that distributed processing elements provide FCFS service can lead to invalid results [34].

Another shortcoming of mathematical modeling is that arrivals at each node of a system are assumed to be independent [57, 66, 77, 93]. However, in a high performance computing environment where parallel applications may have components distributed across many nodes, this assumption also may not hold.

These questionable assumptions commonly made in analytical models of distributed systems (FCFS node processing and independent task arrivals) are also made in many published simulation studies [3, 4, 39, 50, 79, 84, 95, 110]. However, the VCE simulation model described below incorporates more reasonable characterizations of these parameters. In this way, simulation provides a more flexible means for studying system performance than does rigorous mathematical analysis. Parameters like scheduling discipline and task arrival behavior can be modified to reflect conditions of the systems of interest in a simulation. With analytical models these assumptions are required by the queuing theory which underlies the methodology [57, 93].

It might seem that prototyping experiments alone would be sufficient to characterize and validate the design of the VCE. This is not the case. The performance of VCE applications, as well as their impact on the non-VCE applications running on the same hardware, must be precisely measured. Performing controlled experiments on hardware is difficult. Acquiring several parallel computers simultaneously and reserving them for the exclusive use of this research effort is not practical. Further, the instrumentation needed to measure the quantities of interest could themselves impact the outcomes of the experiments [57]. Finally, since network attached high performance computers are expected to be more common in the near future, it is desirable to demonstrate the scalability of the VCE computing approach. Obviously, it is not possible to show experimentally that the VCE (or any computing system) will perform well on hardware configurations larger than those that currently exist.

Prototyping has played an important role in evaluating several aspects of the VCE. It has been used to validate simulation results and identify issues that must be addressed when a production VCE system is developed. However, the simulation model provides opportunities to observe how system behavior is impacted by subtle changes in parameters. These kinds of observations are difficult to perform using prototype systems alone.

6.3.2 Basic Elements of the Simulation Model

The VCE simulation software is a discrete event simulator written in an object oriented style in C++ [97]. Simulation software is extremely amenable to object oriented design methodologies. The motivation for the programming language "C with Classes", which was a direct precursor to C++, was to facilitate development of efficient distributed system simulation programs [98]. A simulation program describes (models) objects in the real world and how they interact with one another. It is natural to write a single module to represent each object being simulated and to encapsulate the behavior of the object within the module. This encapsulation maximizes the flexibility and modifiability of the simulator. The `processor` class, which

is described below, models a CPU time-shared among many tasks using round robin scheduling. The object oriented structure of the simulator makes it easy to change features like this, so changing the model to use FCFS scheduling would only require a small change to a single module. This modifiablity will ease future enhancement of the VCE by allowing new hardware elements and software approaches to be quickly reflected in the simulator.

The simulation model uses a C language [62] library called **smpl** [75]. The library provides functions to manage an event queue. An event queue is a linked list. Each item on the list represents an event which is going to occur later in the simulation. Each list item contains an indicator of the type of event and the time at which it is scheduled to occur. **smpl** functions are provided to add an event to the list (schedule it), remove an event from the list (cancel it), and fetch the event which is to occur next (cause it). The **smpl** library also provides functions to generate random variables from various distributions, collect statistics about simulation objects and format reports.

The simulator employs a class hierarchy. The simplest classes are *tasks* and *processors*. Processors represent a single CPU. Tasks are sequential jobs which run on a single processor. More complex classes are achieved by aggregating these classes. Supercomputers and workstation clusters contain arrays of processors while parallel applications are modeled using arrays of tasks and VCE applications are constructed as arrays of parallel applications. Details of all these constructs are discussed in this section.

Besides being useful as a performance evaluation tool, the simulator also has been helpful in formulating the overall VCE design. It has served as a means to document the assumptions of the design and to experiment with and debug algorithms before they are implemented in prototypes.

Processor Modeling Issues

The most fundamental constructs in the VCE simulation are processors and tasks. Every supercomputer and cluster in a simulation contains processors which can execute

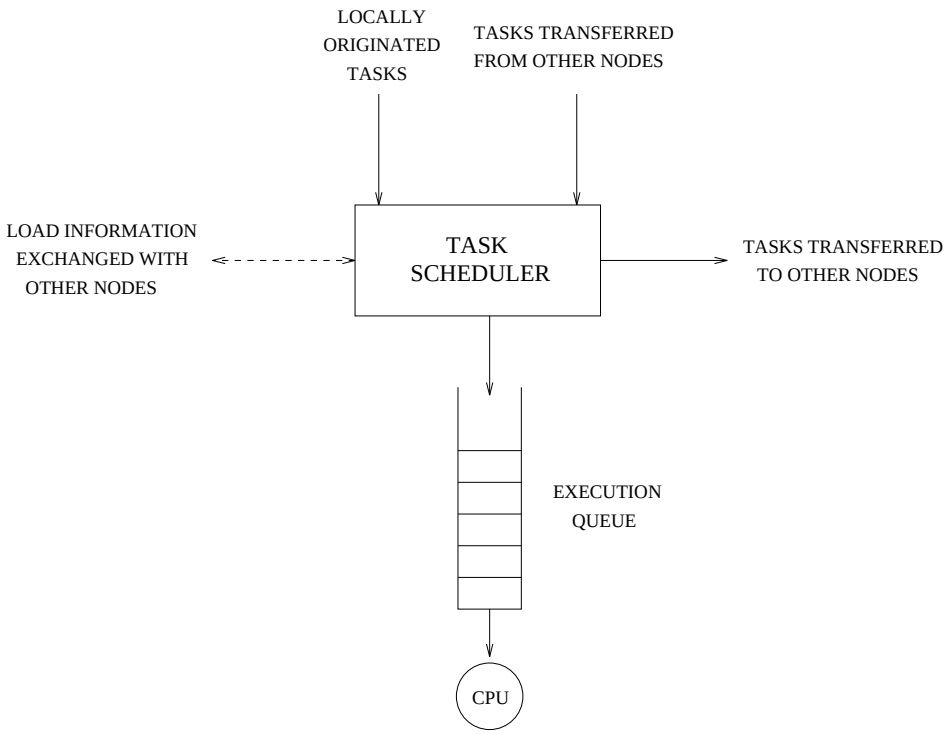


Figure 44: A Typical Node Model

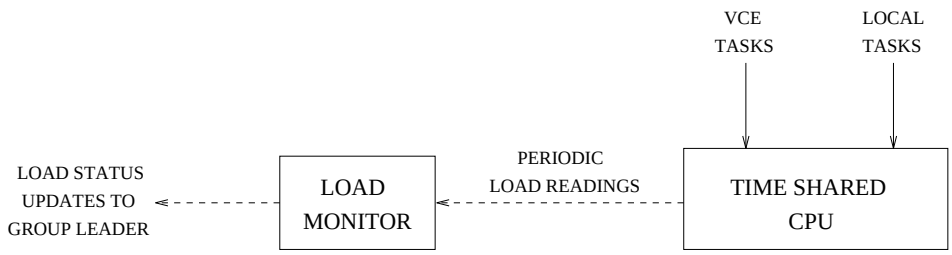


Figure 45: The VCE Node Model

tasks. As was noted earlier, most distributed system simulators used to study load balancing algorithms assume a first-come first-served processor scheduling model [3, 4, 39, 50, 79, 84, 95, 110]. Figure 44 shows how processors are typically modeled in such studies. The *task scheduler* in the figure is a software process which implements the processor load measurement technique, information exchange policy, transfer policy and location policy (see Section 5.1.2) of the load balancing algorithm being considered. Tasks arrive at the node due to actions of a local user or as a result of transfers from heavily loaded nodes. When a task arrives the task scheduler performs the load balancing algorithm and either adds the task to the queue of the local processor or transfers it to another node.

The node model of Figure 44 is not used in the VCE model. The FCFS protocol implied by the figure is not appropriate. If the processor being considered is a high performance workstation or an element of a MIMD parallel computer, it is very likely that processor resources are shared among all locally executing tasks on a round robin basis [100]. The choice of whether to model processors as time shared resources or FCFS queues has a significant impact on the simulation results [34]. In FCFS models, once a task is scheduled on a node its completion time is a function of its running time and the running times of the tasks queued ahead of it. It is unaffected by subsequent load changes. On a time shared processor each time a task is accepted for execution the performance of all the tasks already executing on that node is degraded.

While most load balancing simulations adopt a FCFS node model, several studies have considered a round robin local scheduling protocol [34, 108, 110]. The node

model used in the VCE simulation is shown in Figure 45. The *Load Monitor*, in a real VCE implementation, is a process within the scheduling/dispatching daemon which implements the processor load measurement algorithm described in Section 5.1.2. The processor load measurement policy determines whether the node is eligible to accept VCE tasks. When the load monitor detects a significant and sustained change in the local load, a message indicating the new load level is sent to the group leader. Tasks are generated either locally or are submitted by remote VCE execution processes when the node is lightly loaded.

Processor and Task Simulation Algorithm

In the VCE simulation model each processor $\mathbf{P^n}$ and task $\mathbf{T^m}$ is assigned a unique integer identifier. The static characteristics of processors and tasks are: $\mathbf{P^n_{sp}}$, the speed of processor n, and $\mathbf{T^m_{sz}}$, the size of task m. The latter quantity is defined as the time that would be required to complete task T^m as the only task running on a processor of speed 1.0.

The following dynamic quantities can be measured at any time during the simulation:

CurrentTime The current simulation time.

$\mathbf{P^n_{ld}}$ The current load on processor P^n (that is, the number of tasks it is running).

$\mathbf{P^n_{cm}(T^m)}$ The proportion of task T^m which has been completed on processor P^n.

$\mathbf{P^n_{ex}(T^m)}$ The expected time of completion of task T^m on processor P^n.

The time required to execute a task on a specific processor is called the *lifespan* of the task. The lifespan increases if the load on the processor is high and is influenced by the speed of the processor. More formally, the lifespan of task T^m on processor P^n, $P^n_{ls}(T^m)$, is computed as:

$$P^n_{ls}(T^m) = \frac{T^m_{sz}\, P^n_{ld}}{P^n_{sp}}. \qquad (1)$$

Three types of events can occur on a processor: a task can begin execution, a task can complete, or a task can be cancelled. Each of these events results in an ordinal change in the load on the processor. When any of these events occurs $CurrentTime$ and P^n_{ld} are updated.

When a task begins on a processor its load is incremented and the task's initial expected completion time is computed as:

$$P^n_{ex}(T^m) = CurrentTime + P^n_{ls}(T^m). \tag{2}$$

An *interval* is the time between consecutive events on a processor. The duration of intervals is variable and may be zero. When an event occurs the length of the terminating interval is computed and stored in the variable **int**. Before the load of the processor is incremented or decremented the amount of progress made by each task is calculated using the function:

$$P^n_{cm}(T^m) = P^n_{cm}(T^m)^{old} + \frac{int}{P^n_{ls}(T^m)}. \tag{3}$$

That is, The duration of the interval is divided by the total time needed to complete the task to determine what portion of the task was completed during the interval. This is added to the portion of the task which was completed in prior intervals to obtain the total amount of progress made on the task so far.

Once the processor load is updated and Equation 2 is performed for newly arriving tasks, a new expected completion times must be computed for any task which was executing on the processor in the previous interval and has not yet completed. The updated load value, P^n_{ld} is used to calculate a new lifespan, $P^n_{ls}(T^m)$, for each task using Equation 1. Since $P^n_{ls}(T^m)$ represents the total time needed to execute the task and $P^n_{cm}(T^m)$ represents the fraction of that time which has already elapsed, the function which gives the new expected completion time for each task is:

$$P^n_{ex}(T^m) = CurrentTime + (1 - P^n_{cm}(T^m)) \cdot P^n_{ls}(T^m) \tag{4}$$

Figure 46 shows the complete processor simulation algorithm. The algorithm is executed each time an event occurs on a processor (a task arrival, completion or

for each running task
  compute proportion completed, $P_{cm}^{n}(T^{m})$, using Equations 1 and 3

update the processor load P_{ld}^{n}

for each arriving task
  compute the initial expected completion time, $P_{ex}^{n}(T^{m})$,
  using Equations 1 and 2

for each running task
  compute updated expected completion time, $P_{ex}^{n}(T^{m})$,
  using equations 1 and 4

Figure 46: Processor Event Simulation Algorithm

Table 7: Example Simulated Tasks

Task	Size (T^m_{sz})	Arrival Time
1	20	0
2	5	10
3	10	14

Table 8: Processor Simulation Example

		Updated Simulation Variables						
Time	Event	P^1_{ld}	$P^1_{cm}(T^1)$	$P^1_{ex}(T^1)$	$P^1_{cm}(T^2)$	$P^1_{ex}(T^2)$	$P^1_{cm}(T^3)$	$P^1_{ex}(T^3)$
0	T^1 arrives	1	0	20				
10	T^2 arrives	2	.5	30	0	20		
14	T^3 arrives	3	.6	38	.4	23	0	34
23	T^2 completes	2	.75	33	1	23	.3	37
33	T^1 completes	1	1	33	1	23	.8	35
35	T^3 completes	0	1	33	1	23	1	35

cancellation). The algorithm produces a new expected completion time for each task resident on the processor after the event. Once the expected completion time is known for all running tasks smpl routines are called to schedule the next completion event.

Processor Simulation Example

To clarify these concepts the three tasks in Table 7 will be used in a small example of how the behavior of an individual processor is modeled in the VCE simulation. The table shows arrival times and sizes of the tasks. In an actual simulation these values are determined using pseudorandom variables (Section 6.3.5).

Table 8 shows the events and calculations which would occur when execution of the tasks is simulated on processor P^1 whose processing speed, which is a user specified simulation parameter, is 1.0 ($P^1_{sp} = 1$). When the first task arrives at time 0 it is

beginning execution so no part of the task has been executed ($P^1_{cm}(T^1) = 0$). Since it is the only task on the processor and $CurrentTime = 0$ Equations 1 and 2 reduce to $P^1_{ex}(T^1) = T^1_{sz} = 20$. When the second task arrives ten time units later, Equation 3 shows that the first task is half done ($P^1_{cm}(T^1) = .5$). The arrival of T^2 causes the load on the processor to go to two ($P^1_{ld} = 2$). Now the processor is being shared by two tasks, so the lifespan of both is doubled (Equation 1). The initial expected completion time of T^2, $P^1_{ex}(T^2)$, is set to 20 using Equation 2 and the expected completion time of T^1, $P^1_{ex}(T^1)$, is updated to 30 by Equation 4. The simulation continues in this fashion with the portion of each task that has been completed ($P^n_{cm}(T^m)$) and its expected completion time ($P^n_{ex}(T^m)$) being recomputed each time a task arrives or completes. When a task completes its expected completion time ($P^n_{ex}(T^m)$) becomes an actual completion time.

Simulating Low Priority Tasks

In some cases the VCE design requires VCE tasks to be scheduled at low priority. Therefore, low priority tasks must be accommodated by the simulation model. This is done by maintaining a separate queue of low priority and normal priority tasks. When a simulated processor has both normal and low priority tasks running the low priority tasks are disregarded. That is, they are not figured in to the processor load and they do not make progress towards completion. When a processor has only low priority tasks, they are processed using the time sharing algorithms described above. While most operating systems attempt to give some resources to low priority tasks [100], very low priority CPU-bound tasks competing against higher priority CPU-bound tasks in Unix based systems will make minimal progress [8].

6.3.3 Parallel Applications

The processor and task models described above can be used in a straightforward way to model a MIMD parallel computer. For the purposes of the simulation study, a parallel machine is considered to be a collection of processors with identical speed

values. A primary use of parallel computers is to execute parallel applications.

A parallel application is one that simultaneously assigns tasks to several nodes in the supercomputer. These tasks exchange data with each other periodically. The parallel applications in the simulation study are simplified somewhat to avoid the complexities of excessive detail which would not appreciably improve the accuracy of the results.

Each parallel application has a unique integer identifier and a set of nodes which will perform its computations. That is:

$$\mathbf{A^l_{nodes}} = \{P^h..P^i\} \tag{5}$$

where the sequence $h..i$ represents a sequence of unique though not necessarily consecutive integers. Each processor in the set must belong to the same parallel computer. No processor belongs to more than one parallel computer.

For the purposes of the model, we characterize a parallel application as dispatching some computation on each node. After these computations complete the nodes exchange data and begin a new computation. This period of computation followed by a period of communication constitutes one *phase* of the application. If a node completes its computation for a given phase before the other nodes, then the faster node becomes idle (with respect to this application) until the slower nodes complete the phase. Since parallel application designers strive to equalize the amount of work done by each node in each phase, the model treats each phase as a single task. For an application with k phases this gives the ordered sequence of tasks:

$$\mathbf{A^l_{tasks}} = (T^j, T^{j+1}, ..T^{j+k}). \tag{6}$$

Where the size of each task is equal to the combined computation and communication time of the corresponding phase of the application. That is, T^j_{sz} equals the total computation and communication time of the first phase of application A^l, T^{j+1}_{sz} equals the total time of the second phase, and so on.

Execution of a parallel application is simulated by starting the first task in A^l_{tasks} on each processor in A^l_{nodes}. When each instance of the first task completes, the

second task is dispatched, and so on. Various metrics can be produced relating to the execution of a given parallel application A^l. Two that are of particular interest are the total time to completion of the application and the the average time that processors in A^l_{nodes} where idle with respect to A^l.

6.3.4 Workstation Clusters

Workstation clusters are modeled largely as groups of independent machines. In the VCE the only restriction on workstation clusters is that they be object code compatible. The assumption made for supercomputers that all nodes have the same processing speed is not reasonable for workstation clusters.

6.3.5 Modeling Processor Workload

The main objective of the VCE simulation model is to study the performance of redundantly scheduled VCE-tasks and their impact on conventionally generated computing tasks. Therefore, generating a realistic background workload for simulated machines is important to the results. For workstations, making the common assumption that each machine's workload is characterized by a exponentially distributed interarrival time and an exponentially distributed service time is reasonable. However, this does not imply that each machines conforms to the M/M/1 queue model because Unix workstations are timeshared rather than being single server FIFO queues [8, 70]. Therefore, the background workload generator for a workstation generates task arrival times and task sizes from exponential distributions and the tasks are serviced as described in Section 6.3.2.

Modeling the workload of a parallel supercomputer is even more complex. Supercomputing tasks have more parameters than the usual arrival time and service requirement. As discussed in Section 6.3.3, the characteristics of a parallel application include the number of nodes required, the number of phases and the length of each phase. Determining reasonable estimates for these parameters is difficult. Several thorough studies have been done regarding usage patterns of workstations [80, 81].

Similar studies of supercomputers have not been done. One reason for this may be the relative small number of machines deployed. While it is possible for a researcher to select a sample of a few dozen workstations and monitor their usage over several months, this approach will not work for parallel computing systems because there are rarely more than a few systems at any one site.

For the VCE simulation study the following approach was taken for generating the background workload: Application interarrival times, number of phases and amount of processing per phase were all selected from exponential distributions. Observation has shown that parallel application programmers often favor using a number of nodes which is a power of two. Therefore, the number of nodes for an application is selected by choosing a number from a uniformly distributed discrete random distribution and raising two to that power.

Sets of load generator parameters were developed for both parallel computers and workstation clusters that resulted in a high, medium or low load on the system. The load generator for workstation clusters can also be used to achieve a mixed workload which is typical of many workstation environments.

6.3.6 Modeling VCE Components

In addition to the structures discussed in the previous sections the VCE simulation effort also has to model some system elements that are unique to the Virtual Computing Environment. Specifically, VCE group leaders and execution processes must be modeled. To support these functions a unique feature was incorporated into the processor model. The Unix rstat() system call is simulated by the processor model. The VCE load balancing functions, like many load balancing functions, consider the length of a processors ready queue as an indicator of its load. On real Unix systems, programs running at the user level do not have direct access to this information. Instead, the VCE uses the rstat() system function. rstat() returns, among other things, an indicator of the average length of the processor ready queue over the preceding 60 seconds. The VCE load monitoring function issues rstat() every six seconds and

extrapolates consecutive readings to estimate the ready queue length over shorter periods. So that the same load monitoring algorithm could be used in the simulator as is used in the VCE prototypes, the simulator,s processor class includes an rstat() method which returns the ready queue length over averaged over the past 60 simulated seconds.

Simulated group leaders execute code based very closely on that deployed in the experimental prototype system group leaders. Each group leader monitors one workstation cluster or supercomputer and maintains a list of the nodes in the group which have been lightly loaded for an extended of period of time.

The execution process used in the VCE simulator also mirrors that used in the actual system. It includes a parser for application description framework files described in Section 4.2. The syntax of the application description frameworked used by the simulator is enhanced so the size of each executable file can be encoded in the application description. This size information can be obtained experimentally using the VCE prototype system. This insures that the characteristics of the VCE applications studied with the simulator matches those of actual applications as closely as possible.

The simulated execution process allows for experimentation with many system parameters. VCE tasks can be dispatched at normal or low priority and the redundancy levels can be varied.

6.3.7 Instrumentation

The model provides ample opportunity for collecting a variety of performance and utilization data. Each task (Section 6.3.2) has an indicator of its size. The task's arrival time, completion time and total running time are recorded during the simulation. The difference between a task's size and its running time constitute its *delay*. As discussed in Section 6.3.3, a parallel application is divided into several phases where each phase contains multiple tasks which are executed in parallel. The delay of an application phase is given by the maximum delay of any task, not the sum of the task delays. The delay of an application is computed as the sum of the delays

of each phase. When a VCE application is simulated, the delays it causes for other applications can be precisely measured. This statistic is an important measure of how non-VCE users are impacted by VCE scheduling policies.

In addition to gauging application performance, the model also provides hardware utilization data. Each processor is modeled using a smpl *facility* [75]. Software in smpl automatically collects a variety of information about each facility (processor) including the average number of tasks using it and the amount of time it was idle.

The model tracks the amount of processing performed by each cancelled task. This statistic is important in judging the effectiveness of VCE scheduling and is difficult to measure on prototype systems. On hardware systems when a task is cancelled before it produces output there is no indication of much computation it had performed. The only way to measure this on a prototype is through the use of software monitors which can skew the results of the experiments [57]. On the simulator the portion of each task which is already completed is recorded for each task running on a processor any time an event occurs which changes the number of tasks on the processor. Recording that data within cancelled tasks is quite natural. This data is useful in measuring the cost of redundant scheduling practices.

6.3.8 Simulator Results

A wide variety of experiments are possible using the simulator. Two will be discussed in this section.

Measuring the Effect of Various Redundancy Levels

One series of experiments was conducted using a small simulated system in order to further confirm that a redundancy level of more than three yields only nominal performance improvements. These experiments also demonstrated the importance of the VCE's load measurement and task placement techniques.

The simulated configuration for these experiments was kept simple. The hardware simulated was a single 32 node parallel computer and a cluster of 64 homogeneous

Table 9: Application Performance at Low Load Levels With Selective Placement

REDUNDANCY LEVEL	RUNNING TIME	CPU SECONDS USED
1	7839	20000
2	7341	36521
3	6605	50963
4	6568	64435

workstations. The workstations were simulated as being 25% slower than the nodes of the parallel supercomputer. A small non-VCE workload was placed on the parallel computer. The workstation cluster was established as containing an approximately equal mix of lightly, moderately and heavily loaded machines.

The application simulated was the linear equation solver whose structure was shown in Figure 26 in Chapter 4. The application has five tasks. For the purpose of the simulation it was assumed that each task required four nodes of the parallel computer or four workstations and required 1000 CPU seconds of processing on each node or workstation. In order to allow comparisons between the workstations and parallel computer nodes, but still account for the workstation's lower speed, each "CPU second" on a workstation is characterized as needing 1.25 wall clock seconds to elapse. This means the application, if dispatched non-redundantly, would require 20000 CPU seconds to run on either platform. One of the application tasks can run in parallel with another portion of the application, so if the application had exclusive use of eight nodes of the parallel computer it would complete in 4000 seconds of wall clock time. It would require 5000 seconds of wall clock time to complete on eight otherwise idle workstations.

Tables 9 and 10 show average results of five simulation runs for each of eight experiments. The results in Table 9 reflect the simulated application running on the simulated hardware described above using the scheduling methods described in

Table 10: Application Performance at Low Load Levels Without Selective Placement

REDUNDANCY LEVEL	RUNNING TIME	CPU SECONDS USED
1	13859	20000
2	10817	32391
3	9427	44590
4	9182	57568

Section 5.1. Table 10 represents the same scenario, except that the VCE load measurement routines were not used. Task instances were placed on randomly selected hosts.

These results show that both selective initial task placement and redundant scheduling improved performance significantly, and using both produced better results that using either one alone. The first entry in each table (REDUNDANCY LEVEL 1) measures application performance without redundant scheduling. The difference between these two numbers represents the performance degradations that results from placing application tasks without regard for the load on each computing node when the tasks are initiated (Table 10). Simply placing tasks on nodes that have excess capacity at the time the tasks are initiated improves performance by 43% (Table 9). The size of the performance improvement is reasonable because one third of the workstations in the simulation are heavily loaded. When selective placement methods are used it is unlikely that any of these machines will be chosen to host any portion of the VCE application. Without redundancy or selective placement a small number of heavily loaded machines may degrade application performance significantly.

The other entries in Table 10 show that redundant scheduling does somewhat (but not entirely) offset the lack of a selective task placement policy. Three way redundancy reduces average application run time by 32%, which is still not as fast as running with selective placement and without redundancy.

Even with selective scheduling policies and an environment with high levels of

excess capacity, redundant scheduling can further improve performance. As seen in Table 9, two- and three-way redundancy produces performance gains of 6% and 16%.

These simulation results make it possible to measure the extent to which computing cycles are wasted doing redundant processing. The data shows that when selective scheduling is used, more CPU seconds are wasted than when it is not. This is because, without selective placement, some task instances are placed on overloaded hosts where they are canceled before they consume much CPU time. With selective placement, it is more likely that multiple instances of a task will be placed on lightly loaded machines and the slowest instance will near completion before the fastest instance completes.

Measuring the Effect of Low Priority Task Scheduling

The VCE simulator has also been used to study the feasibility of scheduling VCE tasks at low priority in order to minimize interference with non-VCE users. The advantages of low priority scheduling are considerable. It insures that only excess computing resources are consumed by VCE applications. When a low priority VCE task is executing on a machine, it will make only nominal progress if regular priority non-VCE jobs are present and competing for resources [8]. When other work is not active on a node, the low priority VCE task can benefit from computing cycles that would otherwise go unused. Of course, low priority tasks are always in danger of starvation. The purpose of the simulation experiments described below was to determine to what extent the VCE's redundant scheduling policy compensates for the performance impact of low priority dispatching.

These experiments used the same simulated hardware configuration as described earlier. That is, a 64 node workstation cluster and a parallel computer with 32 faster nodes were used. The linear equation solver was used again as a test application. The load conditions were changed somewhat from the experiments described above. For these experiments the load on the parallel computer was set to *moderate* rather than *low*. The additional load was used because the question under consideration relates to whether redundant scheduling can avoid delays due to task starvation. Starvation

Table 11: Application Performance at Moderate Load Levels With Selective Placement

REDUNDANCY LEVEL	RUNNING TIME	CPU SECONDS USED
1	9595	20000
2	8334	36112
3	6939	50889
4	6853	61121

Table 12: Application Performance at Moderate Load Levels With Low Priority Workstation Dispatching

REDUNDANCY LEVEL	RUNNING TIME	CPU SECONDS USED
1	20881	20000
2	11550	33711
3	10068	45114
4	8002	51704

is unlikely to occur with low load levels. The workstation cluster was again modeled as containing a roughly equal mix of lightly, moderately and heavily loaded machines. Table 11 gives observations of application performance at these higher load levels with all VCE tasks dispatched at normal priority. The results are comparable to those in Table 9 except for the generally poorer performance due to the higher non-VCE workload.

The first set of experiments in this area involved modifying the simulated VCE execution process to dispatch VCE tasks on workstations at low priority. This is expected to be a common method of scheduling VCE work because workstation owners expect predictable performance from their machines. Using low priority for VCE tasks is less likely to cause noticeable workstation performance degradations. Parallel computers are commonly a shared resource, so non-VCE users will be less sensitive

Table 13: Application Performance at Moderate Load Levels With Low Priority Dispatching on All Machines

REDUNDANCY LEVEL	RUNNING TIME	CPU SECONDS USED
1	25003	20000
2	21130	32509
3	12826	45813
4	11601	54089

to competition from VCE applications. Table 12 Shows the result of rerunning the scenario of Table 11 with low priority dispatching of VCE workstation tasks. It is apparent that reducing the dispatching priority of VCE tasks on the workstations in this configuration significantly reduces application performance, particularly when redundant scheduling is not used. It can also be seen that redundant scheduling does mitigate much of the performance loss due to lower scheduling priority. While the running time of 20,881 seconds for non-redundant scheduling with low priority scheduling (Table 12) more than twice as long as the 9,595 seconds needed with normal priority (Table 11), with three way redundancy the performance loss is only 45% (10,068 seconds vs. 6,939 seconds).

To further explore the impact of low priority scheduling, another set of simulation runs was performed with another change to the VCE execution process. Table 13 shows the impact of dispatching VCE tasks at low priority on the configuration's simulated parallel computer as well as the workstation cluster. The advantage of such a technique is that virtually all of the resources used to execute the application will come from excess CPU cycles and have negligible effect on conventionally executed (non-VCE) tasks. Of course, using very low priority for all VCE tasks does further reduce performance. But again, redundant execution lowers the impact markedly.

To emphasize the conclusions that can be drawn from the data in Tables 11, 12

and 13 it is instructive to focus on the **REDUNDANCY LEVEL 3** entry in each table. If the application were run on the simulated supercomputer in this configuration with no competing load, it would require 4,000 seconds to complete. Using three-way redundant scheduling and normal dispatching priority (Table 11) the application completes in 6,939 seconds despite a considerable transient load on the hardware. This performance is 28% better than the 9,595 seconds observed without redundant scheduling. Modifying the dispatching to use low priority scheduling of VCE tasks on workstations, but maintaining three-way redundancy, increases running time to 10,068 seconds, a tolerable level compared to the 20,881 seconds achieved without redundancy. Finally, using low priority scheduling for all VCE tasks and three-way redundancy increases application running time to 12,826 seconds, Low priority scheduling insures nominal interference with normal priority (non-VCE) work [8]. This shows that the VCE scheduling protocols can provide tractable performance with no interference to conventional users. Using low priority scheduling without redundant execution doubles application running time and is unlikely to provide acceptable performance. These experiments demonstrate the types of performance impacts that result from low priority scheduling. System adminstrators and planners can use this data to determine which scheduling policies are to be used in their installations.

6.3.9 Use of the Simulator to Design VCE Configurations

The experiments described in the previous section demonstrate the value of the VCE simulation model. Using it, it was possible to evaluate how changes in redundancy levels and dispatching policies would impact application performance under different load conditions. The experiments described above support the value of key VCE design features including redundant scheduling and simplified load measurement. Simulation experiments make it possible to precisely quantify how a VCE parameter like task dispatching priority can be used to improve VCE application performance or reduce interference with non-VCE workloads. The simulator can be used by system administrators to determine how VCE scheduling policies can best be customized to

satisfy the objectives of their installations.

6.4 Summary

This chapter described the tools which have been implemented to study the characteristics of the VCE execution suite. The prototype systems have been used to build preliminary implementations of key elements of the VCE run time environment (scheduling/dispatching daemons, group leaders, execution processes and application description framework parsers). They have also been used to construct, execute and gauge the performance of several VCE applications in real world environments. Experimentation with the prototypes have demonstrated the implementability of the execution suite and the effectiveness of redundant scheduling. The prototypes can be enhanced in the future to include other aspects of the execution suite such as the communication functions described in Section 5.2. The prototypes will serve as a foundation for future execution suite systems.

The VCE simulation model provides a vehicle for determining how the VCE execution suite will perform in, and impact, possible host configurations. By precisely and efficiently modeling key elements of the run time environment, the simulator will enable planners and system administrators to anticipate what levels of performance they can expect from the VCE execution suite. It can also be used to determine how system parameters can be set, and system components customized, to deliver the best application performance without imposing unacceptable impacts to non-VCE users.

Chapter 7

Conclusions and Future Work

The main goal of the VCE is to provide a comprehensive set of tools to ease the development and execution of high performance applications that can harness the computing and storage capability of a wide range of heterogeneous high performance computers. This chapter will discuss the main contributions of this research. A plan of additional research and experimentation that will enable the VCE to evolve into a robust and widely deployed technology will be described.

7.1 Conclusions

The key points established in this dissertation are:

- New technologies to manage applications that exploit multiple parallel computers may be a more successful approach to high performance computing than developing more complex hardware architectures.
- Existing distributed environments are not adequate to meet the needs of Grand Challenge applications.
- A software engineering environment that separates programming-in-the-small from programming-in-the-large [35] will enable scientists to construct high performance applications without becoming parallel programming experts.

- The VCE application description framework establishes an effective interface between software engineering tools and a powerful run time environment.
- VCE research has yielded a design for a suite of scheduling, dispatching and communication tools that can be used to manage the execution of large scale distributed applications.
- Prototype VCE systems and simulation tools have been implemented. These systems have validated key VCE design points and will serve as a basis for future systems and research efforts.

A fully implemented Virtual Computing Environment will include many emerging technologies. The modular design of the software development suite will allow distributed object [17] and World Wide Web [45] based software development tools to be incorporated into the VCE. The communication daemons described in Section 5.2 will use advanced communication protocols to facilitate efficient communication among application components. This dissertation lays out a blueprint of the structure of the VCE. While the specific tools employed will inevitably change over time, the relationships among the VCE components and the capabilities required of each tool are established here.

7.2 Future Work

There are a wealth of areas of VCE research which may hold considerable promise, not only to increase the robustness and deployability of the VCE, but also to contribute to a better understanding of many issues in high performance distributed computing. A few of these areas are described below.

7.2.1 Application Description Framework Enhancements

The application description framework syntax described in Chapter 4 is adequate for documenting the structure of a wide variety of multistage applications. However,

task: matrix_preprocessor
executables: SUN: matrix_preprocessor.sun
SP2: matrix_preprocessor.sp2
ALPHA: matrix_preprocessor.alpha
nodes: SUN: 1
SP2: 1
ALPHA: 1
input: input_matrix.file
output: output_matrix.matrix
number_of_rows.integer
hints: replicateable
duration: short

Figure 47: Matrix Preprocessor Task Description

there are some simple enhancements that could be made which would enhance the utility of the framework considerably. For instance, the syntax given in Chapter 4 is completely static. The tools in the software development suite are responsible for establishing what tasks are to be executed in what order and how data is to be routed among them. The execution suite can choose which machines should host the tasks and select the communication protocols to support them (Chapter 5), but the structure of the application itself cannot be modified at run time.

One way to enhance the flexibility and expressiveness of the application description framework would be to allow specifications of some application characteristics to be expressed in terms of task output variables. For instance, the task description in Figure 47 is a specification for a matrix preprocessor task. The task runs on a single workstation and reads a `matrix` data file from disk. It generates a `matrix` object and an integer indicating the number of rows in the matrix. This task could be used as the initial task in a variety of dynamic matrix processing applications. For example, Figure 48 shows how the linear equation solver application (see Figure 26 in Chapter 4) could be modified to use the preprocessor task. In the dynamic version of the application, the number of nodes used by each executable task implementation, which is specified in the `nodes` field of each `stage`, is expressed as a function of the `num_rows` variable produced by the preprocessor. Different functions are used for different machines because of the differing granularity requirements of the various architectures. Support for dynamic application descriptions like the one in Figure 48

Application Description

application: solver

stage: preprocess
task: matrix_preprocessor
input: matrix_file
output: initial_matrix
num_rows

stage: LU_decompose
task: LU_decomposition
nodes: SUN: num_rows / 400
SP2: num_rows / 200
ALPHA: num_rows / 200
input: A_matrix_file
output: lower, upper
synchronization: after preprocess completes

stage: inverse_matrix_1
task: inverse_matrix
nodes: SUN: num_rows / 400
SP2: num_rows / 200
ALPHA: num_rows / 200
input: lower
output: lower_inverse
synchronization: after LU_decompse completes

stage: inverse_matrix_2
task: inverse_matrix
nodes: SUN: num_rows / 400
SP2: num_rows / 200
ALPHA: num_rows / 200
input: upper
output: upper_inverse
synchronization: after LU_decompse completes

stage: first_matrix_multiply
task: multiply_matrix_by_vcetor_file
nodes: SUN: num_rows / 400
SP2: num_rows / 200
ALPHA: num_rows / 200
input: lower_inverse, B_vector_file
output: temp_vector
synchronization: after inverse_matrix_1 completes

stage: second_matrix_multiply
task: multiply_matrix_by_vector_giving_file
nodes: SUN: num_rows / 400
SP2: num_rows / 200
ALPHA: num_rows / 200
input: upper_inverse, temp_vector
output: X_vector_file
synchronization: after first_matrix_multiply completes
after inverse_matrix_2 completes

Application Graph

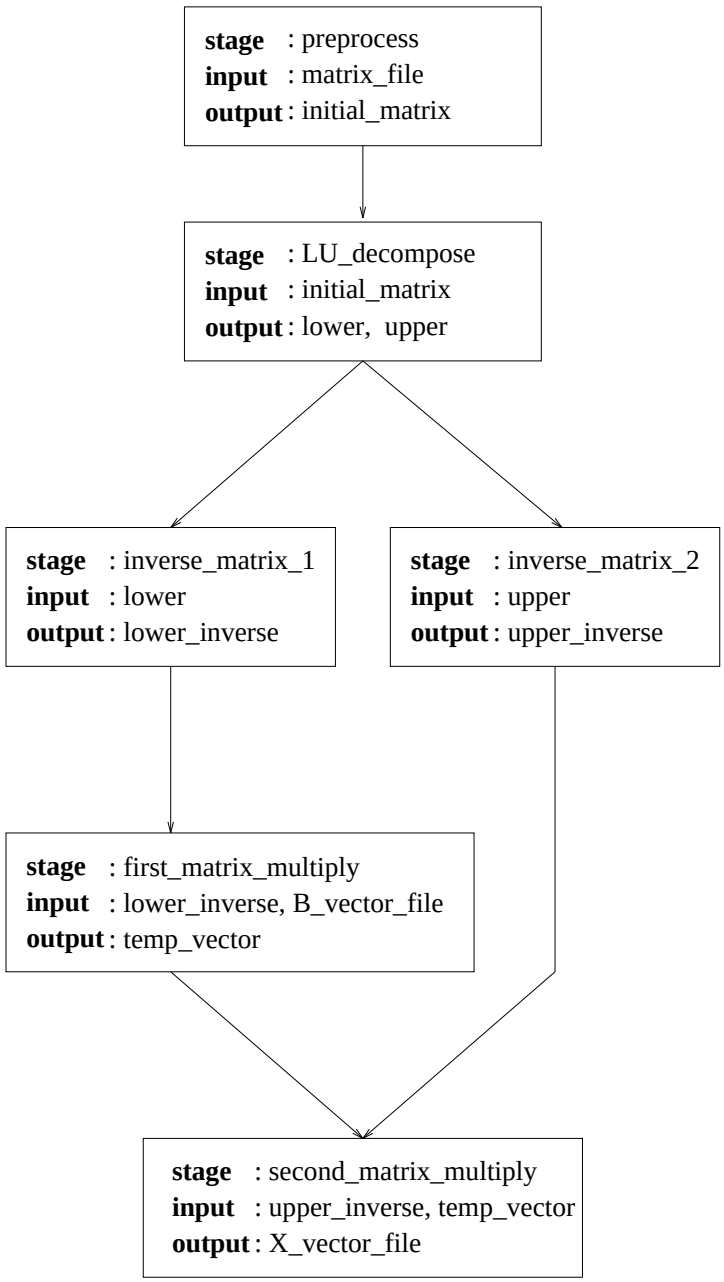


Figure 48: A Dynamic Application Description

would enable the execution suite to tailor the application based the characteristics of the data processed in a particular run.

The application description framework could be made even more robust by adding a few simple directives like **perform_when** or **perform_until** to allow conditional or iterative execution of application stages [14]. It would not be appropriate to expand the application description framework into a full programming language. That level of functionality would increase the complexity of the execution suite. However, addition of a small number of conditional execution directives could make the syntax much more powerful without significantly complicating the execution environment.

7.2.2 Implementation of a VCE Communication System

The design of the VCE communication system is completely specified in Section 5.2. Since the working prototypes of the VCE scheduling/dispatching system are already complete, addition of a communication system would produce a run time environment with enough functionality to run a large repertoire of real world applications. Implementing the communication daemons described in Chapter 5 will provide many opportunities for experimenting with the communication requirements of large scale applications. It will also allow the scheduling and dispatching algorithms to be enhanced to consider communication facilities in task placement decisions. For instance, hosts attached to hich performance ATM networks would be preferred over those attached to less capable communication facilities. The ability to make run time scheduling decisions based on communication facilities as well as load conditions will further increase the value of the execution suite.

7.2.3 Refinement of the VCE User Interface

The VCE application description is human readable, simple and intuitive. Users will have little difficulty developing task and application descriptions to test prototype or production VCE execution suites. This, along with the VCE's modular design, will permit development of the VCE software development and execution suites to

proceed in parallel.

Research into software engineering tools to develop scientific applications on high performance parallel computing platforms is in its infancy [7] and can be expected to be fruitful. Further development of the VCE software development suite will be an important contribution in this arena.

As was discussed in Chapter 3, the most straight forward way to construct a prototype VCE software development suite is to design software to translate output of the AVS [2] modular application builder into the syntax of the VCE application description framework. This would provide a highly functional visual VCE application development environment. Later addition of object oriented [1, 17] or World Wide Web oriented [45] tools would allow users to chose the interface which best suits their preferences and the characteristics of their application.

7.2.4 Simulation Model Enhancements

Simulation of distributed systems is an important area of computer engineering [57]. Simulation models can be used to perform controlled experiments and explore system options that are difficult to construct experimentally and impossible to model analytically. The VCE simulation model described in Section 6.3 includes many unique features. For instance, The simplified processor model efficiently captures the behavior of a time-shared CPU. An object oriented approach is used so that a single processor model is aggregated in simulated parallel computers, workstations, and workstation clusters.

While the simulation model was developed to study the characteristics of the VCE's redundant scheduling policies, it is possible that it may have wider applicability. Considerable research would be needed to compare the models behavior to a wide variety of experimental systems before it could be advocated as a general purpose tool. However, such an effort would be justified if it showed that the model provides a superior way to study scheduling policies for parallel computers attached to wide area networks.

It is possible that the simulator could be structured as a parallel application. Its object oriented structure suggests that each simulated parallel computer or workstation cluster could be placed in a separate processor of a real parallel computer. In fact, VCE simulations could be used as VCE applications with the simulator studying hypothetical VCE configurations while its own execution is used to benchmark a prototype system.

Bibliography

[1] Adler, R. M., *Emerging Standards for Component Software*, Computer, Vol. 28, No. 3, pp. 68-76, March 1995.

[2] Advanced Visual Systems Inc., **AVS Developer's Guide (Release 4.0)**, Part No. 320-0013-02, Revision B, Advanced Visual Systems Inc., Waltham Mass., May 1992.

[3] Ahmad, I., Ghagoor, A., and Mehrotra, K., *A Decentralized Task Scheduling Algorithm and its Performance Modeling for Computer Networks*, Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing, pp. 314-321, December 1991.

[4] Ahmad, I., Ghagoor, A., Mehrotra, K., Mohan, C. K., and Ranka, S., *Performance Modeling of Load Balancing Models Using Neural Networks*, Concurrency: Practice and Experience, Vol. 6, No. 5, pp. 393-409, August 1994.

[5] Allia, M., Arlat, J., Fabre, J.-C., Kanoun, K., and Powell, D., *Dependability Evaluation of the Saturation Technique*, Proceedings of the 5th Annual European Computer Conference (CompEuro '91), pp. 437-441, May 1991.

[6] Anderson, T. E., Culler, D. E., and Patterson, D. A., *A Case for NOW (Networks of Workstations)*, IEEE Micro, Vol 15, No. 1, pp. 54-64, February 1995.

[7] Armstrong, R., and Macfarlane, J., *The Use of Frameworks for Scientific Computation in a Parallel Distributed Environment*, Proceedings of the Third International Symposium on High Performance Distributed Computing, San Francisco, California, pp. 15-25, August 1994.

[8] Bach, M. J., **The Design of the Unix Operating System**, Prentice-Hall Inc., 1986.

[9] Barak, A., Guday, S.,and Wheeler, R. G., **The MOSIX Distributed Operating System, Load Balancing for UNIX**, Lecture Notes in Computer Science, Springer-Verlag, Vol. 672, 1993.

[10] Barak, A., Laden, O., and Yarom, Y., *The NOW MOSIX and its Preemptive Process Migration Scheme*, Bulletin of the Technical Committee on Operating Systems and Application Environments, IEEE Computer Society, Vol. 7, No. 2, pp. 5-11, June 1995.

[11] Barak, A., and Wheeler, R. G., *MOSIX: An Integrated Multiprocessor UNIX*, Proceedings of the Winter 1989 USENIX Conference, San Diego, CA, pp. 101-112, February 1989.

[12] Barbacci, M. R., Doubleday, D. L., Weinstock, C. B., and Wing, J. M., *The Durra Language and Runtime Environment: Tools for PMS Level Programming*, **Decentralized Systems**, Proceedings of the IFIP WG 10.3 Working Conference, pp. 25-27, North Holland, 1990.

[13] Barbacci, M. R., Weinstock, C. B., Doubleday, D. L., Gardner, M. J., and Lichota, R. W., *Durra: a structure description language for developing distributed applications,* Software Engineering Journal, pp. 83-94, March 1993.

[14] Barbacci, M. R., and Wing, J. M., *Specifying Functional and Timing Behavior for Real-Time Applications*, Proceedings of Parallel Architectures and Languages Europe (PARLE) Volume 2. Springer-Verlag, Lecture Notes in Computer Science, Vol 259, pp. 124-140, June 1987.

[15] Benmohammed-Mahieddine, K., Dew, P. M., and Kara, M., *A Periodic Symmetrically-Initiated Load Balancing Algorithm for Distributed Systems*, Proceedings of the 14th International Conference on Distributed Computing Systems, pp. 484-492, June 1994.

[16] Besana, V., De Luca, J., Giard, E., and Reger, J., **Experiences with SOMobjects: Distributed System Object Model (DSOM)**, IBM International Technical Support Centers, Document No. GG24-4165-00, November 1993.

[17] Betz, M. *Interoperable Objects*, Dr. Dobb's Journal, Vol. 19, No. 11, pp. 18-32, October 1994.

[18] Birman, K. P., Schiper, A., and Stephenson, *Lightweight Causal and Atomic Group Multicast*, ACM Transactions on Computer Systems, Vol 9, No 3. pp. 272-299, August 1991.

[19] Birman, K., Cooper, R., Joseph, T., Kane, K. and Schmuck, F., **The Isis System Manual.**

[20] Blumofe, R. D., and Park, D. S., *Scheduling Large-Scale Parallel Computations on Networks of Workstations*, Proceedings of the Third International Symposium on High Performance Distributed Computing, San Francisco CA., pp. 96-105, August 1994.

[21] Bricker, A., Litzkow, M. and Livny, M., *Condor Technical Summary*, Computer Science Department, University of Wisconsin - Madison, January 1992.

[22] Butler, R., and Lusk, E., *User's Guide to the p4 Programming System*, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439-4801, April 1994.

[23] Carriero, N., and Gelernter, D., *Linda in Context*, Communications of the ACM, Vol. 32, No. 4, pp. 444-459, April 1989.

[24] Carriero, N., Gelernter, D., Kaminsky, D., Westbrook, J., *Adaptive Parallelism with Piranha*, Department of Computer Science, Yale University, New Haven, Connecticut, YALE/DCS/RR-954, February 1993.

[25] Carriero, N., Gelernter, D., Mattson, T. G., and Sherman, A. H., *The Linda Alternative to Message-Passing Systems*, Parallel Computing, Vol. 20, No. 4, April 1994.

[26] Casavant, T. L., and Kuhl, J. G., *A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems*, IEEE Transactions on Software Engineering, Vol. 14, No. 11, pp. 141-154, November 1988.

[27] Chang, J., and Maxemchuck, N. F., *Reliable Broadcast Protocols*, ACM Transactions on Computer Systems, Vol. 2, No. 3, pp. 251-273, August 1984.

[28] Chen, J., Xie, L., and Sun, Z.-X., *An Intelligent Network Manager for Distributed Task Scheduling*, Proceedings of the 11th International Conference on Computer Communication, pp. 493-499, October 1992.

[29] Cheng, D.Y., *A Survey of Parallel Programming Languages and Tools*, RND-93-005, NASA Ames Research Center, March 1993.

[30] Cheriton, D.R., *The V Distributed System,* Communications of the ACM, pp. 314-333, Vol. 31, No. 3, 1988.

[31] Chiu, J-F, and Chiu, G-M, *Process-Replication for Fault-Tolerance and Performance Improvement in Distributed Computing Systems*, Proceedings of the Third International Symposium on High Performance Distributed Computing, San Francisco, California, pp. 236-243, August 1994.

[32] Chu, W. W., Holloway, L. J., Lan, M., and Efe, K., *Task Allocation in Distributed Data Processing*, Computer, Vol 13, No. 11, pp. 57-69, November 1980.

[33] Clark, H. and McMillin, B., *DAWGS a distributed compute server utilizing idle workstations,* Journal of Parallel and Distributed Computing, pp. 175-186, Vol.14, No.2, 1992.

[34] Dandamudi, S., *Performance Impact of Scheduling Discipline on Adaptive Load Sharing in Homogeneous Distributed Systems*, Proceedings of the 15th International Conference on Distributed Computing Systems, pp. 484-492, June 1995.

[35] Deremer, F., and Kron, H. H., *Programming-in-the-Large Versus Programming-in-the-Small*, IEEE Transactions on Software Engineering, Vol. 2, No. 2, June 1974.

[36] Eager, D. L., Lazowska, E. D., and Zahorjan, J., *A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing*, Performance Evaluation, Vol. 6, pp. 53-68, March 1986.

[37] Eager, D. L., Lazowska, E. D., and Zahorjan, J., *Adaptive Load Sharing in Homogeneous Distributed Systems*, IEEE Transactions on Software Engineering, Vol. 12, No. 5, pp. 662-675, May 1986.

[38] Elnozahy, E. N., and Zwaenepoel, W., *Replicated Distributed Processes in Manetho*, Proceedings of the Conference on Fault-Tolerant Computing Systems, pp 222-230, 1991.

[39] Efe, K., and Schaar, M. A., *Performance of Co-Scheduling on a Network of Workstations*, Proceedings of the 13th International Conference on Distributed Computing Systems, pp. 525-531, May 1993.

[40] Fabre, J. C., *Improving Fault Tolerance in Distributed Systems: The Saturation Approach*, Proceedings of the 5th Annual European Computer Conference (CompEuro '91), pp. 634-641, May 1991.

[41] Farrell, W. E., Fox, W. Given, J. W., MacRitchie, B., and Wang, J., *Data Processing on Networks of Workstations*, Bulletin of the Technical Committee

on Operating Systems and Application Environments, IEEE Computer Society, Vol. 7, No. 2, pp. 35-42, June 1995.

[42] Fox, G. C., et. al., **Solving Problems on Concurrent Processors, Volume 1,** Prentice Hall, 1988.

[43] Fox, G. C., **The Architecture of Problems and Portable Parallel Software Systems,** Northeast Parallel Architectures Center, Technical Report SCCS-134, Syracuse University, July 1991.

[44] Fox, G. C., and Furmanski, W., *The Use of National Information Infrastructure and High Performance Computers in Industry*, Proceedings of the 2nd International Workshop on Massively Parallel Processing Using Optical Interconnections, October 1995.

[45] Fox, G. C., Furmanski, W., Chen, M., Rebbi, C., and Cowie, J. H., *WebWorks: Integrated Programming Environment Tools for National and Grand Challenges*, Northeast Parallel Architectures Center, Technical Report sccs-0715, June 1995.

[46] Fox, G. C., Hiranandani, H., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C, and Wu, M., **Fortran D Language Specifications**, Northeast Parallel Architectures Center, Syracuse University, 1990.

[47] Garcia-Molina, H., and Spauster, A., *Ordered and Reliable Multicast Communication*, ACM Transactions on Computer Systems, Vol. 9, No. 3, pp. 242-271, August 1991.

[48] Geist, A., Beguelin, A., Dongarra, J., Weicheng, J., Manchek, R., and Sunderam, V., **PVM 3 User's Guide and Reference Manual**, Oak Ridge National Laboratory, Document No. ORNL/TM-12187, September 1994.

[49] Gelernter, D., and Carriero, C., *Coordination Languages and their Significance*, Communications of the ACM, Vol. 35, No. 2, pp. 97-107, February 1992.

[50] Ghafoor, A., and Ahmad, I., *An Efficient Model of Dynamic Task Scheduling for Distributed Systems*, Proceedings of the 14th Annual International Computer Software and Applications Conference, pp. 442-447, November 1990.

[51] Graham, I. S., **The HTML Sourcebook**, John Wiley & Sons Inc., 1995.

[52] Grimshaw, A. S., *The Mentat Runtime System: Support for Medium-Grain Parallel Computation*, Proceedings of the Fifth Distributed Memory Computing Conference, Los Alamitos, CA., Vol II, pp. 1064-1073, 1990.

[53] Grimshaw, A. S., *Easy-to-Use Object-Oriented Parallel Processing with Mentat*, Computer, Vol. 26, No. 5, May 1993.

[54] Hariri, S., *High Performance Distributed Systems: Concepts and Designs*, Artech House, in preparation.

[55] Hariri, S., Park, J., Yu, F., Parashar, M., and Fox, G., *A Message Passing Interface for Parallel and Distributed Computing*, Proceedings of the Second International Symposium on High Performance Distributed Computing, pp. 84-91, July 1993.

[56] Hariri, S., Yadav, R., Thiagarajan, B., Park, S. Y., Subramanyan, M., and Reddy, R., "A Concurrent Multi Target Tracker: Benchmarking and Portability", International Conference on Parallel Processing (ICPP), August 1994.

[57] Heidelberger, P., and Lavenberg, S. S., *Computer Performance Evaluation Methodology*, IEEE Transactions on Computers, Vol. 33, No. 12, pp. 1195-1220, December 1984.

[58] Hwang, K., and Briggs, F.A., **Computer Architecture and Parallel Processing**, McGraw-Hill, Second Edition.

[59] Jacob, B., **SOMobjects: A Pratical Introduction to SOM and DSOM**, IBM International Technical Support Centers, Document No. GG24-4357-00, July 1994.

[60] Ju, J., Xu, G., and Tao, J., *Parallel Computing Using Idle Workstations,* Operating Systems Review, Vol. 27, No. 3, 1993.

[61] Kaashock, M. F., and Tanenbaum, A. S., *Group Communication in the Amoeba Distributed Operating System*, Proceedings of the 11th International Conference on Distributed Computing Systems, pp. 222-230, May 1991.

[62] Kernighan, B. W., and Ritchie, M. R., **The C Programming Language - Second Edition**, Prentice Hall, 1988.

[63] Kramer, J., *Configuration Programming - A Framework for the Development of Distributable Systems,* The Proceedings of the IEEE International Conference on Computer Systems and Software Engineering (CompEuro 90), pp. 374-384, 1990.

[64] Kramer, J., Magee, J., and Finkelstein, A., *A Constructive Approach to the Design of Distributed Systems*, Proceedings of the 10th International Conference on Distributed Computing Systems, pp. 580-587, June 1990.

[65] Kramer, J., Magee, J., and Sloman, M., *Configuration Support for System Description, Construction and Evolution*, **Managing Complexity in Software Engineering**, pp. 247-260, edited by R. J. Mitchell, Peter Pereginus Ltd., 1990.

[66] Kremien, O., and Kramer, J., *Methodical Analysis of Adaptive Load Sharing Algorithms*, IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 6, pp. 747-760, November 1992.

[67] Krueger, P., and Cawla, R., *The Stealth Distributed Scheduler,* Proceedings of the 11th IEEE International Conference on Distributed Computing Systems, May 1991.

[68] Krueger, P., and Livny, M., *A Comparison of Preemptive and Non-Preemptive Load Distributing*, Proceedings of the 8th International Conference on Distributed Computing Systems, pp. 123-130, June 1988.

[69] Kunz, T., *The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme*, IEEE Transactions on Software Engineering, Vol. 17, No. 7, pp. 725-730, July 1991.

[70] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quaterman, J. S., **The Design and Implementation of the 4.3BSD Unix Operating System**, Addison-Wesley, 1989.

[71] Litzkow, M. and Solomon, M., *Supporting Checkpointing and Process Migration Outside the Unix Kernel,* Proceedings of the Winter 1992 USENIX Conference, pp. 283-290, 1991.

[72] Liu, R.-C., *Analysis on the Effects of System Parameters on Load Balancing*, Information Sciences, Vol. 81, No. 1-2, pp. 37-54, November 1994.

[73] Lord, H. D., *Improving the Application Development Process with Modular Visualization Environments*, Computer Graphics, Vol. 29, No. 2, May 1995.

[74] Mechoso, C. R., Farrara, J. D., Spahr, J. A., *Running a Climate Model in a Heterogeneous, Distributed Computer Environment,* Proceedings of the Third International Symposium on High Performance Distributed Computing, pp. 79-84, August 1994.

[75] MacDougall, M. H., **Simulating Computer Systems - Techniques and Tools**, MIT Press, 1987.

[76] Mattson, T. G., *Programming Environments for Parallel Computing: A Comparison of CPS, Linda, P4, PVM, POSYBL, and TCGMSG*, Proceedings of the 27th Annual Hawaii International Conference on Systems Science, Vol. II, pp. 586-594, January 1994.

[77] Mirchandaney, R., Towsley, D., and Stankovic, J. A., *Analysis of the Effects of Delays on Load Sharing*, IEEE Transactions on Computers, Vol. 38, No. 11, pp. 1513-1525, November 1989.

[78] Mullender, S.J., and van Rossum, G., *Amoeba: A Distributed Operating System for the 1990s,* Computer, pp. 44-53, Vol. 23, No. 5, 1990.

[79] Munetomo, M., Takai, Y., and Sato, Y., *A Genetic Approach to Dynamic Load Balancing in a Distributed Computing System*, Proceedings of the 1st IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence, Vol. 1, pp. 418-421, June 1994.

[80] Mutka, M. W., *Estimating Capacity for Sharing in a Privately Owned Workstation Environment,* IEEE Transactions on Software Engineering, Vol. 18, No. 4, 1992.

[81] Mutka, M. W., and Livny, M., *The Available Capacity of a Privately Owned Workstation Environment,* Performance Evaluation, pp. 269-284, Vol. 12, No. 4, July 1991.

[82] The Object Management Group, **The Common Object Request Broker: Architecutre and Specification**, OMG Document Number 93.12.43, December 1993.

[83] Parasoft Corporation, *Express Reference Manual,* 1988.

[84] Park, C., and Kuhl, J. G., *A Fuzzy-Based Distributed Load Balancing Algorithm for Large Distributed Systems*, Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems, pp. 366-273, April 1995.

[85] Peterson, L., Hutchinson, N., O'Malley, S., and Rao, H., *The x-Kernel: A Platform for Accessing Internet Resources,* Computer, pp. 23-33, Vol. 23, No. 5, May 1990.

[86] Prieto-Diaz, R., and Neighbors, J.M., *Module Interconnection Languages,* The Journal of Systems and Software, No. 6, pp. 307-334, 1986.

[87] Rehmet, G. M., **Remora - Implementing Adaptive Parallelism on a Heterogeneous Cluster of Networked Workstations**, Master's Thesis, Rhodes University, South Africa, September 1994.

[88] Rinaldo, F., and Wolbers, S., *Loosely Coupled Parallel Processing at Fermilab*, Computers in Physics, Vol. 7. No. 2, pp. 184-190, March 1993.

[89] Rosenberry, W., Kenney, D., and Fisher, G., **Understanding DCE**, O'Reily & Associates, 1992.

[90] Rousselle, P., Hariri, S., and Tymann, P, *Redundant Task Scheduling in a Heterogeneous Computing Environment*, Northeast Parallel Architectures Center, Technical Report SCCS-658, October 1994.

[91] Rousselle, P., Tymann, P., Hariri, S., and Fox, G., *The Virtual Computing Environment* Proceedings of the Third International Symposium on High Performance Distributed Computing, San Francisco, California, pp. 7-14, August 1994.

[92] Rousselle, P., Yadav, R., Hariri, S., Kim, D., Boddapati, S., and Tymann, P., *Application Development and Execution in a Virtual Computing Environment*, Northeast Parallel Architectures Center, Technical Report SCCS-705, April 1995.

[93] Schaar, M., Efe, K., and Delcambre, L. M., *An Analytical Model for Predicting Performance in a Heterogeneous System*, Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Computing Systems, pp. 328-335, May 1991.

[94] Shoja, G.C., *A Distributed Facility for Load Sharing and Parallel Processing Among Workstations,* Journal of Systems Software, pp. 163-172, Vol. 14, No. 3, 1991.

[95] Simpson, R. A., and Harget, A. J., *A Simulation Study to Determine the Importance of Load Balancing Algorithms for Loosely Coupled Distributed Systems*, IFIP Transactions on Computer Science and Technology, IFIP 13th World Computer Congress, Vol. A-51, pp. 69-74, August 1994.

[96] Stevens, W. R., **UNIX Network Programming**, Prentice Hall, 1990.

[97] Stroustrup, B., **The C++ Programming Language - Second Edition**, Addison-Wesley, 1991.

[98] Stroustrup, B., **The Design and Evolution of C++**, Addison-Wesley, 1994.

[99] V. S. Sunderam, **PVM: A Framework for Parallel Distributed Computing**, Oak Ridge National Laboratory, 1991.

[100] Tannenbaum, A. S., **Modern Operating Systems**, Prentice-Hall, 1992.

[101] Theimer, M., and Lantz, K., *Finding Idle Machines in a Workstation-Based Distributed System,* IEEE Transactions on Software Engineering, pp. 1444-1458, Vol. 15, No. 11, 1989.

[102] Theimer, M., and Hayes, B., *Heterogeneous Process Migration by Recompilation*, Proceedings of the 11th IEEE International Conference on Distributed Computing Systems, May 1991.

[103] Thornborrow, C., Wilson, A. J. S., and Faigle, C., *Developing Modular Application Builders to Exploit MIMD Parallel Resources*, Proceedings of Visualization '93, pp. 134-141, San Jose CA, October 1993.

[104] Waldspurger, C.A., Hogg, T., Huberman, B.A., Kephart, J.O., and Stornetta, W.S., *Spawn: a distributed computational economy,* IEEE Transactions on Software Engineering, pp. 103-117, Vol.18, No.2, 1992.

[105] Walker, B., Popek, G., English, R., Kline, C., and Thiel, G., *The LOCUS Distributed Operating System,* Proceedings of the 1983 SIGOPS Conference, pp. 49-70.

[106] Wang, C. -J., Krueger, P. E., and Liu, M. T., *Intelligent Job Selection for Distributed Scheduling,* Proceedings of the Thirteenth International Conference on Distributed Computing Systems, pp. 517-524, May 1993.

[107] Wood, M. D., *Replicated RPC Using Amoeba Closed Group Communication*, Proceedings of the 13th International Conference on Distributed Computing Systems, pp. 222-230, 1993.

[108] Zhou, S., *A Trace-Driven Study of Dynamic Load Balancing*, IEEE Transactions on Software Engineering, Vol. 14, No. 9, pp. 1327-1341, September 1988.

[109] Zhou, S., Zheng, X., Wang, J. and Delisle, P., *Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems*, Software - Practice and Experience, pp. 1305-1336, Vol. 23, No. 12, December 1993.

[110] Zhu, W., and Goscinski, A., *Low Cost Load Balancing Algorithms*, Proceedings of the 27th Annual Simulation Symposium, pp. 226-235, April 1994.

Biographical Data

Name:	Philip Rousselle
Date and Place of Birth:	September 10, 1962 Buffalo, New York
Elementary & High School:	Buffalo Public Elementary School #84 Graduated 1976
	Kensington High School Buffalo, New York Graduated 1980
College:	SUNY College at Buffalo BS (Info Systems Management), 1983
	Union College Schnectady, New York MS (Computer Science), 1989
Employment	International Business Machines Inc Endicott NY