RESOURCE MANAGEMENT SERVICES FOR PARALLEL APPLICATIONS

By

James C. Pruyne

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1996

Abstract

Users' demands for computational resources have been increasing dramatically, particularly for use by parallel applications that exploit multiple resources simultaneously. The number of computational resources has also been growing tremendously, leading to compute environments which are large, heterogeneous and dynamic. Due to these two trends, services of powerful Resource Management Systems (RMS) that can match customers with resources in such complex environments have become essential. This thesis addresses issues related to the structure, functionality and interfaces of such systems, and demonstrates their ability to enable customers with parallel applications to harness the computing capacity of large, heterogeneous collections of resources.

Our approach to building Resource Management Systems is based on a layered framework. By carefully defining the interactions between layers, we have achieved the high reliability necessary in a RMS. The most crucial abstraction utilized by a RMS is its representation of resources and customers. To address this concern, we developed an extensible structure called a "classified" that can describe both entities and the constraints that define when they can be matched

with one another. Classifieds also provide a structure for communication among the layers of a RMS.

The interface between an application and the RMS is one layer of our framework. At this layer, applications must be provided with communication and resource management services. To provide both types of services, we defined a method of interfacing an existing communication system to a RMS. With this interface, we created CARMI, a set of services that allow applications to request, utilize and gather information about resources. We also developed WoDi, a programming interface that uses CARMI to manipulate resources and simplifies writing master-workers style parallel programs. We demonstrated WoDi's effectiveness with a number of real-world applications.

Due to the dynamic nature of the compute environment, a RMS can often improve resource utilization by using checkpoints to reallocate resources among running applications. To improve checkpointing performance, we developed "checkpoint servers" that allow a RMS to control how checkpoints are stored. To extend the benefit of checkpointing to CARMI applications, we developed CoCheck which permits a RMS to checkpoint parallel programs.

Acknowledgements

I first wish to thank my advisor, Miron Livny, for steering me through my life as a graduate student and both challenging and aiding in completing my degree.

I also wish to thank my thesis committee, Raghu Ramakrishnan and Pei Cao for reading my dissertation, and Marvin Solomon and Parmesh Ramanathan for participating in my defense. Their comments have tremendously improved this dissertation, and helped me to organize and re-evaluate my work.

All of the work done for this dissertation would not have been possible without Condor, and Condor would not have been possible without the efforts of Mike Litzkow. I'm indebted to him not only for providing me with a sandbox to play in, but more importantly for helping me learn the right way to play. His willingness to discuss ideas and concerns has taught me an incredible amount about building computer systems. I have also benefited from contributions to the Condor project by numerous others. Some of these people include Emmanuel Ackaouy, Jim Basney, Huseyin Bektas, Dhruba Borthakur, Weiru Cai, Raghu Mallena, Sriram Narasimhan, Ajitkumar Natarajan, Rajesh Raman, Dhaval Shah, Todd Tannenbaum, and Derek Wright. To these, and all the other contributors to Condor (some of whom I've never even met), my heartfelt thanks. Also thanks to Hsu-lin Tsao for his implementation of the checkpoint server.

I have also benefited from contributions and interactions with people outside of the Condor project. Most notably has been the help and support of the PVM team, in particular Jack Dongarra, Al Geist and Bob Manchek. I'm also grateful for the collaboration with Georg Stellner on CoCheck. In addition, I've been fortunate enough to get feedback from users of my research. I thank Nigel Starling from Durham University and Adam Beguelin and Kip Walker at Carnegie Mellon University for installing my system, and letting me know what works and what doesn't.

I have been supported financially for my last three years of graduate school by an IBM graduate fellowship. I thank IBM and my sponsors for the honor of receiving this fellowship.

My friends who played in Rich's World, went to Taco Tuesday and Muskies' games, and played for the Correspondents helped make grad. school a fun time for me. Thank you all for the good times.

My parents and family are ultimately responsible for helping me reach this point. Without their love, support, and instilling a value of education I could not have pursued my goals this far. I'm grateful for everything you've taught me, and made me.

Finally, and most specially, I must thank my wife, Melissa. Her unconditional support through the ups and downs of graduate school has been my source of endurance. Words cannot express the gratitude I have. Quite simply, I could not have done this without her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Two undeniable trends in high performance computing are the increasing complexity of the problems users wish to solve, and the rapid proliferation of computing resources. These seem to be complimentary trends since increases in available resources should facilitate solving these larger problems. Unfortunately, this is not always the case simply because it is difficult to harness the computational power of all these resources. Due to this complexity, users are not always able to obtain solutions to their problems, and resources are often not fully utilized. To bring users and resources together, Resource Management (RM) services, which allow their customers to focus all of the available computing power towards solving their problems, are essential.

Resource management services are responsible for providing resources to individual customers' applications in a manner in which they can be effectively utilized. These services are fundamental to all programming environments because without them there simply are no resources on which to compute. As computational environments have evolved to complex, distributed systems, the availability of good resource management services has become increasingly important. The complexity of the environment is a result of a number of factors. First, the number of individual resources is large and constantly changing. New resources may be introduced to the environment at any time, and other resources may become unavailable due to failure, maintenance, or simply being decommissioned. Because of this large number of resources, the frequency of changes in the environment is high. Second, all resource pools are heterogeneous. Whether in relatively static ways such as processor architecture, or dynamic ways such as disk space or processor load, every resource in the environment differs from every other resource. This dynamic and varied resource environment is impossible for individuals to monitor and therefore they are unable to make effective resource utilization decisions. Resource management services, therefore, must insulate their customers from the dynamics of the environment and provide a consistent environment for running applications, regardless of the state of the underlying resources.

Not only are distributed resources nearly impossible for a person to keep track of, it is also frequently the case that the *ownership* of resources is distributed. When ownership is distributed, users typically have direct access only to those resources owned by their group or department. Access to resources owned by other groups is not permitted, even if the owners are not using them. Good RM services can overcome these ownership barriers by providing customers access to resources owned by other groups, while enforcing usage policies agreed upon by the groups.

Because parallel applications use multiple resources concurrently, they are especially dependent on having powerful RM services. In particular, a parallel application must be able to gather a collection of resources for simultaneous use. Typical parallel applications require not only must there be the proper quantity of resources, but also that these resources have the proper characteristics. For example, a parallel application may be carefully load balanced to perform well on a particular number of homogeneous resources. If the wrong number of resources with different processor performance characteristics are provided, the load balancing will not be effective. More sophisticated, malleable [1], applications may be able to adapt at run-time to different resource allocation levels. To do this, though, the application must have services that notify it when resources become available or unavailable. Additionally, such applications must be provided with sufficient information to make intelligent use of the resources allocated to them. Malleable applications are desirable because they can lead to high resource utilization. By adapting, they can expand to utilize idle resources or contract to free resources for other applications. A common side effect of developing malleable applications is that they become fault tolerant. Fault tolerance is especially desirable for parallel applications because the large number of resources being utilized, and the typical long running times of these programs, makes failures increasingly likely. A failure means that not only must a computation be re-started, but also that the resource time dedicated to the application has been wasted.

In addition to RM services, parallel applications also require Inter-Process Communication (IPC) and synchronization services to allow their independent execution threads to coordinate and share data. The development of IPC services has been an active area of investigation, and has led to the development of a number of parallel programming systems. The most common communication paradigm provided by these systems is message based, so we refer to these as Message Passing Environments (MPEs). Due to the fundamental need for RM services, each of these systems does provide them, but they have not been the principle focus of the MPE development efforts. Therefore, the RM services provided are simplistic, and place a great burden on programmers to locate and determine how to utilize resources. This lack of robust RM services has limited the utility of these systems in many environments, or has required additional, proprietary extensions to be developed to suit a particular environment. This situation has contributed to the lack of acceptance of some of these systems, particularly in compute intensive, and heterogeneous production installations.

1.1 Resource Management Services for Parallel Applications

A much more effective approach to providing RM services to parallel applications is to allow them to take advantage of the services already provided by a Resource Management System (RMS). The job of a RMS is to monitor resources and provide its customers access to them to run their applications. Typically, the applications executed by a RMS are long running, and compute intensive. Parallel applications are also generally long running and compute intensive, so it seems natural that they could benefit from a RMS' services. Unfortunately, there has been little effort to develop a RMS which works with a parallel application to ensure it has its desired resources throughout the course of its run.

This thesis addresses issues related to providing RM services to parallel applications, particularly by utilizing existing RMS and MPE technologies to create a powerful environment. By combining these two types of systems, we give application developers access to new RM services provided by a RMS without sacrificing their experience with existing MPEs. This approach also allows researchers to focus on one component while benefiting from work done on the other. Generating a combined system which satisfies the needs of parallel applications has presented us with a number of fundamental challenges. These include how the RMS will bring resources and applications together, how the two formerly distinct systems will communicate and exactly what RM services should be provided to the applications.

1.1.1 Matching Resources and Customers

The basic purpose of a RMS is to bring its customers and the resources it controls together. This purpose is the same whether the applications customers wish to run are parallel or sequential. To do this, a RMS must have a method of representing the resources and customers in an expressive way which also permits it to determine when a match can be made. All other RM functionality will be constrained by the expressiveness of these representations, so a flexible structure is essential. To address these requirements, we have developed an extensible, symmetrical structure for representing both resources and customers. Because it is extensible, this structure can be used to represent any type of resource or customer in a RMS, and can grow to provide more details about these entities as needed. In addition, because it is symmetrical, it allows not only customers to specify what sort of resources they need, but also resources can indicate the conditions under which they will serve a customer. These conditions clearly specify when a match can be made, and simplify a RMS by providing a single structure which can be used for all entities within the system.

1.1.2 Interfacing a MPE and a RMS

Allowing a MPE and a RMS to work together to provide services to a single parallel application requires close communication and a well defined division of responsibility between the two systems. We also require a method for the application to request services from either component in a consistent manner. We have developed a method of interfacing these two types of systems based on the notion that a RMS can communicate with both the MPE and the application using the communication services provided by the MPE. This provides the RMS with a simple manner of interacting with the MPE, and allows us to clearly define the services provided by the MPE. By using the MPE services, the RMS and the application can communicate just as easily as the components of the application. This permits us to develop whatever RM services we feel are necessary without modification to the MPE. To access RM services, we have developed an asynchronous model in which an application registers requests via a procedure call, but receives the results later via a MPE message. This permits a single application process to have multiple requests outstanding simultaneously, and continue to communicate with other processes normally while the requests are being serviced.

1.1.3 Run-Time Services

Given the separation of responsibility between a MPE and a RMS, and the means for an application to make a RM service request, the next challenge is to determine what services should be provided. We have broken the problem down further by asking what services are essential to allow an application to effectively utilize resources, and what services provide non-essential, but still useful functionality. We started by developing a programming interface which provides the essential services. This core set of services uses the asynchronous model defined previously, and allows an application to request or release resources, and utilize resources by creating processes. These services also permit an application to gather information about the resources allocated to it either by retrieving all of the RMS' knowledge of the resource or by requesting notification when the state of the resource changes. These informational services are extremely valuable to a parallel application because they enable it to make intelligent resource utilization decisions and adapt to changes in the resource environment in which it runs.

Application Frameworks

Utilizing these core services to write an efficient application requires considerable thought and effort. It is valuable, therefore, to provide a set of easy to use services that efficiently handle resources for the programmer. We refer to these types of services as application frameworks because they provide a model for writing a specific style of parallel application. Programmers benefit both from an application framework's ease of use and the effort invested in it to use resources intelligently. We have developed one framework which uses our core RM services to support master-workers style applications. Our framework provides applications with services which not only insure that all of its work is completed, but also that its resources are used as efficiently as possible. Our framework strives for efficiency by scheduling work items and by determining proper resource levels that insure that all resources are used productively. Because master-workers is a common model for developing parallel applications, we have been able to apply our framework to a variety of problems.

Checkpointing

We have also investigated services which, while not essential, allow an application or RMS to more efficiently utilize resources. Checkpointing is an example of this type of service because it allows a running application to be repartitioned onto new resources as the availability of resources changes. The utility of checkpointing is limited by the overhead involved in creating a checkpoint, so we have devised a means of allowing a RMS to carefully control how checkpoints are stored to help reduce the time required. We have also developed a set of services for checkpointing parallel applications which fit into our overall model of developing RM services. This requires utilizing the MPE to RMS communication infrastructure to both manage the checkpoint as well as insure that no communication primitives fail as a result of the checkpoint.

1.2 Implementing a Resource Management System

A crucial attribute of a RMS is that it must run in "production" mode to be effective. That is, it must run continuously, 24 hours-a-day, 7 days-a-week for it to be of value to its customers. To survive in this environment, a RMS must be able to satisfy three groups. In decreasing order of importance, these groups are resource owners, system administrators, and customers. Owners are the most important group because they have ultimate control of the resources. If they do not believe a RMS will be beneficial, there simply will not be any resources in the pool. System administrators must feel confident that the system can run steadily without consuming too much of their time with maintenance. If administrators feel the RMS requires too much attention, they simply will remove it. Customers who utilize the system are in most ways the easiest group to please. As the beneficiaries of the system, customers are often more flexible than owners or administrators. However, they must be able to rely on a RMS' availability or else it will not be considered trustworthy, and will fail to satisfy their needs.

Building such a reliable distributed system is a complex task, particularly considering a RMS' fundamental requirement to adapt to failures and new arrivals of resources. The success of a RMS can therefore only be assessed when a system is running in production mode with customers who rely on it daily to accomplish their work. The ability to meet these goals by providing an environment for scientists to perform computationally intensive experiments has been a primary concern of the work presented in this dissertation. To increase the reliability of our system, we have taken a layered approach to building this system in which each layer has well defined roles, and well defined interactions with other layers. The use of a layered approach to architecting complex systems has been shown to be effective in generating robust system software dating back as far as Dijkstra's "The" [2] system in 1968. Our layered framework is fundamental to all of our RM services development, so it is important to first understand it.

One of the reasons for the current deficiency in RM services is the scope of the problem of providing these services. To be robust, RMS's must be built using an "end-to-end" approach that brings together individual pieces of hardware, customers and application developers. This requires a strong infrastructure, including close interaction with and monitoring of individual resources at one end of the spectrum, and abstractions and interfaces for application developers and customers to use resources at the other end. Our layered structure for implementing a complete RMS is shown in figure 1. Each layer provides information required by other layers as well as abstractions to simplify their interaction.

The most basic layer of a resource management system is the hardware resource itself which is generally tied closely to system software, such as an operating system, for managing the local, private resources. An important

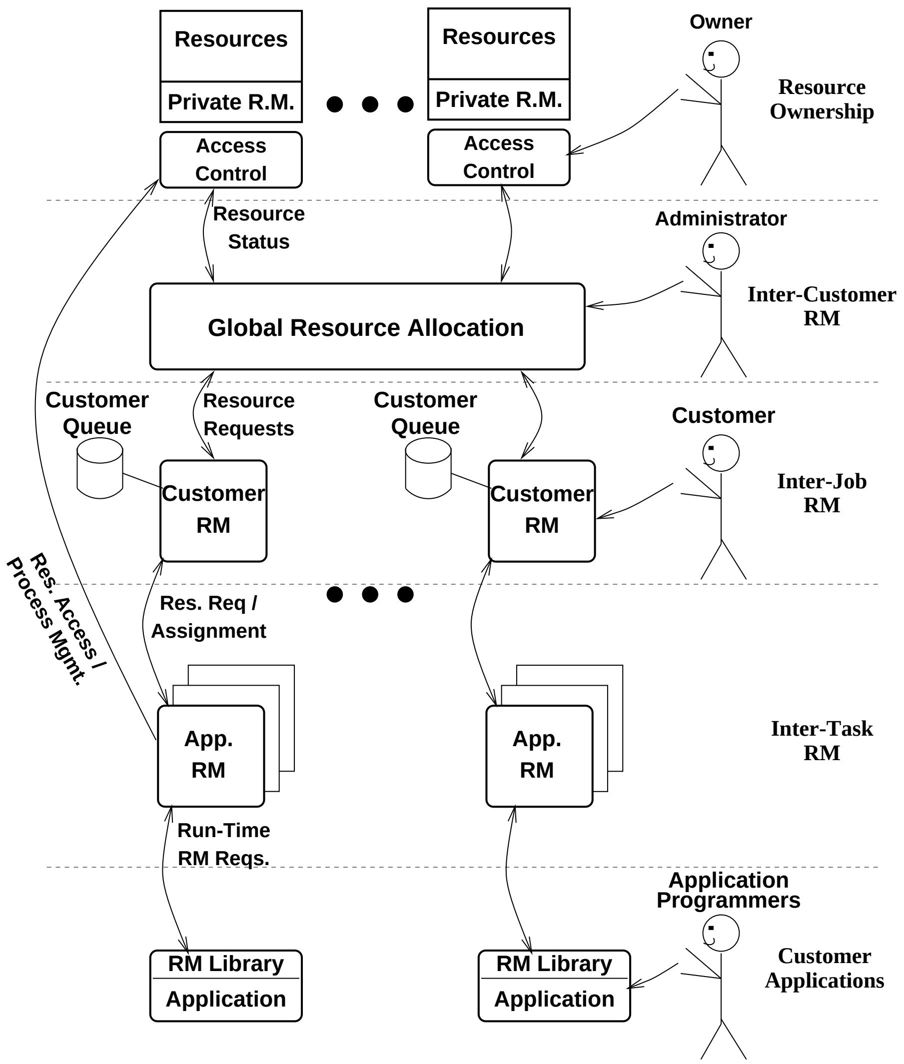

Figure 1: The layers of a Resource Management System

consideration for the RMS is that resources may have *owners* who impose policy on resources beyond the RMS itself. The RMS must, therefore, provide an *access control* mechanism which interacts with and enforces the owner's policy. Within the constraints of these policies, it is also up to the access control mechanism to inform the RMS' *resource allocator* of the characteristics and availability of the resource.

The resource allocation layer of the RMS is primarily a *matchmaker*. It determines which resources will be allocated to which customers. To do this, it must receive information both from resources about their characteristics and availability, and from customers about their needs. When making these matches, the resource allocator also implements the highest level, or *Inter-Customer scheduling policy* in the system. This policy dictates the priority among customers contending for the same resources. Customers store their resource requests in queues from which the resource allocator receives them. The distinct queues provide the resource allocator with the customer abstraction, but a queue may actually be shared by a group of individuals. From the perspective of the resource allocator's scheduling policy, it is a single logical entity with the same priority level.

Once a resource is allocated to a particular customer by the global resource allocator, it is up to the *customer resource manager* associated with the queue to make *Inter-Job* RM decisions. This requires deciding which job within the queue will be granted the resource. The customer RM can use any desired queuing discipline to implement a local priority scheme. When a resource is assigned to a particular job, it is passed to the *application resource manager*. One application RM runs for each active job. The application RM is responsible for communicating with the resource's access control module to establish the job's run-time environment. The application RM also provides the run-time services the application requires for querying, utilizing and requesting more resources. In implementing these services, the application RM performs *Inter-Task* prioritization by determining which resource will be used in fulfilling which application request. The utilization services can usually be carried out directly by the application RM and the resource access control module while other services will require the application RM to interact with the customer RM to gather new resources.

Developing a model for the structure of a RMS, such as the one described here, is essential for providing the required level of reliability. By clearly defining the roles of each layer, and the interfaces between them, it is possible to develop the system a module at a time, and to locate errors in the system by tracking the interaction between these layers. Without this structure, such a complex distributed system would be difficult to maintain, and in some cases errors in one area of the system may lead to failures in another logically unrelated area. These sorts of errors are particularly difficult to locate and solve dependably.

1.3 Thesis Structure

The remainder of this thesis describes how we have built on our layered framework to provide powerful resource management services to parallel applications. The fundamental goal of a RMS is to bring resources and customers together. The most basic issue in performing this function is how resources and customers will be represented. As described in chapter 3, we have developed a flexible structure called a *classified ad* which can be used to represent both resources and customers. In addition, this structure provides a method of describing how these two entities specify when they can be matched together. Chapter 4 details our work to establish the necessary framework for interfacing an existing MPE to a RMS. By utilizing this framework, we permit programmers to continue to use the MPE services with which they have become familiar, but open up new possibilities for RM services. Using this framework, we have gone on to develop CARMI, a set of RM services intended to give parallel applications the power to effectively utilize all resources under control of a RMS. Building on top of CARMI, we wanted to make our services as easy to use as possible. Therefore, we developed an application framework which is described in chapter 5. The framework we built, WoDi, provides a simple interface for writing master-workers style parallel programs. Finally, we turned our attention to checkpointing which is an RM service which can increase the utilization of the resources in the environment. In chapter 6 we describe development of a *checkpoint server* which permits an RMS to have fine control over how checkpoints are stored. We have also taken part in developing CoCheck, a method of checkpointing programs which communicate using messages. CoCheck is consistent with our RM service implementation methodology and allows us to extend the advantages of checkpointing services to parallel, message passing applications. CARMI was then extended to make CoCheck's checkpointing services directly available to application developers.

An important goal of the work described here has been to generate a useful, production quality system for use by computational scientists. To do so, we required a testbed for the implementation of our work. From the resource management side, we have utilized Condor [3] as a foundation on which to build. Condor is a RMS designed to work in an opportunistic environment of privately owned workstations. In this context, opportunistic means that resources are used whenever they are not actively utilized by their owners, but have to be returned whenever the owners require them. The work described in this dissertation has been integrated with Condor, and has increased the system's flexibility in how it manipulates resources, and in the types of applications it can run for its customers. The MPE we built on was PVM [4] which was originally designed to support parallel programming on networks of workstations, but which has also been used on dedicated parallel computers. The target PVM environment was consistent with that of Condor, so it provided a good match for our development efforts. Like our work with Condor, the enhancements we have made to PVM have been integrated with the production system, and are available to all PVM programmers and RMS developers. Our work has also had an impact on the continuing development of the MPI [5] standard. The MPI standard committee is now considering RM service extensions, and the terminology and basic structure is consistent with our framework. As we have developed each component of the system, we have strived to find customers with problems which we can help solve. This benefited the customers because they received access to new resources. We have also benefited because we were able to evaluate our framework in a real-life production environment.

Chapter 2

Related Work

In this chapter, other work related to the problem of providing resource management services to parallel applications is presented. In the past, researchers have approached the problem from two directions: the development of systems for managing distributed resources, and the development of environments for doing either parallel or distributed programs. While both areas have been studied extensively, we find little previous work which focuses on unifying the two areas of study.

2.1 Resource Management Systems

Some of the most ambitious efforts at managing resources in distributed systems have been "distributed operating systems." The philosophy behind these systems has been to extend the notion of the single node operating system into the distributed environment. Where the single node operating systems generally present a virtual machine model, the distributed operating systems have attempted to extend the virtual machine model to include distributed resources. Examples of distributed operating systems include Mach [6], the V-Kernel [7], Amoeba [8], Sprite [9], and Locus [10]. The scope of these systems with respect to resource management varies.

Mach and the V-Kernel are the most simple of these systems with respect to resource management. These "microkernels" do not provide any notion of global resource management. They do, however, provide services for transparent communication between distributed resources. Any desired global RM services must be built on top of these kernels' basic communication facilities.

Amoeba, Sprite and Locus take a more aggressive approach to resource management. Amoeba provides a "processor pool" on which it transparently starts users' programs in order to balance the load across all of the processors. Sprite uses migration to perform its resource management functions. It is able to migrate processes onto idle workstations, and then migrate them away to another idle workstation or the process creator's workstation when the remote workstation owner returns. Locus periodically measures the load on each of the resources in its system, and performs migrations in an effort to balance the load.

While all of these distributed operating systems can be considered resource management systems, their scope and ultimate goal is different than ours. These systems strive to be general purpose computing environments, and in most cases they attempt to hide the distributed nature of resources in an effort to provide a “single system” image. That is, they are attempting to make a number of distributed resources appear the same as traditional, time shared systems. Techniques such as automatic load balancing support this model. However, we want our customers to be aware of the multiple resources so that they can intelligently utilize them for their parallel applications. We therefore do not support the idea of automatic load balancing. Instead, we use queues to store jobs which cannot be allocated their required resources.

Two other systems, the Load Sharing Facility (LSF) [11] which is a product based on the research system Utopia, and GLUnix [12] are also attempts to build a single system image. Unlike the distributed operating systems, LSF and GLUnix are built on top of pre-existing single node operating systems. Both systems allocate interactively invoked programs on lightly loaded nodes in an effort to maintain a balanced load across all nodes. Once again, the single system image model is contrary to our parallel programming model. Both of these systems, however, appear to be moving toward supporting parallel programs. Indeed, LSF has adopted the RMS to parallel programming framework we developed and which is described in chapter 4. We are curious to see if they continue in this direction by providing more advanced RM services to parallel applications. GLUnix is still in an early stage of development, and it is not clear what its approach to communication with parallel applications will be.

Another approach to developing an RMS is exemplified by a variety of batch systems. Examples of these include the Distributed Queueing System (DQS) [13], LoadLeveler [14] from IBM, and the Portable Batch System (PBS) [15]. All of these were designed with the intention of executing long running sequential or parallel applications. These systems, however, allocate resources on a one-time basis, and following the initial allocation do not communicate with the application any longer. Because of this, applications running in these environments are unable to change their resource allocation level after start-up. Like LSF, LoadLeveler has adopted our interfacing strategy, but is so far not using it to continue communication with an application after start-up.

2.1.1 Scheduling

Scheduling is a crucial component of any RMS. In our layered structure for building a RMS, scheduling takes place at a number of levels. The highest level of scheduling is the Inter-Customer scheduling performed by the global resource allocator. Broadly defined, the goal of an Inter-Customer scheduling algorithm is "fairness." In other words, these algorithms wish to insure that each customer gets their proper share of the resources within the system. However, how this share is defined varies. A number of economic models [16] have been used for this purpose. Other algorithms such as Fair Share [17] and Up-Down [18] have also been proposed.

The next level of scheduling within our framework is Inter-Job. Inter-Job scheduling for parallel applications has received a great deal of study [19]. Indeed, entire workshops have been held on the topic [20, 21]. The goal of Inter-Job scheduling has typically been to reduce the average response time of queued jobs. Our goal has been to provide a framework in which a variety of scheduling models may be investigated and tested. Development and evaluation of individual scheduling approaches is outside the scope of this thesis.

2.2 Parallel Programming Systems

The last area of research which relates to what we present in this thesis is work on parallel and distributed programming environments. Clearly, since we wish to run parallel programs, these efforts have been of interest to us. The most widely studied model for parallel programming has been message passing. This model has lead to the development of a number of systems including Express [22], PVM [23, 4], MPI [5, 24, 25], P4 [26], TCGMSG [27], LAM [28] and others. All of these systems provide a set of services which allow parallel applications to communicate by passing messages between processes. From the perspective of our research, the interesting portion of these systems is how they manage resources, and what RM specific services they provide. In this area, most of these systems are greatly lacking, and in fact, among the systems listed here, only PVM provides any form of run-time interface for managing resources. PVM's RM services all require the user to specify the names of individual resources to be used. This provides applications a means of accessing new resources while they run, but it still requires that users make all of the resource allocation decisions. The other systems require their users to provide a list of resources to be used by the application, and then simply start the application running on those resources. MPI deserves a special mention because it is a standard, not a single implementation. In the standardization process, RM services were intentionally omitted, but requests from users have lead to a follow up standard, MPI-2 [29], which will include some RM services.

In addition to these message passing systems, a number of other approaches to parallel programming have been proposed. Among them, Linda [30] and Cilk [31], are interesting from a RM perspective. Linda uses the notion of a "tuple-space" for communication between application processes. The tuple-space allows application processes to store, read or remove typed collections of data. Linda provides a single RM-like function for creating a new process and assigning it a tuple to compute. The Linda model has been extended in a system called Piranha [32] which utilizes idle workstations for running Linda processes. As we describe in chapter 5, this model is similar to our master-workers application framework. Piranha does not, however, address concerns related to interfacing to a RMS, which is central to our work.

The Cilk programming model is based on the notion of a parallel function call. Programmers annotate normal procedural programs to specify that certain functions should be evaluated in parallel. Like Piranha, the Cilk run-time can execute these function evaluations on idle workstations. However, also like

Piranha, the Cilk researchers have made no effort to interface to an existing RMS.

While many systems for managing resources and performing parallel computations have been developed, there has been little previous work on interfacing the two types of systems. This is surprising because parallel programs benefit from having resources readily available, and the goal of a RMS should always be to provide services its customers need to run their applications. In this thesis we provide new approaches for bringing these two types of systems together, and for providing RM services to parallel applications. We believe that our results will complement past and future work on both types of systems, and lead to better environments for parallel processing.

Chapter 3

Bringing Resources and Customers Together

The fundamental purpose of a resource management system is to bring its *customers* and *resources* together so that computations can be performed. To perform this matchmaking, a RMS must first of all have a method of representing resources and customers. A general and flexible method of representing resources and their usage policies gives a RMS power in the type and manner in which it will manipulate resources. A lack of power in the representation will constrain the RMS' ability to perform its job. Likewise, the method of representing customers must be expressive enough to permit them to precisely describe themselves and their needs in a way in which the RMS can interpret them.

The most important area of flexibility in resource representations is the type of resources which can be represented. In a general RMS, not all resources of interest are compute nodes. Other possibilities include, for example, software licenses and communication channels. In essence, any allocatable physical or logical entity within a computing environment may be under the control of a RMS. A representation technique must, therefore, avoid any assumptions about the nature of the resources to be represented. For example, a method assuming that every resource has an attribute such as a processor architecture would be unable to represent a resource like a communication link which has no processor.

In addition to the differences in types of resources, individual resources also vary in a number of static and dynamic ways, some of which may not be obvious a priori. It may be immediately apparent, for example, that a static difference among workstations is that they have different types of microprocessors. Other static differences are not so obvious. Even machines that can execute the same binary files may vary greatly in aspects such as their processing speed, the amount of memory they have, and their position on the network. Examples of dynamic differences among resources include the amount of free disk space or the load on the processor. Furthermore, the environment itself fluctuates. Resources are not always available. They may be powered down or otherwise revoked from the control of the RMS. Likewise, new resources may be added to the environment. When combined with the notion that even the basic function of the resources may be unknown ahead of time, it is apparent that it is not possible to make any assumptions about general or specific characteristics of entities being represented. These representations must be able to adapt as the resources themselves change. The resource representation must not only be flexible enough to represent the widely varying resources in the environment, but it must also facilitate the desired resource allocation and scheduling policies. For example, a scheduling policy which uses information about the available memory on a compute node must be provided with information about memory at each node in order to implement its policy. Individual resources may also have usage policies associated with them outside any global scheduling policy implemented in the RMS. It must be possible then to express the policies, and associate them with the resource so they will be enforced by the RMS.

Just as the method of representing resources must be extensible and flexible, so must the manner in which customers are identified. The attributes of individual customers with respect to the RMS will vary just as resource attributes vary. Because of the variety in types and characteristics of resources, customers must be allowed to make requests which are equally varied. In addition, it should be possible for customers with simple requirements to have a simple method of making a request, while those with complex requirements have the power to express their needs.

Most resource management systems rely on the abstraction of a *queue* to provide the needed resource classification. In a queue based system, a customer submits a job to one of a number of queues defined by the system administrator. Usually, these queues are given names which specify the characteristics of the jobs which will be run there. For example, a queue might be called "overnight" if it is intended to run long jobs during night time hours. The system administrator also specifies which resources within the pool will serve jobs from each queue. Once this assignment is made, a resource will generally serve only jobs submitted to this queue, even if it is empty and there are other jobs waiting in other queues. In this scheme, the matching between the customer and resource is done by the customer at the time a job is submitted. Examples of queue based systems include NQS [33], DQS [13] and the `lsbatch` extension to LSF [11]. Queue based systems such as these provide a very rigid method of describing resources. This technique does not permit customers to specify the characteristics of resources which are important for their jobs. Customers may be forced to "shoe horn" their jobs into an inappropriate, but closest matching queue. It also puts a great deal of burden on the administrator. For every new resource, the administrator must decide which queue it will serve. The assignment of resources to queues may also have to be re-balanced as resources fail or are retired, or as the load imposed by customers changes.

We have developed a novel approach to the resource and customer representation problem called *Classified Advertisements* [34]. The classified advertisement scheme is modelled, to a certain extent, after the concept of a newspaper classified or personal advertisement. This is because, at a high level, the goal of a RMS is the same as the goal of newspaper classifieds. That goal is to bring together two parties who are looking for one another. In the case of a newspaper personal ad, the parties are looking for potential dates. In the case of a RMS, the parties are customers and resources. Either way, the match is made by each party advertising something about itself, and something about their desired match. Unlike the queue method which exemplifies the asymmetrical client-server model, this is a symmetrical relationship. Not only is the customer searching for a resource, but a resource can be viewed as searching for a customer. Because the relationship is symmetrical, a resource can specify as much about its customer as the customer does about it. This symmetry also permits us to use a single structure, the classified ad, to represent both customers and resources. In the rest of this chapter, we describe the components of a classified ad, and how it facilitates matchmaking between customers and resources. We also describe how the basic classified model allows us to support other operations such as breaking a match or executing parallel jobs. Finally, we demonstrate the use of classifieds in a production RMS, Condor, and how they allow us to extend the RMS across multiple administrative domains.

3.1 Classified Ads

We developed classified ads with the intention that they represent all entities within a RMS and facilitate efficient matchmaking between entities. Thus, our classified ad model consists of four components: a *self type*, a *match type*, an *attribute list*, and a *constraint*. The self and match types provide a very basic description of the entity publishing the ad and its desired match. It is similar to the column headings found in newspaper classified ad sections which say something like "Men seeking women." These help the reader to avoid scanning every single ad to find one of interest. Likewise, the type fields permit the resource allocator to avoid exhaustively searching for matches between all entities publishing classifieds.

We designed the attribute list to be flexible enough to accommodate ongoing changes in resources and customers alike. The attribute list provides the self description, and consists of an arbitrary collection of assignment statements of the form `name = expression`. Each attribute also has a data type (e.g. integer, floating point or string) associated with it. The attribute list is the key to the classified's ability to represent resources and customers of any type, with whatever characteristics are appropriate. When new resource types are added to the system, an attribute list appropriate to the resource can be created, and the resources can be advertised to the system. New names can be added to the attribute list over time as more characteristics become important to the functioning of the RMS. Likewise, attributes which lose significance can be deleted from the list. Since we place no restrictions on the size of the list or the attribute names which can be used, their use can evolve with the system and with the customers' needs.

The constraint is a boolean expression which is used to describe the conditions in which an entity can be matched with another. This expression consists of attribute names in the classifieds of both the entity publishing the classified, and its potential match. We chose to use a boolean expression because it provides tremendous flexibility in the match criteria which can be expressed, but can also be evaluated efficiently and unambiguously. Example constraints, and how they are evaluated, will be discussed in the next section.

Classified ads can be viewed as "self describing" because all of the information about the entity it represents is carried inside of it. This is in some ways similar to distributed object systems such as CORBA [35] which contain a description of each object along with the object itself. The goals of object systems and classified ads are very different however. The distributed object systems provide a description of executable methods which can be performed by the objects, while we created classified ads solely to permit a RMS to match resources and customers. A collection of example classified ads is shown in figure 2. These include classifieds for a workstation type resource, a customer's request for a workstation, and a software license. In these examples, the self and match types are shown at the top, followed by attribute lists, and constraints.

The first example in figure 2 is the classified for a workstation type resource. First are the self and match types. This resource is a "Workstation" which is looking for a "job" which is the common name for a customer's resource request. The next section is the attribute list. The first few attributes give the name of the resource, the state of the machine (in this case, a customer's job is running there), and some details about the architecture and the operating system. More information about this particular workstation is provided next, including the

SelfType	`Machine`
MatchType	`Job`

Attribute List	`Machine = sun12`
	`State = Running`
	`EnteredState = 807463222`
	`OpSys = SunOS`
	`Arch = sun4m`
	`Cpus = 1`
	`Memory = 31`
	`Disk = 143355`
	`SwapSpace = 80636`
	`MIPS = 45`
	`KFLOPS = 5286`
	`ConsoleIdle = 73231`
	`KeyboardIdle = 0`
	`LoadAvg = 0.086`
	`ClientMachine = sun30`
	`RemoteUser = joe`
	`ClockDay = 4`
	`ClockMin = 652`

Constraint	`LoadAvg < 0.5 &&` `Owner == ‘‘joe’’`

SelfType	`Job`	`License`
MatchType	`Machine`	`Job`

Attribute List	`Owner = joe`
	`Universe = CARMI`
	`Executable = a.out`
	`State = Idle`
	`ImageSize = 1000`
	`RemoteCPU = 0`
		`Package = MathLib`
		`Host = mathhost`
		`Count = 10`
		`Avail = 5`

Constraint	`Memory > 32 &&` `OpSys == ‘‘SunOS’’`	`Count > 0`

Figure 2: Sample Classified Ads

amount of memory, free disk space, and free virtual memory space. Also, a measure of the machine's CPU performance (as measured by periodic tests by the RMS software) is included. Information about the current utilization of the machine, including idleness (both on the console and from remote logins) and current load is listed. Details about the usage of the machine by the RMS is also given by naming the machine and customer who's job is currently running there (in this case, a job submitted by `joe@sun30` is running). Finally, the workstation's clock's day and time are included.

The second example shows how the classified ad structure can be used to represent a customer's request for resources. Here we see an attribute list for a job looking for a workstation which was submitted by customer "joe." The job will run in universe "CARMI." The universe of the job specifies what programming paradigm is being used by the job, and helps the RMS to establish the proper run-time environment including the proper application resource management process when it starts the job. In this case, the job uses the CARMI parallel programming environment which is described in the next chapter. Attributes also show the name of the program the job wishes to run (`a.out`), the current state of the job (currently, it is idle in the customer's queue), the size of its executable, the time stamp when the job was submitted, and the amount of processor time the job has accumulated thus far. In this example, the request was recently been made, and the job has not yet begun running.

The last example shows how a software license is represented using a classified ad. It contains the name of the software package, the host which must be contacted to obtain a license, and the total number of licenses purchased. As licenses are obtained and released, the number of licenses available would change, and the license attribute list would be updated.

The attribute list provides an extremely flexible and extensible method for describing the resources and other entities within the RMS. Because they are updated as resources change (for example whether a particular machine or job is currently running or idle), they also provide an excellent method of monitoring the state of the RMS. By viewing each resource's attribute list we can determine what it is doing within the system, and by monitoring them over time we can see trends in the utilization of the system. Alone, though, attribute lists do not provide a method of deciding which resources should be allocated to which jobs. For example, by looking at the attribute list of the resource request, we cannot determine what sort of resource is actually desired. Would it be safe to allocate the workstation to the job in the above examples? Nothing in the attribute list gives us this information. Thus, we devised the last component of the classified ad, the constraint, to provide the matching and allocation rules.

3.2 Matchmaking with Classified Ads

The fundamental goal of any resource management system is to make allocation decisions by making matches between customers and available resources. To do this, not only must it have extensive information about the characteristics of the resources, as demonstrated by the attribute list, but it also must have a method of expressing when a match is appropriate. The method of describing match criteria should be both simple for those cases when the customer has little concern over all of the characteristics of the resource they will be granted, but also powerful enough to provide exact specifications when necessary. Likewise, it should be possible to express complex policies about which resources can be granted to which jobs. In our classified ad scheme, these match criteria are specified via the constraint.

We defined the constraint as an arbitrary boolean expression describing the conditions under which an entity is willing to be matched with another. A match between a resource and a customer is made when the constraints of both entities are satisfied. The constraints are composed of logical operators using constant values and the names of attributes specified in the attribute lists of the entities' being considered for a match. Because of the free form nature of the attribute lists, evaluation of the constraints is not always straight-forward. One complication is the logical merging of the name-space of two attribute lists. Since the constraint can use attributes defined in either its own, or the match's classified, there is a possibility for a name to exist in both lists. To overcome this ambiguity, the names may be prefixed with "my." or "target." to specify from which attribute list to take the value. The "my" prefix indicates that

And	T	F	U	E	*Or*	T	F	U	E	*Not*	
T	T	F	U	E	T	T	T	T	E	T	F
F	F	F	F	E	F	T	F	U	E	F	T
U	U	F	U	E	U	T	U	U	E	U	U
E	E	E	E	E	E	E	E	E	E	E	E

Table 1: Truth tables used for evaluating constraints

the attribute must come from the same classified as the constraint being evaluated. Prefixing with "target" indicates the attribute value must come from the potential match's attribute list. If no prefix is used, but a naming conflict is encountered, the value is taken from the attribute list in the same classified as the constraint. We use this rule because we assume that in writing the constraint, the user was already aware of the attributes in the classified being used, but may not be aware of what is defined in another classified. Two additional complications occur when a name is used which is not defined in either attribute list, or when a type mismatch occurs during evaluation of the classified. Any sub-expression using a name which is not defined in either classified's attribute list is given the value *Undefined*. This undefined sub-expression cannot contribute to the evaluation of the entire constraint, but if the constraint's boolean value can be determined without the contribution of this sub-expression, it will be accepted. A type mismatch occurs when a comparison between two attributes or constants of differing types is encountered. When this occurs, the entire constraint is considered to be erroneous and therefore not satisfied. The truth tables for the logical "and", "or" and "not" operations which implement these rules is shown in table 1.

(1)	`(LoadAvg <= 0.3 && KeyboardIdle > 15 * 60)`
(2)	`(target.Owner == my.Owner &&` `my.VirtualMemory > target.ImageSize + 10000)`
(3)	`(ClockDay == 0 || ClockDay == 6 || ClockMin < 7 * 60)`

Figure 3: Sample Constraints on Workstations

We used boolean expressions to satisfy our desire to have a simple method of specifying allocation policies and requirements, but also to provide tremendous expressiveness. Customers and owners can write expressions which are as complex as their request or usage policy needs. Example constraints are shown in figures 3 and 4.

Figure 3 shows example constraints which might be placed on workstations. The first constraint specifies that a job can be run on the resource provided the processor load is less than 0.3, and that there has been no activity by any owner (either on the console or via a remote login, as expressed by the `KeyboardIdle` attribute) for more than 15 minutes. In this example, only the state of the resource itself determines its availability. This condition could be used in an opportunistic RMS where resources are available for allocation only when not in use by their owners. The second example is more specialized; any job submitted (owned) by the owner of the workstation is allowed to run as long as the machine has 10 megabytes more free swap space (`VirtualMemory`) than the job requires. This constraint is explicit in saying that the "Owner" attribute of the job's classified must match the owner attribute of the workstation's classified.

```
(1)  (Arch == ``sun4m'' && OpSys == ``SunOS4.1.3'')

(2)  (Arch == ``sun4m'' && OpSys == ``SunOS4.1.3'') &&
     (Memory >= 32 && Dedicated == TRUE)

(3)  (Machine == ``chestnut'' || Machine == ``cedar'')
```

Figure 4: Sample Job Constraints

Furthermore, the "ImageSize" attribute value comes from the classified of the potential matching job, but the attribute "VirtualMemory" is to come from the workstation's classified. This workstation owner is logically saying that any job he submits is free to run on his workstation as long as there is enough free swap space for the job plus an additional 10 megabytes for the owner's expansion. The third example demonstrates how a resource's availability policy can be built based on time of day. This expression states that the resource is considered to be available on Sunday (the 0th day of the week), Saturday (the 6th day of the week) or before 7am on any other day. The time values used are generated by the resource itself (and placed in its attribute list), so there are no time-zone concerns which may otherwise occur in a wide-area, centrally administered system.

The constraint placed on a job describes the sort of resources on which it can run. Examples of typical constraints on jobs are shown in figure 4. The first example provides the least information needed in order to make an allocation. It specifies only that it requires a machine of architecture type "sun4m" running release 4.1.3 of SunOS. The second example further states that, in addition to the previous requirements, a machine with at least 32 Megabytes of memory, and containing an attribute “Dedicated” set to true is needed. Note that in the workstation attribute list in figure 2, there was no “Dedicated” attribute. If no workstation in the pool has such an attribute, this job will never run because the sub-expression using the attribute “Dedicated” will always evaluate as undefined. The third example gives a very restrictive set of resources for this job. Only the machines named “chestnut” or “cedar” are acceptable. Here, the customer is limiting the job to one of only two workstations based solely on the name of the machines, and not any other characteristics. Presumably, the customer knows about some special feature of these machines, for example a tape drive, and is therefore requesting them. If, however, the desired tape drive is removed from one of the machines, the job is still eligible to run there, and may fail due to the missing drive. Likewise, if a new tape drive of the appropriate type is added to another machine, this job will never be able to use it because the job is constrained to run on either of the two specified machines. Therefore, we encourage customers to write constraints which describe the *characteristics* of the desired resources rather than requesting particular resources. In this way, as the characteristics of individual resources change, customers jobs adapt to running in the updated environment.

```
(1)  Vacate = (LoadAvg > 1.5 || KeyboardIdle < 5)

(2)  Vacate = (ClockMin > 8 * 60 && ClockMin < 17 * 60 &&
     target.Owner != ``joe'')

(3)  Vacate = (CurrentTime - EnteredCurrentState > 8 * 60 * 60)
```

Figure 5: Sample "Vacate" constraints.

3.2.1 Other types of "Constraints"

We developed constraints to provide a method of specifying how resources and customers are matched by the resource allocation component of the overall RMS. We have also found it beneficial to use the basic form of a constraint, a boolean expression, to trigger other actions on resources. These constraints can be maintained locally at the resource for evaluation when needed. For example, deciding upon a policy which governs when a resource must be released by a running job can be just as important as deciding when a resource is eligible for allocation. We use a new constraint, called *Vacate*, for this purpose. When the Vacate condition becomes true, any job running on a machine is forced away. Sample Vacate constraints are shown in figure 5. The first example demonstrates how Vacate can be used to insure absolute priority to resource owners. This constraint states that the machine must be vacated when the processor load on the machine goes above 1.5 (indicating CPU usage by the owner of the machine) or when a key has been touched within the last five seconds. The second constraint demonstrates how an owner might guard their machine. It states that all jobs must be vacated during normal work hours (between 8

AM and 5 PM) unless the job was submitted by "joe" (presumably the greedy workstation's owner). Note once again the use of the qualifier "target" to specify the source of the owner attribute. In this case, the target is the classified of the job running on the machine. The third constraint states that a job will be vacated after it has been running for eight hours (that is the difference between the time the machine entered its current, running state, and the time when the evaluation is done). We might use this constraint on an unowned machine to insure that no job dominates it for too long.

Classified ads, and the constraints within them, provide a tremendous amount of flexibility in the types of resources and customers that can be represented. Both the amount of information about each entity, and the requirements each entity places on a match are variable. Additionally, because we allow constraints to be set on a per resource basis, system administrators and resource owners can use any utilization policies they desire for a particular resource.

3.3 Requesting Multiple Resources for a Single Job

The classified model, as we have presented it thus far, provides a powerful method for allocating individual resources to customers. We have not, however, described a method for requesting multiple resources for a single job. A customer could request multiple resources in two separate dimensions. In one dimension, the customer could ask for a number of homogeneous resources which are bound by the same constraint. In the other dimension, a customer could require multiple, heterogeneous resources for which different constraints are needed. In both cases, we have found that multiple resource requests can be accomplished by augmenting the way the customer RM manages the job's classifieds. The basic structure of the classified itself need not be altered.

To support requests for multiple homogeneous resources, we require three attributes in each customer classified: `CurrentResources`, `MinResources`, and `MaxResources`. These attributes define the current number of resources allocated to a job, the minimum number with which the job can possibly run, and the maximum number it can potentially utilize. A job which is not currently running will always have `CurrentResources` set to zero, and a running job will always have `CurrentResources` between `MinResources` and `MaxResources`, inclusive. We require the customer's RM to insure that these conditions are met. When communicating with the resource allocator, the customer RM must continue to request resources for the job by resending its classified until either no more resources can be matched with it, or the `MaxResources` level is reached. If at least `MinResources` are received, the job can be started. By placing the burden on the customer's RM, we avoid modifying the global resource allocator to support multi-resource jobs. We therefore simplify the global resource allocator by keeping it focused on its single matchmaking responsibility.

For a customer to request a heterogeneous collection of resources, it must be able to specify a number of distinct constraints. To permit this, we have introduced the notion of a *cluster* of classifieds which consists of a collection of distinct classifieds all of which must be satisfied in order for the job to run. Once again, we place the responsibility for insuring that sufficient resources of each type are available on the customer RM.

3.4 Classified Ads in the Condor "Kernel"

To fully evaluate the utility of the classified ad model, we wanted to apply it to a RMS running in production mode. For this purpose, we have modified the Condor [3] RMS to use classifieds as its method of representing all entities under its control. We also use classifieds as the principle form of communication between components of Condor. By using a single structure, we simplified the methods of communication among the components of the system. Beginning with the next release, classifieds will be a standard part of Condor. Condor was specifically designed to run in an opportunistic environment with privately owned resources. Because of this, the resources' availability for use by Condor is never guaranteed. The portion of Condor which makes the heaviest use of classifieds ads is the *kernel* which provides services for discovering, requesting and allocating individual resources to customers. We call this the kernel because these services are the core upon which all other portions of Condor are built. The kernel is responsible for building and maintaining classified advertisements for every entity within the system, and for performing matchmaking based on

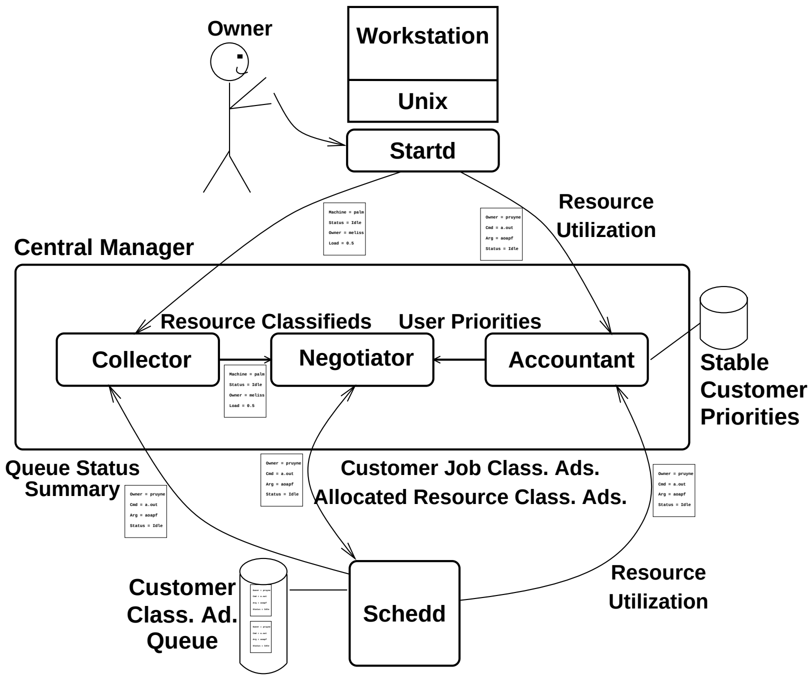

Figure 6: Structure of the Condor Kernel

a well defined allocation policy. The structure of the kernel is shown in figure 6, and the overall architecture of Condor, including the role of the kernel are presented in [36].

We implement the kernel as three distributed components: resources (represented by *Startds*), a *Central Manager*, and customers (represented by *Schedds*). The central manager supervises the entire system via its three daemons: the *Collector*, the *Accountant*, and the *Negotiator*. The collector is the repository for all of the classifieds generated in the system, and all entities which report to the same collector are considered to be part of the same Condor *pool*. Every entity within Condor periodically sends its classified to the collector. Each classified received by the collector is stored on a list corresponding to its `SelfType`. To view the state of a pool, a list of classifieds can be requested from the collector via a remote procedure call by providing the type of classifieds desired, and a constraint that must be satisfied by all of the returned classifieds.

The accountant process in the central manager implements the customer prioritization scheme for the system. It receives information about each resource utilization interval as a classified ad, and uses this information for determining priority among customers for future allocations. Condor uses the *Up-Down* [18] algorithm for assigning priorities to customers. The Up-Down algorithm penalizes customers who have made heavy usage of the system in favor of new customers who have received few resources in the past. By making the accountant a separate process, we can introduce new inter-customer scheduling policies, such as fair share [17], without modifying any other portion of the system. In order to maintain the customer priority information in case of a failure, we have the accountant store this information on stable, secondary storage.

The accountant provides the customers' priorities when requested by the third component of the central manager, the *negotiator*, which is responsible for making matches between customers and available resources. To perform these matches, the negotiator also retrieves classified ads from the collector for all of the resources in the system, as well as a classified ad representing each customer in the system which contains a non-empty job queue. With these two lists, the negotiator begins a *Negotiation Cycle* by contacting the customers' RMs in the priority order specified by the accountant, and requesting the classified for each job in its queue. As each job classified is received, the negotiator evaluates its constraint against each of the resources' classifieds, and the resources' constraints against the job classified. If a match exists, the customer is granted permission to use this resource, and the matching resource's classified is sent to the customer. When all customers' RMs have been contacted, the negotiation cycle is complete.

Outside the central manager, the *startd* (short for start daemon) is responsible for generating classified ads which describe the state of a resource. A startd runs on every resource within the Condor pool, and monitors its status. It gathers all the desired information about the resource, and places it into a classified which is sent to the collector. As part of its job composing the classified, it also evaluates any local constraints, such as the `Vacate` expression, and triggers actions accordingly.

A scheduler daemon (*schedd*) maintains the classified ads for each customer in a stable job queue. The classifieds stored in this queue contain all the information needed for matching a job with a resource as well as information needed to run a job. The schedd also generates a classified ad summarizing the state of the customer's queue and sends it to the collector. To maintain a stable job queue, a log of all updates to the classifieds is stored on disk. If, at any time, the schedd should go down, the state of all the job classifieds can be recreated by replaying this log. When contacted by the negotiator, the schedd responds with the customer's request classifieds. If the negotiator grants a resource, the schedd establishes an application RM process (for example the CARMI RM process described in the next chapter) to manage its run.

3.4.1 Flocking - Providing Gateways Between Kernels

To expand the reach of a RMS to as many resources and customers as possible, we wanted interconnect distinct resource pools to allow sharing to take place between them. In Condor, the ability to share resources across pools is referred to as *flocking* [37]. We believe flocking is desirable for a number of reasons. The most common is to join resources in separate administrative domains. By sharing via a flock, administrators need not agree on a single central manager and need not even agree on allocation policy. Flocking also allows these administrators to place policies on how they will share their resources across pools. Additionally, flocking increases the scalability of the overall system. This is essential because maintaining classifieds for every entity will become a limiting factor as the environment grows in size.

We have found that the classified ad structure provides a simple method of sharing resources by creating a flock. We accomplish flocking via classified ad *gateways* which are responsible for conveying information about free resources in one pool to the gateway located in another pool. To communicate, these gateways must first be able to locate one another. Classified ads again provide us a solution since each gateway can advertise its presence in its local pool. Its classified may also contain information about its policies and a summary of the state of the local pool. To connect across pools, one gateway needs only to locate the global resource allocator in another pool. From there it can locate the remote gateway and establish communication with it.

A gateway is able to determine which resources are free in the same way that the negotiator does: by requesting a list of all classified ads from the collector. The gateway can then select classified ads which are representative of the free resources in its pool to be sent to the gateway in the remote pool. Upon receiving these classifieds, the remote gateway can forward them on to the collector in its own pool, substituting its own name and network address for those found in the original resources' classifieds. Once these classifieds have been passed to the collector, they are eligible for allocation by the negotiator on the same basis as all the other resources in its local pool. When a remote resource is allocated to a particular schedd, it will use the gateway's network address, which it finds in the classified, to request access to the resource. The gateway simply forwards the request to the negotiator in the remote pool just as a schedd would. When the negotiator allocates a resource to the gateway, the gateway passes the classified of an actual resource back to the submitting schedd.

3.5 Summary

The heart of any resource management system is its matchmaking between customers and resources. To perform this function, the RMS requires a method of representing both resources and customers which is expressive and adaptable. To meet these needs, we developed the classified ad structure which permits arbitrary attributes of resources and customers to be specified, and arbitrary boolean constraints to be defined which establish the conditions under which matches can be made. This structure allows for both simple and complex resources and requests to be specified, and provides room for growth as more detailed information is gathered about the system. The basic structure of a classified also allows us to perform other functions such as breaking matches, and allocating multiple resources to a single job. We have implemented classified ads, and are using them in Condor for representing resources and customers as well as a structure for communication among Condor's components. Classifieds have also provided us with a straight forward method for sharing resources across pools via classified gateways.

Chapter 4

Fundamental Resource Managment Services

For all a RMS' ability to monitor and manipulate resources, it provides no benefit if applications cannot utilize these resources. For single process, sequential applications, utilization of the resource is usually as easy as running the application's executable file. More and more customers are finding, however, that in order to generate results within an acceptable time period they must employ several resources in parallel for use by a single application.

Unlike sequential applications which generally do not need to interact directly with the RMS, parallel applications can benefit greatly from this interaction. A parallel application can use information about resources allocated to it to make important algorithmic decisions, such as load balancing, which can greatly impact its performance. Furthermore, by continuing to communicate with the RMS, a parallel application may be able to grow and shrink as more and different resources become available. To perform well, therefore, parallel applications need specialized resource management services.

Unfortunately, the typical environment in which parallel programmers work does not provide these services. Most parallel programmers today write their programs using a Message Passing Environment (MPE) that provides services for communication among an application's processes via messages. These MPEs often provide little support for RM services, and we believe they should not provide RM services at all. Instead, we have defined an interface for communication between a MPE and a RMS. Keeping these systems separate has a number of advantages. First, a well defined interface means that adapting a new RMS to a MPE does not require any changes to the MPE code. This allows researchers to specialize on one portion of the problem without becoming an expert or involved with the implementation of the other part. In our case, this means we have been able to concentrate on RM issues and benefit from others' work on MPEs. Second, a RMS may customize or extend the set of RM functions defined by the MPE to suit the environment in which it runs. Finally, a single RMS does not need to be changed greatly to support new MPEs. Only the portion which utilizes the interface needs to be changed. This means that high quality RM services can quickly be provided to new MPEs as they are developed. In the next section of this chapter we describe our method of interfacing a MPE to a RMS as well as our approach to permitting communication between a running application and a RMS. The remainder of the chapter discusses CARMI, a set of RM services we developed for use by parallel applications. We describe CARMI's services, along with their implementation and experience using them.

4.1 Building a framework for Resource Management

Our first step in migrating resource management services out of a MPE was determining exactly which services will be handled by the RMS and which will continue to be handled by the MPE. From the perspective of the MPE, the resource manager is the decision maker in the system. Therefore, all service requests pertaining to resource allocation, task assignment, and queries about the state of the system must be in the domain of the resource manager. The MPE, on the other hand, is responsible for communication and other services which help insulate the application and the RMS from the underlying operating system and hardware. The MPE must be careful not to provide applications with access to services which circumvent the policies imposed by the RMS. For example, process creation is a desirable service for the MPE to provide, but this service should only be available to the RMS. By handling process creation, the MPE shields the RMS developer from the diverse process creation primitives supplied by various operating environments. Also, by creating the

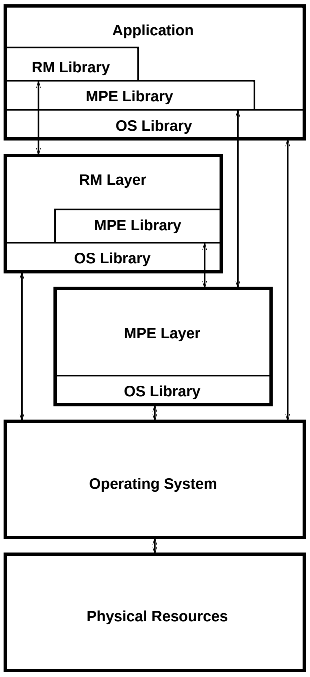

Figure 7: Layers of a system with distinct MPE and RM components

application processes, the MPE is able to initialize any needed communication paths. However, if application tasks are able to create processes directly they may defeat the efforts of the RMS to maintain load balancing. Figure 7 shows our layering of a combined RMS and MPE system.

To enforce the distinction between resource management services and communication services, we have split them into separate libraries which are linked into applications. The communication library contains only an interface to the communication primitives supported by the MPE. Communication among tasks comprising a parallel application is accomplished by making calls into this library. The RM library is layered on top of the MPE library because it uses the communication library to send requests for service to the RMS. Within our layered structure for building a RMS described in chapter 1, it is the role of the application RM process to receive these service requests and either handle them internally, or pass them along to the rest of the RMS. The application RM itself may also use the communication library to access services provided by the MPE for communicating with application tasks. All of these components access the physical hardware via the operating system.

4.1.1 Processing RM requests outside the MPE

Our layered structure gives well defined roles to the various components of the system, but we also require a method of interfacing between the components. In particular, by breaking a previously monolithic MPE into distinct communication and RM pieces, we must provide a method for the application to communicate with the RM layer to make service requests. The easiest way to do this is to implement the RM layer as a separate process which can use the communication primitives provided by the MPE. Again, the application RM process serves this purpose. In this way, the RMS can appear to the application as another process within the communication domain of the MPE. Application RM requests are made by sending a message to this process. Figure 8 shows a generic resource management request exchange between an application task and the RM process using the MPE for communication.

Although the figure does not explicitly show it, our model allows for RM

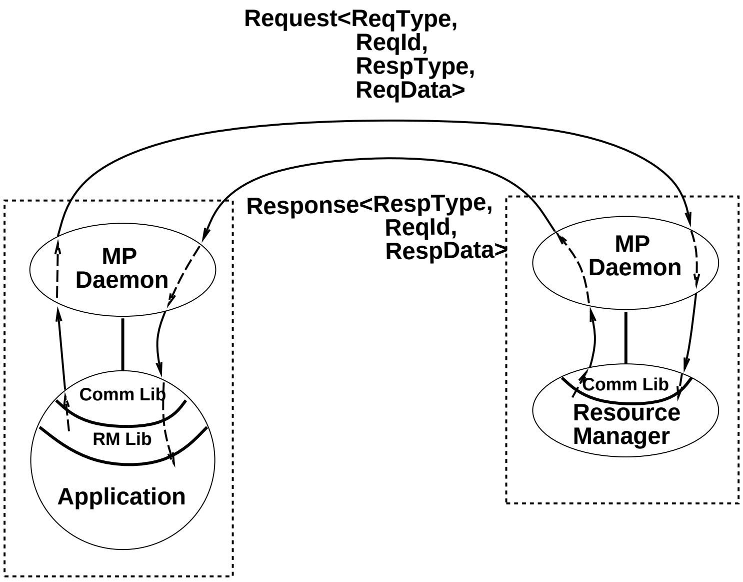

Figure 8: Flow of a RM service request between an application process and a RM servicing process

service requests to be handled asynchronously. That is, when a process makes a request for a RM service, it need not block waiting until the request is fulfilled. Making RM requests asynchronous has a number of advantages. First, the time required to fulfill a RM request may be unbounded. For example, a request to allocate a resource may take a very long time if all of the appropriate resources are in use. Blocking the application for this length of time is unacceptable. Second, asynchronous handling of requests allows for more parallelism by permitting a task to have more than one RM request outstanding. This is especially advantageous when a single request triggers a series of operations such as creating a group of new processes. Instead of using a stream of individual process creation requests, an application can make a number of requests without waiting for a response. When the set of requests has been made, the application can then wait for each of the responses if it desires. Since these are basically independent requests, the RMS may handle them in parallel without worrying about dependencies among the requests. This approach can reduce the overall time required to start all of the processes.

Figure 8 also shows the format for general RM request and response messages. The request message contains not only the type of the request and data associated with the request, but also a request identifier and a response type field. The request identifier field is the key to asynchronous handling of RM requests. The RM library locally generates a unique request identifier for each request made by an application task, and returns its value to the caller immediately. The request identifier is also passed to the RM process as part of the request message. When the resource manager composes a response to a service request, it includes the request identifier. The application task then uses the request identifier field in the response message to match the response with the corresponding outstanding request. By using unique request identifiers, a single application task can differentiate between responses for any number of outstanding requests, even if the requests are for the same type of service. The request message also contains a field specifying the desired message type to be used by the RMS when sending a response to this request. Allowing the programmer to specify the return message type provides assurance that the RM service handling library will not interfere with normal communication among the application processes.

4.1.2 Interfacing PVM to a RMS

To demonstrate the utility of our MPE to RMS interfacing framework, we wanted to integrate this interface with an existing MPE. For our initial implementation, we selected PVM [23, 4]. We elected to start with PVM for a number of reasons. First, the PVM source code is freely available which made implementation of the interface possible. Second, PVM is widely used, and we hope that our work will benefit as many users as possible. Third, unlike many other message passing environments, the PVM communication infrastructure will handle dynamic addition and deletion of resources and processes. Finally, PVM provides a comparatively rich resource management application programming interface. This provides us with a basic set of operations to implement within the the new framework rather than having to define new primitives before gaining experience with the system. We worked with the PVM development team at the University of Tennessee while developing the interface, and they have included it in the standard release of PVM since version 3.3.

Prior to our separation of RM functionality from PVM, all requests for RM services were handled by a combination of the PVM library that is linked into every application process and the PVM daemons that run one per every workstation utilized by the application. Applications made requests by calling a function in the PVM library which translated the request into a message to

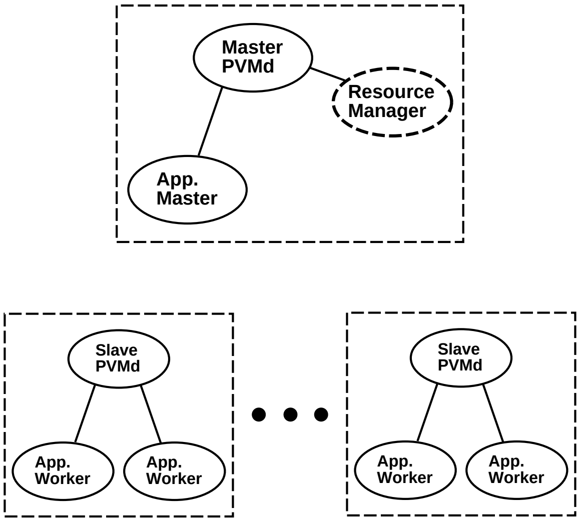

Figure 9: A PVM system with a single RM process

the PVM daemon on the same host. The PVM daemon handled the request locally when possible (for example a process creation request where the new process will be started on the same host), or forwarded the request to a remote PVM daemon that it believed would be able to handle the request best. The remote PVM daemon then carried out the request and sent a reply back to the daemon local to the requesting task. The local daemon completed the loop by forwarding the the response to the task that initially made the request.

To move the handling of resource management requests outside the PVM daemons, our first step was simply to introduce the notion of resource manager tasks that identify themselves via the new call `pvm_reg_rm()`. Each PVM daemon associates itself with a single RM task. This RM task need not be on the same host as the daemon since communication between the RM task and the PVM daemon is accomplished with the PVM message passing facility. Figure

9 shows the most basic RM configuration possible in which there is only one RM task in the system that is responsible for all of the hosts and application processes.

Within our framework, PVM daemons will perform some services for RM tasks that they will not perform for any other tasks within the system. The most important service that is restricted to a RM task is process creation and monitoring. When a RM task chooses a host on which to run a new process, it sends a message directly to the PVM daemon on that host. The daemon responds by starting the process, and reporting the status of the new process back to the RM task. When a process exits, the PVM daemon sends a message to the RM task containing the exit status and the processor usage of the process. This scheme helps us to insulate the RM task from the underlying operating system. No matter what type of resource is used, the RM task can start a process and receive status information simply by sending and receiving messages. As more and more types of resources (such as massively parallel machines) come to be supported by PVM, this service becomes increasingly useful to us for keeping RMS implementations simple. When the PVM daemon initializes an application process, it also informs it of the RM task assigned to it. Any RM service request made by this process will be sent to the assigned RM task.

Incorporation of our framework for externalizing RM service handling has been received well by the research and development community. In addition to CARMI, other RM service providers have developed PVM resource manager processes to provide interfaces to their system. Examples include the commercial resource management packages LoadLeveler [14] from IBM and the Load Sharing Facility (LSF) [11] from Platform Computing. Unlike CARMI, which we describe in the next section, neither of these packages attempts to extend the set of RM services available to their applications. They simply support the existing PVM defined services.

4.2 The CARMI programming environment

After modifying PVM to allow resource management to be handled externally, we proceeded to use the interface to run PVM programs with an existing RMS, Condor. All resource allocation, and process creation decisions required by an application were taken over by Condor. When the basic interface was put in place, it quickly became apparent to us that the Application Programing Interface (API) defined by PVM was not sufficient for the opportunistic environment provided by Condor.

We viewed the fundamental shortcoming of the PVM API to be its synchronous nature. Although our RM framework calls for RM service requests to be made asynchronously, PVM provides only a blocking, procedure call interface. Under Condor, this was clearly not acceptable because some calls, particularly those involving resource allocation, may take an unbounded amount of time.

Furthermore, because of Condor's opportunism, we wanted parallel applications to be able to adapt to changes in the number of resources available to them while they are running. PVM's use of machine names and its limited ability to provide notification when resources come and go was not sufficient in this environment. Finally, PVM provided only limited information about the resources on which an application is running. Often, parallel applications can use detailed information about resources in order to make load balancing or other decisions. Condor, on the other hand, maintains a great deal of information about the resources it controls in its classified ads. We wanted to make this information available to applications at run-time.

To better support parallel applications running under Condor, we have developed the Condor Application Resource Management Interface (CARMI) [38, 39]. CARMI provides a set of RM services that are well suited to the opportunistic environment provided by Condor. In developing CARMI, we still rely on the PVM primitives for application communication, and we support the PVM resource management API to ease porting of existing PVM applications to CARMI.

4.2.1 CARMI Services

As suggested by our basic framework for interfacing a MPE to a RMS, CARMI presents an asynchronous API in which services are requested via procedure calls that return a service request identifier. The return of the request identifier indicates that a request has been registered with CARMI. When the request has been serviced, the application receives notification via a PVM message. This notification carries the same request identifier so that the application can easily match the notification with the corresponding request. Every request completed notification also contains a status value indicating the success or cause of failure for the request. Since the notification comes in a normal PVM message, application tasks need only use a single interface for handling communication with other tasks, and discovering the completion of RM services. Using a separate notification mechanism for completion of RM service requests would have required the application to poll continuously for each sort of event separately.

CARMI's procedure call interface is familiar to application programmers, so they should be comfortable using it. The asynchronous completion, though, is unique and we believe it has a number of benefits described previously. The services provided by CARMI fall into three categories: request management, resource management, and task management. We describe the full CARMI API, along with other information about using CARMI, in appendix A.

Request Management

The request management portion of the CARMI API consists of two functions, one to request the state of a previous request, and one to cancel a request. The request state function allows an application to determine whether a particular request is still outstanding. The cancel function simply requests that a previous request be ignored. The side affects of cancelling a request depend upon the type of request being cancelled.

Resource Management

Because CARMI applications are allocated resources by Condor, we must provide them with a method of requesting resources which is consistent with Condor. As we discussed in the last chapter, Condor's resource and request representation model is classified ads. We must therefore provide services to applications which permit them to manipulate classifieds. As a first step in doing this, we introduce the notion of a resource class. All resources within the same class come as a result of requests made in a single classified ad. Therefore, all of the resources within a class must satisfy a single constraint. Resource classes are defined either at the time the job is submitted, or at run-time via the function `carmi_new_class()`. Applications may request new resources at run-time using the function `carmi_addhosts()` which takes as arguments the name of the class and number of new resources desired.

The classifieds used to describe resources contain a great deal of information which may be useful to a running application. We therefore provide a function for retrieving the full classified for any resource allocated to an application. The classified is returned as a C++ object which contains methods for querying the names of all of the attributes in the classified's attribute list, and for retrieving the value associated with any attribute. Applications may use this information in any way they see fit to enhance their performance.

Finally, since CARMI runs in a dynamic (and often opportunistic) environment, we provide services for receiving notification when resources are lost, suspended or resumed. Suspension occurs in CARMI when an owner first returns to a machine. Instead of immediately evacuating the machine, Condor first suspends all application processes running there. This is done in hope that the owner will only be active for a short time, and the processes may be resumed without having to be aborted. If the owner remains active, the resource must be vacated which forces termination of any CARMI processes running there and application processes that have requested notification will be notified.

Task Management

The last type of service we provide in CARMI allows applications to make use of resources they have been allocated by creating processes using the `carmi_spawn()` function. A CARMI process creation request can specify either a resource class or a particular resource on which a process should be created. Any executable file that is not available on a resource where the process will be executed is automatically transfered to the target resource. When a process exits, a notification message will be sent to the process' parent giving the exit status of the process.

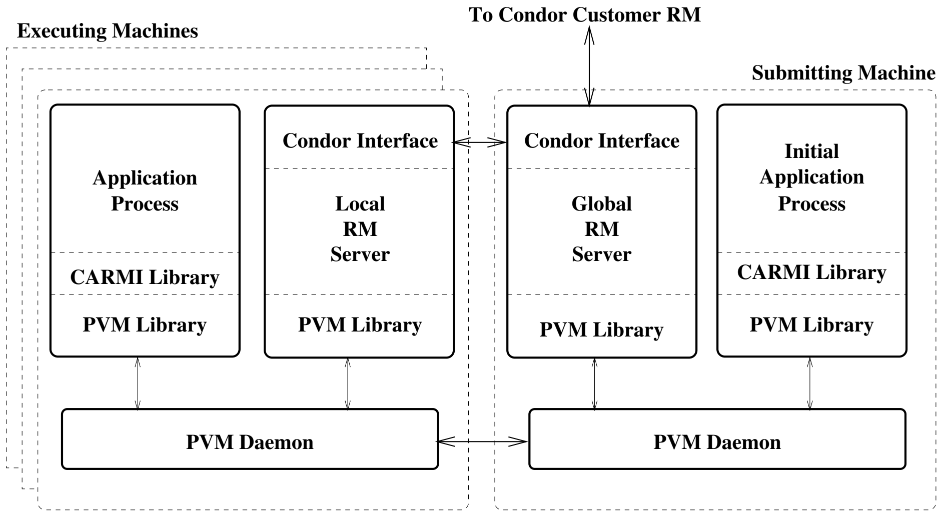

Figure 10: CARMI System Architecture

4.2.2 CARMI Implementation

In figure 10, we show the architecture of the CARMI implementation. CARMI takes advantage of the modular run-time environment provided by Condor. When Condor determines that sufficient resources are available to start a job, it creates an application RM process on the machine where the job was submitted. Condor uses a different application RM process for each job *universe*. CARMI is one of the Condor job universes, so the CARMI application RM process will be started by the Condor customer RM process. Because this application RM process runs throughout the life of the application and coordinates all the resources provided to the application, we refer to it as the "global" RM process for the application. The customer RM also provides the application RM with the names of the resources allocated to the job.

When the CARMI application RM process starts, it first contacts the access control component of each resource to claim the resource, and requests that a local application RM process be started on the machine. We consider this local application RM to be an extension of the application RM process for this resource. The local application RM process is responsible for monitoring the application processes running on its resource. The first process the local application RM runs is the PVM daemon, and it then registers itself as the RM process for that daemon. When the PVM daemon has been started on all resources allocated to an application, the global application RM process starts it running.

The global application RM services all of an application's RM requests and, after servicing the request, responds to the process from which the request originated. Because the local RM process has also been registered as a PVM RM process, it will receive all RM service requests made by local processes. When the local RM receives such a request, it forwards it to the global RM via a PVM message formatted the same as the message the CARMI library generates for this type of request. To the global RM this looks the same as any other service request, so when it is completed, the response will be sent back to the local RM. When a local RM receives a response, it simply forwards the response on to the local process that initially made the request.

Any type of resource allocation requires interaction with the Condor kernel. The global RM, therefore, handles all run-time allocation requests by updating the classified ad for the resource class in which the application has made a

Elapsed Time	Total Occupancy	Accumulated CPU Time	Spawn Occupancy	Suspend Time	Total Hosts
9:28:59	528:40:32	500:43:35	6:24:59	15:03:03	161

Table 2: Resource utilization by a synthetic CARMI application

request. Once the classified has been updated, it becomes the responsibility of the customer RM to negotiate with the kernel to allocate a new resource for the application.

4.2.3 Experience with CARMI

During the development of CARMI, we also developed a synthetic application that attempts to acquire as many machines as possible while competing with owners and other jobs submitted to Condor. Our goal in developing this application was to test CARMI in a production environment as well as to show its power in providing compute cycles to real applications. For every resource that is allocated to this application, a worker process that repeatedly loops for a number of iterations specified by the initial or master application process is spawned there. When the loop iteration count is completed, the worker reports its result along with the amount of CPU time it consumed, and then goes back to looping. Table 2 shows the results of one long execution of this application. From this table, we see that the job was able to accumulate nearly 53 times as much CPU time as elapsed wall clock time. The last three columns of the table require some explanation. The only RM service that we did not handle asynchronously for this run was creation new processes. We did this to show the potential value of handling requests asynchronously as is normally done in CARMI. While a spawn request is outstanding, the application master process blocks waiting for the result of the request. The spawn occupancy column reports the product of the total occupancy on all machines while it was blocked. This gives a worst case amount of CPU time lost due to the blocking. As mentioned previously, Condor first suspends hosts, for up to ten minutes, before removing them from a computation. In the "suspended" time column we show that hosts allocated to the computation spent slightly more than 15 hours in the suspended state. This column was computed by the application itself as it received notification from CARMI about these suspensions and resumptions. Accounting for the time lost due to suspends, the job was able to utilize 97% of the CPU time allocated to it. Finally, in the last column we show that over the course of the run, the job was scheduled on 161 different hosts. This number corresponds to the number of machines that were reclaimed by owners during the run. An important result that table 2 does not show is that when a machine is lost, a replacement was typically obtained from Condor and ready to begin computation less than 30 seconds later.

For this run, we requested resources in two classes: older, slower MIPS based machines and relatively newer and faster machines with Sparc processors. On this application, the Sparcs are a little more than two times faster than the MIPS machines. Unfortunately at the time this experiment was run, we had fewer Sparc machines in the Condor resource pool, and the demand for them was

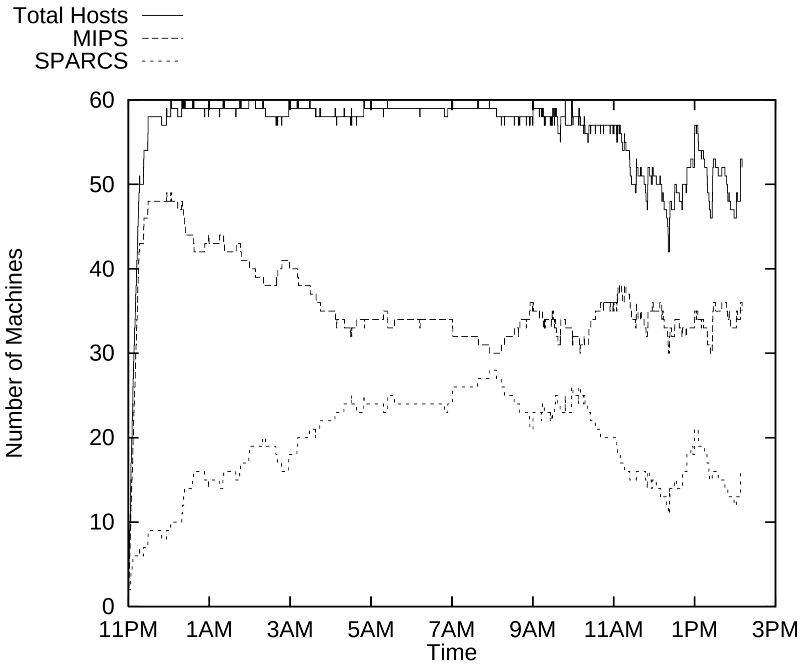

Figure 11: Host Occupancy

much greater than for the MIPS machines. To utilize as many Sparc machines as possible, the application removes a MIPS machine to allow itself to request a new Sparc machine when a software imposed upper limit of sixty machines is reached. If a Sparc machine is reclaimed by an owner, we replace it with the first available machine in either class. Figure 11 shows the result of this scheme. Initially, the application is able to rapidly acquire MIPS machines until the limit of sixty total machines is reached. At this point, it begins to remove MIPS machines in favor of Sparcs which are acquired much more slowly. When owners start arriving in the morning, Sparcs were re-claimed, but there were also no MIPS machines available to bring us back up to our desired level of sixty. This run demonstrates both the large amount of CPU time available to CARMI applications, but also the flexibility CARMI provides applications in manipulating resources to achieve the number and type of resources they desire.

Other installations of CARMI

One of the stated goals of our efforts has been to build systems that are usable by other scientists to achieve their research goals. In support of this goal, we have made CARMI freely available to other researchers. We are aware of two other installations of CARMI outside of our own department. The first is at the University of Durham in England. There they used CARMI to develop an application for "fitting of molecular potential energy surfaces to experimental spectroscopic data [40]." This is a computationally intensive task that was carried out on a pool of as many as 80 Hewlett-Packard workstations running CARMI. CARMI has also been used as a foundation for higher level parallel programming systems as part of the Dome [41] project at Carnegie-Mellon University. The Dome group is studying tools that support both malleable and fault tolerant parallel programming. In the process of working with these groups, we've received a great deal of feedback on CARMI, and it has allowed us to make it a more useful tool.

4.3 Summary

Applications must be provided with services which make it easy for them to utilize resources under control of an RMS. This is particularly essential for parallel applications which attempt to exploit a number of resources concurrently.

To provide these services, we first developed a method of interfacing an existing MPE with a RMS. This allowed us to benefit from others' work, and allow application programmers to start with a programming model with which they are familiar. We implemented this interface for PVM, and it has become a standard part of PVM. We next developed the CARMI programming interface, which utilizes the MPE to RMS interface to make resources managed by Condor available to parallel applications. We have demonstrated CARMI's effectiveness with synthetic applications and on production applications developed at other sites.

Chapter 5

Higher Level Programming Models

While CARMI provides a powerful set of services for managing resources within an application, it is sometimes burdensome for a parallel application to deal with individual resources. The complexity of managing resources within an application becomes even greater when those resources are dynamic such as in an opportunistic environment like Condor. On the other hand, careful management of resources by applications can lead to improved performance. For example, the synthetic application described in the previous chapter was able to gather a large number of resources, and move to more desirable resources as they became available.

To provide the benefits of extensive management of resources without burdening the application programmer, we have developed an *application framework*. An application framework presents a higher level programming interface which closely matches a particular model of parallelism. Internally, the framework uses CARMI's services to achieve efficient resource utilization. The framework not only simplifies the job of writing a parallel program, it can also lead to a more efficient application because a new parallel programmer automatically benefits from the experience of the framework developer.

Our first framework was intended to allow easy development of parallel programs in the opportunistic setting which Condor provides. To do this, the framework must be able to allow an application to automatically adapt to losing and gaining resources while it runs. One approach to parallelism which works well in this sort of environment and is also applicable to a large number of problems is *master-workers*. In a master-workers application, there is a single master process which generates work steps to be computed, and a collection of worker processes. Each worker process receives a work step from the master, computes a result, and sends the result back to the master. This process continues until all of the work steps have been completed. Master-workers parallelism works well in a dynamic environment because when a new resource becomes available, a worker process can be started there, and a work step given to it. If a resource is lost, the master can give the work step which was being computed there to the next available worker.

Because master-workers is such a common approach to parallelism, we have built an application framework to simplify writing this type of application. We call this framework the Work Distributor (WoDi). The goal of WoDi is to make writing master-workers applications for a dynamic environment very easy, and to relieve the application writer of the burden of managing individual resources. It is the responsibility of WoDi to monitor the status of all the resources allocated to a job, and insure that the results of each work step are returned to the master exactly once. In some respects, WoDi is similar to Piranha [32] which also helps master-workers applications adapt to dynamic resources. Our development of WoDi, though, has a few advantages over Piranha. First, Piranha is restricted to working with the Linda [30] tuple-space based communication environment where as WoDi is intended to work in a general message passing environment. Second, Piranha relies on a dedicated resource monitoring system. WoDi, on the other hand, is able to utilize any resource in a Condor pool via its calls to CARMI. Therefore, WoDi has access to a larger number of resources, potentially spanning a wide-area network. Finally, we have enabled WoDi to monitor the history of both the resources it is using and the work it is distributing to make intelligent work assignments. As we gained more experience with the system, WoDi improved the performance of its customers' applications. The services provided by WoDi, as well as a sample WoDi master program are described in the next section. Next, we discuss the implementation of WoDi, and how it has been used to generate visualizations of customers' applications. This chapter closes with a description of applications which have been run with WoDi, including a detailed account of our experience with a materials science application which was adapted to utilize WoDi.

5.1 WoDi Services

We present all of WoDi's features to the programmer as a single API. This API provides all of the services needed to write a master-workers style application, so a WoDi programmer will usually never use CARMI directly. The WoDi server process, which is automatically started when the programmer initializes their WoDi program, makes all of the CARMI calls needed to keep the application running.

We require the applications's master process to first initialize WoDi by informing it of the resource classes to be used, and the desired number of resources in each class. The master also provides the name of the worker executable file, and a set of PVM message buffers which should be sent to each new worker for initialization purposes. WoDi strives to maintain the desired number of resources in each class. When a new resource is allocated, WoDi starts a worker process, and sends the initialization messages to the new worker.

After WoDi has been initialized, the master normally would go into a loop of generating work steps, and receiving results until all of the pieces of work to be done have been completed. The master can generate work steps at any time, and pass them on to WoDi. Any time there are more work steps outstanding than there are workers to compute them, WoDi simply buffers them. To define a work step, the master process packs a message defining the work to be done into a PVM buffer, and hands this buffer over to WoDi using the `wodi_sendwork()` function. WoDi in turn sends this message on to an idle worker process. When the worker completes the work step, it sends a result message back to WoDi using the function `wodi_sendresult()`. WoDi records that the work step has been completed, collects statistics pertaining to the length of time required to complete the step, and forwards the result on to the master. If a worker process should fail before completing its assigned work step, WoDi will re-send that work step to the next idle worker. In this way, WoDi guarantees that the result of every work step will be received by the master exactly once.

In some master-workers applications, work steps come in groups in which all of the results from one group must be calculated before the next group can be started. We refer to this grouping abstraction as a *work cycle.* Often the characteristics of work steps (such as the amount of CPU time required to compute a work step) are relatively consistent between cycles. That is, the each step within a cycle takes nearly the same amount of time to compute each cycle. WoDi applications may specify the beginning and ending of a work cycle using the functions `wodi_begin_cycle()` and `wodi_end_cycle()`. When cycles are used, WoDi maintains a history of the computation times of all the work steps within a cycle. This work step history can be used in a variety of ways to improve the performance of the application.

One use of the work history is ordering the distribution of work within a cycle. Long running work steps will be sent to workers as early as possible to try to avoid having workers waiting for a single long running step to complete before the next cycle can start.

WoDi also uses the work step history when deciding how to assign work steps to worker processes. Long running work steps are sent to the fastest available resources. The speed of a resource is determined in one of three ways. First, the application master process can assign a speed to an entire resource class. All resources within a class are considered to be equivalent. Second, WoDi can use CARMI services to get the classified ad for each resource assigned to it. These classified ads contain the results of processor performance benchmarks run by Condor. Finally, WoDi can send a benchmarking work step to each new worker process after it is created. Because the same step is sent to each new worker, its performance on that step can be used as an estimate of the performance for the new machine. The benchmarking approach provides the most accurate indication of a resource's performance because it is measured on the type of computation which the application performs. Benchmarking also requires processor time on the new resource which does not contribute to the completion of the job, so when the default performance data available from CARMI is sufficiently accurate to order the resources benchmarking is not needed.

A final use of the work history is for determining desirable resource levels.

In general, more resources provide better response time for the application, but at higher cost in terms of allocated compute cycles. Beyond a certain level, additional resources cannot reduce response time because work steps cannot be sub-divided. For example, there is no need to allocate twenty workstations to an application that never has more than ten work steps outstanding at one time. This also means that response time for a cycle cannot be smaller than the time required to compute the longest step. We take advantage of this information in WoDi by running a heuristic which tries to determine the fewest resources needed for a cycle to be completed in the minimum possible time. This goal is achieved when there are enough resources to complete all steps except for the longest step in the same amount of time it takes for the longest step to be computed. The output of this heuristic can be used in place of the programmer's initial request for a desirable resource level if the application requests WoDi to do so.

Figure 12 is an example of a master program which uses WoDi. The program starts by packing a PVM buffer which is used to initialize each worker when it is started. It then initializes WoDi, giving it the initialization buffer as well as the number of resource classes and desired resource levels, and the name of the worker executable file. The program then loops through all of the needed work cycles. At the start of each cycle, initialization-data specific to that cycle is placed in a PVM buffer, and this buffer is passed to WoDi to be sent to each worker. Next, the work steps for this cycle are sent to WoDi. WoDi

```
main()
{
  buf = pack_initialization_data();
  num_classes = 2;
  class_needs[0] = 10;
  class_needs[1] = 5;
  wodi_init(buf, num_classes, class_needs, ``a.out''
            WORK_TAG, RESP_TAG);

  for (cycle = 1; cycle <= CYCLES; cycle++) {
    buf = pack_cycle_initialization_data();
    wodi_begin_cycle(cycle, buf, STEPS);

    for (step = 1; step <= STEPS; step++) {
      buf = pack_workstep();
      wodi_sendwork(step, buf);
    }

    for (step = 1; step <= STEPS; step++) {
      wodi_recvresult();
      result = unpack_result();
      process_result(result);
    }

    buf = pack_end_of_cycle_info();
    wodi_end_cycle(cycle, buf);
  }

  wodi_complete();
}
```

Figure 12: Sample WoDi master program

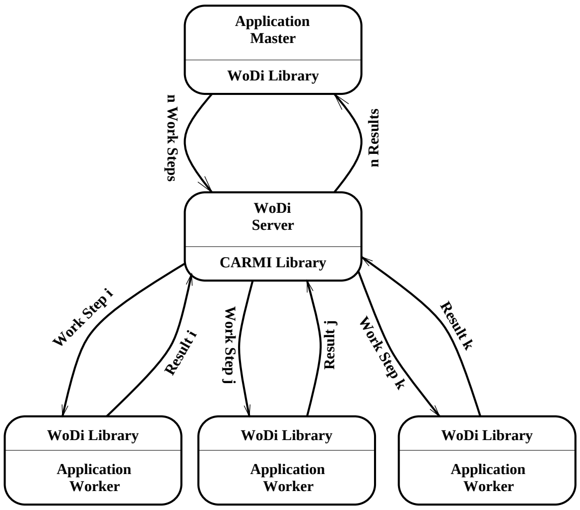

Figure 13: WoDi Architecture

distributes these to workers, and the master simply loops to collect all of the result messages. When all the results are received, the end of the cycle is signalled, and finally the completion of the application is indicated via the function `wodi_complete()` when all cycles have been completed. Although this application is simple, by utilizing WoDi it can run on any number of resources, and will be robust in the face of failures either due to resources being reclaimed by their owners, or do to actual hardware failures. Furthermore, because it is using cycles, it automatically benefits from WoDi's ability to analyze the work steps and re-order the steps to provide efficient utilization of the resources.

5.2 WoDi Implementation

Figure 13 shows the logical structure of a WoDi program. Like CARMI, WoDi is implemented as a combination of a library and a server process. The library passes work steps and other service requests (such as begin and end cycle requests) to the server using PVM messages. The WoDi server in turn passes work steps on to worker processes, and forwards their results back to the master. Because WoDi handles all resource management concerns for the application, only the WoDi server must be linked with the CARMI library. Application processes do not make any direct use of CARMI services.

We have implemented WoDi in a server process, as opposed to entirely within a library, to increase the amount of parallelism in the system. The application master can do any needed processing on results as they arrive without being concerned with having workers block while waiting for another piece of work. The WoDi server does very little processing of results, so it uses very little CPU time and is virtually always ready to send more work to a worker when it becomes free. Another advantage of this approach is that a single WoDi server may be able to service more than one master. This could increase resource utilization because while one application may be in a phase where few work steps are generated, another application might be generating a lot of work. With multiple independent WoDi servers, resources would have to be released by one and allocated to another application by the global resource allocator to adapt to this situation. With one shared server no resource release and re-acquisition needs to be done between applications, so overall resource utilization is higher.

5.3 Visualizing runs with WoDi

Visualization is one of the most popular and powerful methods of understanding the behavior of parallel programs. A number of general visualization tools for parallel programs have been developed for the purposes of debugging and performance tuning. These include UpShot [42], ParaGraph [43] and ParaDyn [44]. While these tools are powerful, and can provide good insight into the behavior of a parallel program, the are designed to be general in nature. That is, the statistics they gather and the visualizations they generate are for generic parallel programs, and are not customized to the application or application domain on which they are working. This leads to a semantic gap between what the tools generate, and what the application programmer is doing.

The opposite extreme is for every parallel application programmer to also be a tool programmer and generate visualizations particular to their application. This approach provides the best possible feedback to the programmers, but puts an unreasonable burden on the them. Generic data visualization tools, such as DeVise [45], provide an alternative, but the programmer is still required to lean how to generate input for the tool.

At the application framework level, we once again find ourself in a position

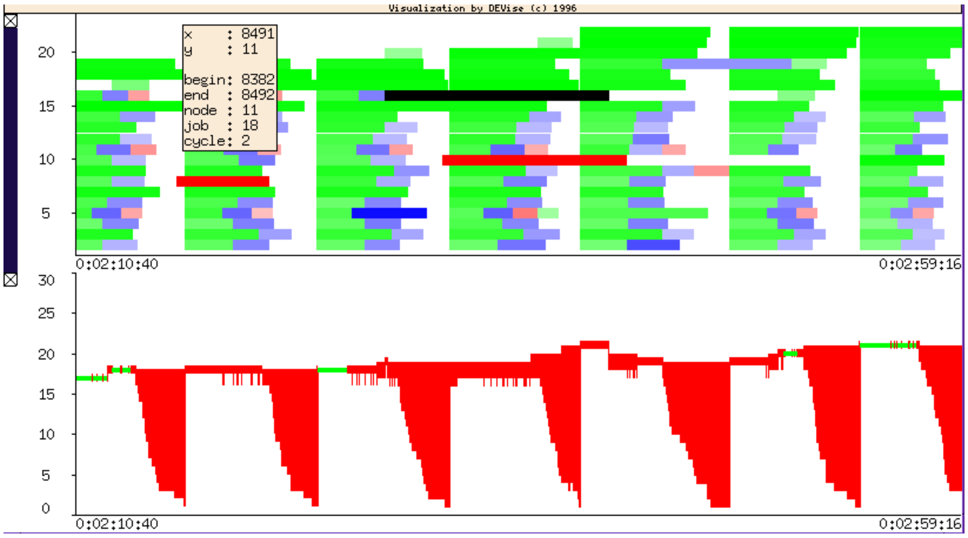

Figure 14: Visualizations of a WoDi run using DeVise

of knowing a good deal about the application we are running, so we can provide a higher level abstractions to the customer. With this in mind, WoDi generates a number of trace files which can be visualized with DeVise. To visualize these traces we require the customer to know only how to use the DeVise user interface. They do not need to learn how to define a visualization. Example DeVise visualizations, derived from WoDi output, are shown in figure 14. In this figure, the top graph shows the computation of each work step. Each row represents a processor, and each shaded box represents the computation of one work step. Time moves along the horizontal axis. In this graph, we are able to see exactly how the processors are being utilized, and can get further information on a step by clicking on it as shown in the upper left hand corner. By using WoDi and DeVise together, we get visual feedback using language like "node," "job," and "cycle" which is specific to the WoDi environment, and provides more meaning to a WoDi customer than general parallel programming terms such as "send" or "receive." We're also able to see CARMI specific events such as processor loss, as indicated by the large black bar in the upper middle, which are not generally considered by other parallel program visualization tools. White space also clearly shows wasted processor time. The lower diagram shows the number of processors available to the application as the top of the bar, and the bottom is the number of resources working. Therefore, when the bar is narrow resources are being well utilized, where lots of shaded area appears resources are being wasted. By using these tools together, we can provide customers with program visualizations at the same level of abstraction and with the same terminology used in developing their programs with WoDi. This greatly aids them in understanding the behavior of their programs for debugging or performance tuning purposes since they do not have to translate WoDi abstractions into lower level abstractions such as message sends and receives.

5.4 Experience with WoDi

Using WoDi, we have run a variety of parallel applications from a number of problem domains. These include graphics rendering, parallel compilations, processing of database queries, and studying the properties of materials. In this section, we provide a detailed description of our experience with a first principles based materials science application [46] developed at Oak Ridge National Lab.

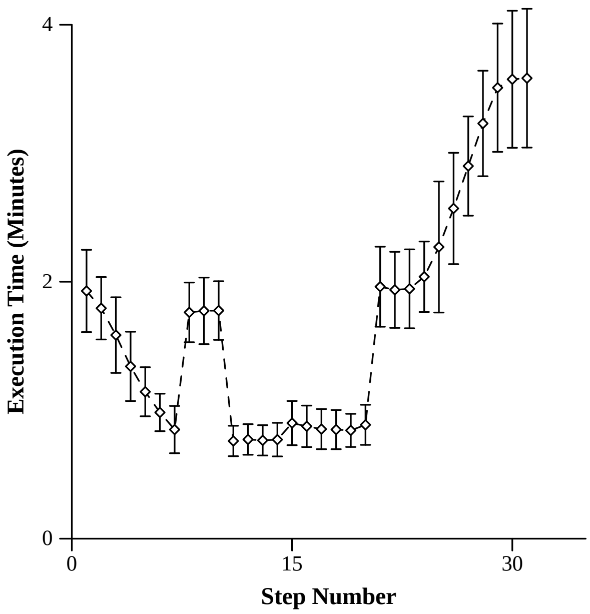

Figure 15: Average time and standard deviation by step number

This application was originally written using stand alone PVM and a master-workers approach to parallelism, and it demonstrates many of WoDi's features. Because it was written using PVM, it assumed a constant allocation of resources. However, by simply replacing the PVM communication primitives with WoDi send and receive work calls, the application was automatically able to run in a dynamic environment. Additionally, as we will see, the use of WoDi's heuristics greatly improved the performance of the application.

Logically, this application consists of two nested loops in which all work for the inner loop must be completed before any work for the next iteration of the outer loop can be started. This structure matches a WoDi cycle where all steps within one iteration of the inner loop make up a cycle. The data set used for these runs consisted of 31 steps per cycle, and a total of 35 cycles were executed per run. Figure 15 shows the average and standard deviation for each of the 31 steps across all cycles for one particular material. These steps vary greatly in the length of time they require to compute, but each step has relatively small variance across cycles.

5.4.1 Performance Objectives

The principle performance measures we use when evaluating WoDi are *execution time*, *occupancy* and *efficiency*. We measure execution time from when the job is first scheduled on a resource until the last step of the last cycle has been completed. Queueing time is not included, but it is usually low because a WoDi application can begin as soon as a single resource becomes available to it. Occupancy can be viewed as the cost of running an application if we assume that customers are charged for the time resources are allocated to them. Efficiency measures the fraction of the allocated processor time actually used. An efficiency of one means that all allocated processor time was used by the application. Efficiency therefore tells how well our investment in resources is being used. Two sources of efficiency loss are the required synchronization at the end of a cycle, and latency in communication.

In general, there is a tradeoff between efficiency and execution time as more resources are allocated to an application. As more resources are used, execution time decreases, but efficiency also tends to decrease because any time blocked at a synchronization point forces a larger number of resources to idle. Figures 16

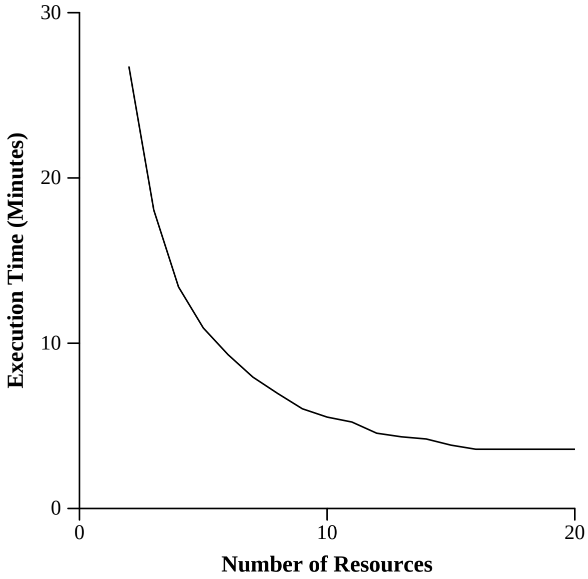

Figure 16: Execution Time for one work “cycle”

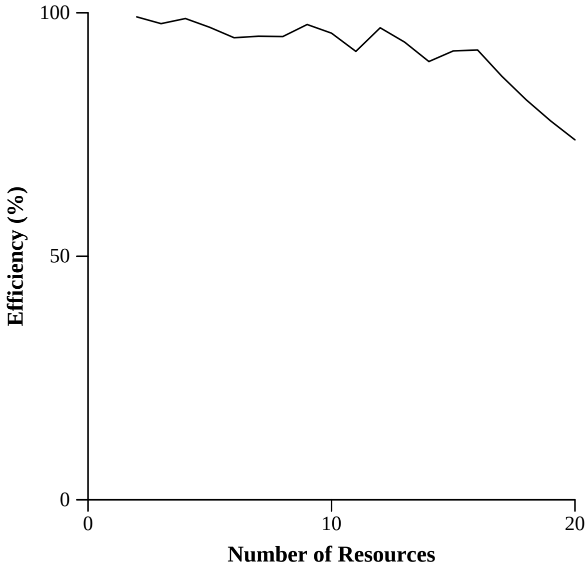

Figure 17: Efficiency for one work “cycle”

Work Ordering	Run Length	Total Occupancy	CPU Time	Eff.
No	5:57	116:53	53:03	47%
Yes	4:50	84:04	53:31	67%

Table 3: The effect of ordering

and 17 show the theoretical execution time and efficiency for different resource levels using the distribution from figure 15. To generate these simulated results, we assume that all resources are identical and that none are lost during a cycle, and that each work step always uses its measured average run length. Work steps are distributed to workers using a greedy approach described in [47] in which the longest work steps are sent out first. These results show that in this ideal environment, there is no further benefit to execution time beyond 16 processors. WoDi implements a heuristic which tells it that nothing beyond 16 processors is useful, so it can set this level automatically. Otherwise the programmer is left with the challenge of the cost versus execution time trade-off.

5.4.2 Work Step Ordering

WoDi's primary decision making responsibility is ordering of work steps. Previously, we assumed a greedy work step distribution algorithm [47], and this is exactly what WoDi employs. Table 3 shows the advantage of using this strategy as compared to the original application's approach of sending out work steps in sequential order. Each of these sample runs was made at night on a collection of

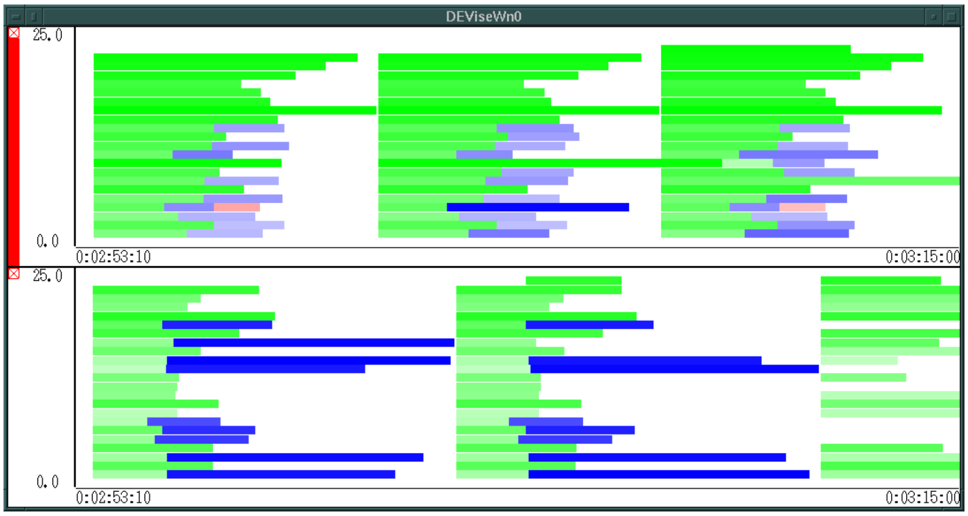

Figure 18: Graphical view of the impact of work step ordering

HP workstations most of which are in a public lab. This lab is closed at night, so the effect of resource reclamation by owners was reduced. They also used identical input data sets so that the total processor time required by the job was constant. Ordering the work steps reduced the run time by more than an hour, saved over thirty hours of resource occupancy, and increased efficiency by 20%. These are dramatic increases, and for this reason ordering across cycles is always done by WoDi.

In figure 18 we show a DeVise visualization of how the ordering improves performance. This graph clearly demonstrates the cyclic behavior of the materials science application. The synchronization at the end of each cycle is apparent in the vertical alignment of the work steps. The lower graph shows what happens when WoDi simply sends steps to workers in the same order in which they are received from the application. Long work steps are sent out last, and therefore

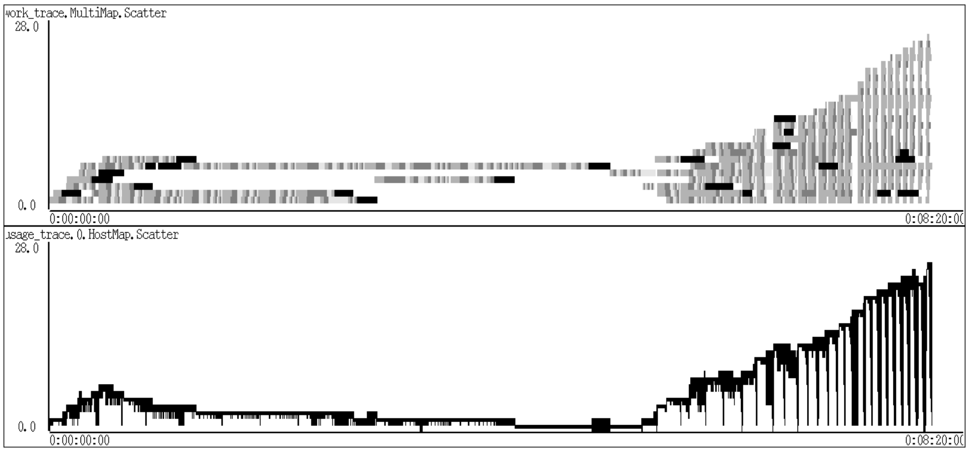

Figure 19: Resource utilization for one run of the materials science application

a number of processes are left idling until these last steps are completed. The large amount of white space shows the wasted compute cycles. In the upper graph, WoDi's ordering has been turned on. Not only is the white space reduced indicating higher efficiency, but the ordered run is well into computing its third cycle by the time the unordered run completes its second cycle showing the improved response time as well.

5.4.3 Adaptability

Figure 19 demonstrates how WoDi is able to adapt to changes in resource availability. One of WoDi's goals is to use the facilities provided by CARMI to adapt to changes in available resources. These changes include newly available resources, resource suspension and resumption, and resource failure. The lower graph shows that at start-up, the application was able to quickly acquire seven machines, but was then unable to acquire additional machines to replace those

	Avg.	Std. Dev.	Min.	Max.
Avg. Occ.	12.5	5.3	3.9	29.3
Run Length	5:07	2:37	2:31	10:35
Efficiency	82%	10.3%	36.43%	94.15%
Tot. Occ.	45:37	11:23	33:34	100:12
Adds	28.2	12.0	11	50
Losses	15.2	11.5	0	40

Table 4: Summary statistics for 61 runs of the materials science application

which had been reclaimed by their owners, and eventually dropped down to as few as one. At approximately the two-thirds point of the run, many new resources became available to the job. The upper graph shows that the majority of the work steps were actually completed in the last third of the job's run time. Without the ability to adapt to changes in available resources, this job would either have had to wait in the queue until sufficient resources were available, or it would have had to limp along with the few resources which were available at start-up time. In either case, the time from the customer submitting the job until its completion would be longer.

To gain a better understanding of how the dynamic environment effects application performance, we ran the application repeatedly on a production cluster of Sun workstations. Over this period, our application had to compete with other Condor customers as well as owners for access to resources. During the day, competition from both sources often made it difficult to acquire machines even though the Condor pool contained around 90 machines in the desired class. At night, competition from owners is very small, and we are only

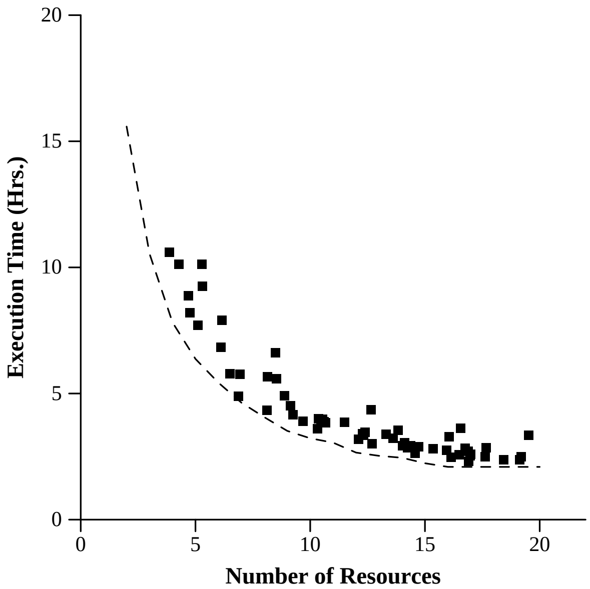

Figure 20: Theoretical and Actual Execution Time and Number of Resources

competing against other Condor customers, so resources are more plentiful. Table 4 summarizes the results of 61 executions.

The top row of the table shows the average number of workstations the application was able to hold during an entire run. As expected, this value varies significantly because of the difference in competition for resources during different runs. The run time also varies due to the changes in resource availability. As we would expect from looking at the theoretical efficiency graph, the efficiency of the application does not vary greatly. Because the amount of work to be done is constant across runs, the steady efficiency leads to occupancy remaining relatively constant. The same amount of allocated processor time yields the same amount of results, regardless of the number of resources being used concurrently. Figure 20 plots the execution time and the average number of resources in use for each of the 61 runs. The dashed curve gives the theoretical execution time for the entire run at different resource levels. The experimental results follow the theoretical curve fairly closely. The difference is due primarily to the effect of resource reclamation by owners.

During an average run of about five hours, 28 resource allocations are performed. This corresponds to one new resource every 10 minutes. As should be expected, resource losses are less common, occurring about once every 20 minutes. This is as expected because if resources were lost more frequently than they could be added, the application would be unable to collect a significant number of resources. The variance in these values is quite high (in fact, one run never had a resource reclaimed while another lost 40). This too should be expected since the frequency of resource gains and losses depends greatly on other activities in the system such as owner reclamation and competition from other Condor customers.

5.4.4 Other applications of WoDi

The master-workers programming paradigm supported by WoDi is versatile, and can be applied to problems in a wide variety of domains. In addition to the purely scientific application described previously, WoDi has been used to parallelize a number of other types of problems including graphics rendering, parallel compilations, and processing of database queries.

POV-Ray

The first application to be developed for WoDi was an extended version of the popular POV-Ray [48] graphics rendering package. POV-Ray is a ray-tracer, and falls into the category known as "embarrassingly" (or pleasantly) parallel because it can easily be decomposed into independent sub-tasks. For this renderer, the sub-tasks are small rectangles of pixels which combine to make the overall image. While not all portions of a ray-traced image take the same amount of time to compute, the WoDi implementation of POV-Ray allows these sub-regions to be made as small or as large as desired. By making them small, the variance in compute time is reduced, which allows nearly perfect speed-ups.

MaMa

Another pleasantly parallel task, which is not truly a parallel application is large compilations. Large compilations generally consist of compiling a number of independent source files into object files, and a final link step which generates the final executable or library. If the compilation of one source file should be aborted, it can simply be restarted on another machine. Therefore, the various components of the compilation only need to synchronize before the linking is done. This also closely fits the WoDi model, so we developed what we call the "Make Machine" (MaMa) which uses WoDi to distribute compilation steps to workers on various resources.

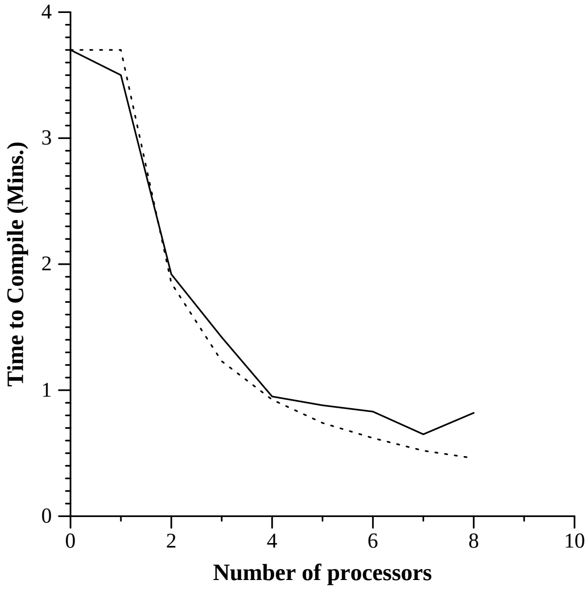

Figure 21: Time to compile POV-Ray via MaMa

To use MaMa, customers need simply to annotate their `Makefiles` to specify which steps should be processed using MaMa. This is generally easy to do because the common steps to be run in parallel are individual source file compilations, and the make program provides the means of specifying a single, default rule for doing single source file compilations. The other step is to get the make program to handle the parallelism. Fortunately, many make programs, including the GNU implementation, already have this capability in order to do parallel compilations on other parallel architectures such as Shared Memory Processors (SMPs).

Figure 21 shows the time required to compile POV-Ray using MaMa to distribute the compilations onto Sparc 10 workstations running SunOS. POV-Ray consists of roughly 28 thousand lines of C code in 41 separate source files.

There are no dependencies between the C files, so POV-Ray is an ideal candidate for parallel compilation via MaMa. The solid line in the figure shows the actual times to do the compilation while the dotted line shows the times if linear speedup were achieved. The time for zero processors is the time required by a single workstation without MaMa. The first interesting point is MaMa actually achieves a lower time to perform the compilation with only one workstation than a single workstation did without MaMa. This is because workstations only become available to MaMa via Condor when they are otherwise idle. Therefore, this machine is essentially dedicated to the compilation job while the local workstation used for compilations has both processor and memory resources used by interactive users. Between two and four processors, the measured results closely match the optimal, but they begin to diverge afterward, and in fact the MaMa time actually increases when moving from seven to eight processors. Both the divergence and the anomaly at eight processors can be traced to the distribution of time required for compiling the individual source files. The majority of the individual source files take between three and six seconds to compile, but one source, which appears roughly in the middle of the compilation procedure, requires 23 seconds to compile. Just as with the materials science application before, this late occurring, long running work step causes a good deal of inefficiency. Because of the way the distribution maps to the processors, this long running step actually starts earlier when there are seven processors which gives it a lower overall compilation time. In fact, combining the necessary

23 second single source compilation with an average link time of 7 seconds gives an optimal compilation time of 30 seconds. With seven processors, the total time required was 39 seconds which is quite close to this optimal value.

It would be preferable if MaMa could generate cycle notations for WoDi just as the materials science application did so that the compilation steps could be re-ordered automatically. This is not practical, though, because in general compilations do not follow a cyclic pattern. A full compilation of all source would generally be followed by a compilation of only a subset of the source which has been changed, or perhaps by a full compilation of an entirely different set of source. We therefore recommend that MaMa customers simply write their `Makefiles` such that the largest source files are compiled first. This can be done once, and then all subsequent compilations, whether they be full or partial, would benefit from this ordering.

Reef

The Reef project has used WoDi to distribute database queries as work steps via WoDi. In Reef, the worker processes were each full Coral [49] deductive database servers which were modified to receive queries via WoDi. The Reef master process acts as a front end to which queries are submitted, and they are then distributed to the various database servers via WoDi. To achieve better performance, Reef introduced the notion of *affinities* to WoDi. An affinity states that there is a preferred, but not required, worker for a particular work step.

In Reef, affinities occur because the Coral servers will cache relations on which they run a query. Therefore, performing another query on the same relation will run faster if we assign it to a server which has already cached the relation. By annotating each work step with an affinity value, WoDi will first look for a worker which has recently computed another work step with the same affinity value when deciding which worker to give the step to. If such a worker is available, the work step is sent there. If no worker with the desired affinity value is free, the work step will be sent to any idle worker.

5.5 Summary

Even with the availability of a powerful set of resource management services, it is often desirable to provide applications with higher level services which closely match a particular programming model. By providing these higher level services, we insulate programmers from managing individual resources, so they can concentrate on their application. This makes writing new applications easier, but it also allows them to benefit from our experience with managing resources. This can lead to better performance than even an application written to explicitly manage its resources. We have implemented an application framework, WoDi, which provides a high level of abstraction for writing master-workers style parallel programs. In addition to insuring that applications written using WoDi run correctly, WoDi is able to gather information about the work being done, and the resources provided by CARMI to improve the performance of the application. We are able to do this both by reordering the work steps generated by the application and by choosing appropriate resource levels. We have used WoDi with applications from a number of problem domains, and have been able to get programs running easily, and with a high degree of efficiency.

Chapter 6

Checkpointing Services

Checkpointing is the act of saving the entire state of a program, whether it be sequential or parallel, to secondary storage in a way that it can be later restarted on other resources and have the computation continue. Providing checkpointing services adds to the flexibility of a resource management system in a number of ways. First, the ability to checkpoint a customer's job means that the system is not locked into any particular resource allocation decision if conditions within the system change such that a different resource allocation decision should be made. The RM system can accomplish this by checkpointing a job and restarting it on the new, more appropriate resource. For example, Condor's ability to checkpoint sequential programs has allowed it to effectively utilize the idle time of privately owned workstations for long running jobs. By checkpointing a program when an owner reclaims a machine, Condor is able to run programs which take much longer than any single idle interval at a workstation. As another example, checkpointing permits the RMS to insure that no low priority jobs cause other, higher priority jobs to wait. The low priority jobs can simply be checkpointed to make room for those with higher priority.

A second use for checkpointing services, particularly for parallel programs, is to perform *dynamic partitioning* which has been shown [50] to be more effective than static methods of allocating resources to parallel programs. In a dynamic partitioning scheme, the number of resources allocated to a job is changed while the job is running based on changes in load on the overall system. For example, as new jobs are submitted, resources may be revoked from running jobs. Without the ability to checkpoint and save the state of running processes, it would not be possible to move processes to perform a dynamic partitioning resource reallocation without interacting with the running application. The conditions under which dynamic partitioning is beneficial depend greatly on the overhead involved in doing resource re-allocation. When this overhead becomes high, the benefits of dynamic partitioning are lost. It is therefore important to perform checkpoint and restart operations as quickly as possible.

A final usage of checkpointing is to provide fault tolerance. By periodically checkpointing a running program, the RMS can insure that if any resource being used by the program should fail, it can be restarted from an intermediate state. This prevents losing all of the results computed up until the failure. Again, parallel programs benefit greatly from this ability because their use of multiple resources increases the likelihood of a failure. Frequent checkpoints improve fault tolerance because the probable interval between a checkpoint and failure will be reduced. This reduces the amount of computation which has to be performed again after the restart. To perform checkpoints frequently, though, the time required to create a checkpoint needs to be short. Otherwise, if the time to make a checkpoint is long, and the interval between checkpoints is short, the program will spend too much time checkpointing, and not enough computing.

Techniques for checkpointing parallel and distributed programs have been understood for quite some time. For example, Chandy and Lamport proposed a "distributed snapshot" protocol in 1985 [51]. Thus far, however, implementations of these techniques for production parallel systems have been rare. Building on the theory and the experience gained doing checkpointing for single process jobs in Condor, we have developed a system for checkpointing message passing parallel programs called CoCheck (for Consistent Checkpointing). CoCheck implements a network consistency protocol much like Chandy and Lamport's distributed snapshot protocol, and utilizes the single process checkpoint ability of Condor to save the state of each process in a parallel application.

In practice, checkpointing a parallel program tends to be a time consuming operation. The exact attribute which makes parallel programs desirable, their ability to perform computations which are extremely large in both computation and memory requirements, makes them difficult to checkpoint. In particular, a checkpoint must by definition include the entire state of the running program.

A parallel program's state consists of the state of the interconnection network as well as the address space of each process. The combined memory of all of the processes often amounts to a huge overall state which must some how be moved into secondary storage. There are two potential bottlenecks in saving this data: the interconnection network over which the data must travel, and the secondary storage devices on which the data will be stored. Because the RMS makes decisions related to checkpointing, it must determine how and where the checkpoints will be stored. To give the RMS flexibility in making these decisions, we have introduced a *checkpoint server* which performs checkpoint file store and retrieve operations on demand. Using checkpoint servers, the scheduler is able to precisely direct the movement of checkpoints, and is not at the mercy of an external mechanism, such as a distributed file system, in which it cannot precisely control resource utilization.

In the next section, we examine alternative methods for storing checkpoint files, and describe our checkpoint servers which have been developed explicitly for this purpose. Following this, we investigate the performance of checkpoint servers in various conditions, and how the performance will influence their deployment. Next, we describe how checkpoint servers are handled by Condor. The chapter concludes with a description of CoCheck, our system for checkpointing parallel applications, and how it has been integrated with our previous work on CARMI and WoDi.

6.1 Methods of storing Checkpoint Files

When a RMS receives a request for an application to be checkpointed, it must also determine how that checkpoint will be stored. That is, it must decide what resources to dedicate to the checkpoint operation. Checkpointing a parallel application creates a very large burst of data which must be stored reliably and as quickly as possible. This bursty pattern is exactly the condition under which most communication and storage systems perform poorly.

The tolerance to latency in performing a checkpoint will depend on the environment in which the parallel application is running. In a situation in which use of a resource may be revoked (such as for privately owned workstations), there is a degree of real-time bound in saving the data. Condor, for example, places a constraint in each workstation's classified ad which specifies the upper bound on the time allowed for a checkpoint when an owner reclaims a machine. If the checkpoint is not complete within this interval, Condor kills the job rather than waiting for the checkpoint to complete. In an environment where resources have no policy or limits imposed by owners there may be no hard constraint, but it is still very important to complete the checkpoint as quickly as possible in order to free the resources for other jobs. Time spent checkpointing is time when useful computation is not taking place.

The simplest, and perhaps most desirable method of storing a checkpoint of a parallel program is to simply use an existing distributed file system. Examples of these include the Network File System (NFS) [52] and the Andrew File System

(AFS) [53]. Using these systems for storing checkpoints is attractive because it allows them to be stored in the same way as other user data. The problem of where and how data is stored is handled by the file system. However, these systems were not designed to perform well on operations which involve one time transfers of large files such as checkpoints.

Specifically, NFS has stateless servers which handle file requests a single page at a time. This leads to poor performance because the file must be moved across the network via a series of page size requests to the server. On other hand, AFS uses a more complex, full file caching scheme in which all files are moved in their entirety between the server and client disks. Still, practice has shown that AFS is not adequate for parallel systems. As an example, the Cornell Theory Center, which operates a large IBM SP-2 parallel computer, recommends that AFS not be used when data transfers become large [54]. There are a number of reasons for this poor performance. First, the caching scheme used by AFS does not perform well when writing large results such as checkpoint files. These results generally will not be re-used on the node where they are generated, so caching them locally provides no future benefit, and may cause other, useful data to be flushed from the cache. Second, AFS executes two disk writes (one locally and one on the server) for the entire file. With fast interconnection networks, the latency of disk accesses becomes a bottleneck. A final difficulty with AFS is the inflexibility in placing file servers. AFS servers are considered insecure unless placed in a "locked room" to which users do not have access. This limits our ability to place AFS servers so that they will be close to the processes generating checkpoints.

An additional complication with using existing distributed file systems is the file system may encompass different resources than the RMS. If a checkpoint were to be stored via a distributed file system, it could only be restarted on a resource sharing the same file system domain. This puts a limit either on where a job might be restarted, or perhaps even a limit on the size of a pool that might be managed by a single RMS. Complex, and wide-area sharing of resources, such as provided by Condor's flocking mechanism, would not be possible.

6.1.1 Checkpoint Servers

Due of the perceived shortcomings of the existing solutions, we have developed a *Checkpoint Server* specifically suited for the problem of storing and retrieving checkpoints. The goal of our checkpoint server is simply to move data between the network and a local disk as quickly as possible. It is the RMS' job to determine when a checkpoint should take place, and to allocate a server for storing each checkpoint file which is generated.

Like other simple components, checkpoint servers can be combined to form more complex structures. A RMS can therefore use multiple checkpoint servers as building blocks to provide good checkpointing performance. The opportunity to select the number and placement of resources, in this case checkpoint servers, is a luxury that RMS's rarely have since they commonly are deployed in pre-existing compute environments. To exploit this opportunity, we considered a number of factors. Perhaps most important is the topology and characteristics of the underlying communication network. In a large, fragmented network with high latencies and low bandwidth, we want to scatter checkpoint servers about to insure that any checkpointing process has as fast a link as possible to some checkpoint server. In a smaller, more tightly connected network, we do not need as many checkpoint servers since every potential checkpointing process will always have a fast path to a server.

In addition, we considered the characteristics of the checkpoint servers themselves. In particular, even when network bandwidth is plentiful, other characteristics of the servers such as disk bandwidth or processor speed may become a bottleneck. Also, the capacity of the disk is important. A server with a small disk should not be placed in a location where it will be required to store many checkpoints. A RMS must understand these characteristics of its checkpoint servers, and use them when making scheduling decisions.

A final consideration we made when deciding how to use checkpoint servers is how frequently checkpoint operations take place. In an opportunistic system, such as Condor, the return of a single owner may cause a large parallel application to checkpoint. For this environment, it is worthwhile to allocate significant resources to checkpointing because they will be needed frequently. In all environments the frequency of checkpoint operations is going to be determined by the way in which checkpointing services are utilized.

The availability of checkpoint servers may also influence other scheduling decisions made by a RMS. For example, the application RM may use information about the location of checkpoint servers when deciding where to place application processes. Processes should be spread around the resources such that no single checkpoint server will be overloaded in case there is a need to checkpoint. Knowledge of the characteristics of the checkpointing infrastructure should be used in making this decision. The RMS must balance its desire to distribute checkpoints evenly with the application's need for high bandwidth and low latency communication that results from clustering the application's processes. Applications which do not perform extensive communication may be distributed based on the expected checkpointing requirements, while communication intensive applications may be positioned to reduce application communication time at the cost of higher checkpoint times.

6.2 Measuring the performance of checkpoint servers

As described in the previous section, before deploying checkpoint servers, we had to understand the need for checkpointing services as well as the characteristics of our communication infrastructure. The Computer Sciences department at the University of Wisconsin has approximately 200 desktop workstations

Checkpoint Route	Time
Source and server on same sub-net	46
Source and server on separate sub-nets, same router	64
Source and server on separate FDDI connected sub-nets	79
Source on Ethernet, server on FDDI	49

Table 5: Times, in seconds, to write a 32Mb checkpoint file

most of which are available to Condor for executing long running sequential or CARMI applications. Checkpoints in this environment are triggered whenever an owner returns to a workstation running a Condor job, which occurs relatively frequently. We therefore require a checkpoint server architecture which can service numerous checkpoints.

The principle limitation in our environment, as in many other environments, is the available network bandwidth. Each of our workstations lies on an Ethernet class sub-net. Each Ethernet is connected to one or two routers which in turn directly connect each sub-net to three to five other sub-nets as well as an FDDI backbone. The path between any two workstations, therefore, is at best the Ethernet rate of 10 $\frac{Mbit}{sec}$, and may require crossing one or two routers. The department also has AFS available to all of the workstations, and the AFS servers are connected directly to the FDDI ring which has a theoretical bandwidth of 100 $\frac{Mbit}{sec}$. We therefore wish to explore alternatives in placing checkpoint servers on various sub-nets as well as directly on the FDDI ring.

Table 5 summarizes the results of experiments to determine how the network topology affects the time to write a checkpoint. In each of these experiments, a 32Mb checkpoint file was generated on a SPARC workstation running SunOS

4.1.3. The checkpoint files were received at checkpoint servers running on Dec Alpha workstations running OSF/1 V2.1. The checkpoints themselves were transferred via the Transmission Control Protocol (TCP) because it is a reliable protocol, and is available on all of our workstations. The results reported are the average of a number of checkpoint operations. In all cases, the variance in the time to checkpoint was low. As would be expected, placing the checkpoint server and checkpointing process on the same sub-net produced the best results. Placing the checkpoint server on FDDI performed nearly as well. In the tests where the checkpoint had to move off of one sub-net and onto another, the time increased considerably.

From these results, it seems that the most desirable method of placing checkpoint servers would be one per sub-net. In this way, every workstation will have the fastest available path to a server. However, there are two disadvantages to this. First, the number of sub-nets is large (approximately a dozen containing owner's workstations), so many resources would have to be established as checkpoint servers. Second, although placing a checkpoint server on each sub-net will improve checkpoint times, to gain the same advantage at restart time would require re-scheduling a job on the same sub-net as when it last checkpointed. This severely limits the number of resources available for a restarting job. As an alternative, a hierarchical scheme could be used. A small checkpoint server could be placed near to the resource on which the checkpoint is taking place, but after the checkpoint is complete, it could be moved to some larger higher level server from which the restart will occur. This movement to the higher level server could take place off-line, when there is no immediate need for the checkpoint at any particular site.

Placing checkpoint servers directly on the FDDI ring appears to be nearly as desirable as having a checkpoint server per sub-net. However, at some point the single FDDI bandwidth will also become a bottleneck. Requiring a router to put traffic onto an Ethernet sub-net, whether coming from another Ethernet or from FDDI significantly reduces the available bandwidth.

Our second set of tests was intended to determine if checkpointing performance scales as the number of checkpointing processes, servers and size of individual checkpoints is increased. We also wanted to see exactly how well an existing file system, AFS, performs on these operations. Once again, our tests were limited by the bandwidth on our network. It is clear that no single checkpoint can occur faster than the bandwidth of a single sub-net, so we did not want two checkpointing processes to lie on the same sub-net. This constraint limited us to checkpointing no more than two processes simultaneously. Figure 22 shows the results of different checkpoint server configurations and checkpoint sizes.

In all cases, the time to checkpoint scaled nearly linearly with the size of the checkpoint files. The most striking result is how poorly AFS performs for checkpoint operations. As mentioned previously, this is due to the method in which AFS caches files. As a checkpoint is being written, it is stored entirely

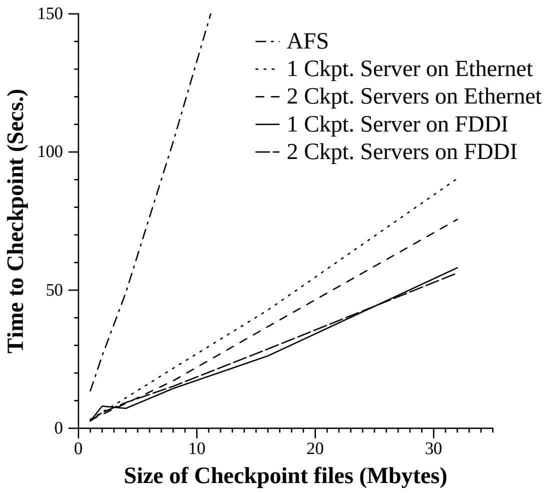

Figure 22: Time to checkpoint 2 processes of various sizes

on the local disk. When the file is closed, it is read off the disk, and transfered across the network to the file server where it is written to the server's disk. This requires three disk I/O's as opposed to one for the checkpoint server. Nonetheless, it is surprising that AFS required approximately an order of magnitude more time to produce a checkpoint.

In the checkpoint server cases, the results are what one might expect by extrapolating from the single checkpoint tests. Sending two 32Mb checkpoints to the same checkpoint server takes almost exactly twice as long (90 seconds) as sending one checkpoint to one server. When sending to two checkpoint servers on separate sub-nets, we were constrained by our environment to go across the FDDI ring. The last of two checkpoints completed in virtually the same time as one checkpoint taking a route across FDDI. Sending two checkpoints to one server on FDDI took only slightly longer (58 seconds) than sending one checkpoint onto FDDI (49 seconds). This implies that the network is still the bottleneck in this operation, and that the processor and disk are not limiting performance. As more sub-nets feed the same FDDI connected checkpoint server, we would expect the disk to become the bottleneck. The fact that an additional server on the FDDI ring does not improve checkpoint performance further demonstrates that the single server is not yet a bottleneck.

6.3 Allocating Checkpoint Servers with Condor

As discussed in chapter 3, the structure for representing, and allocating resources in Condor is the classified ad. The goal of these structures is to provide a simple, and consistent method for allocating resources of various types. Therefore, for Condor to be able to allocate checkpoint servers on demand, these servers must also generate a classified ad with sufficient information for them to be used by other entities within Condor.

We must, therefore, determine what attributes need to be included in a checkpoint server's classified ad. The attributes provided should be sufficient so that constraints can be written to find an appropriate checkpoint server based on the attributes of the checkpoint being saved. The classified used for checkpoint servers in our production Condor pool is shown in figure 23.

Like all classified ads, the checkpoint server's ad starts with a definition of

SelfType	= CheckpointServer
MatchType	= CheckpointFile
Machine	= elm
Disk	= 7143355
Subnet	= 128.105.1
IP_ADDR	= <128.105.1.20:5651>
IntervalStart	= Fri Jul 5 11:01:48 1996
IntervalEnd	= Fri Jul 5 12:08:11 1996
NumSends	= 18
BytesSent	= 163206192
TimeSending	= 237
AvgSendBandwidth	= 663653
NumRecvs	= 20
BytesReceived	= 148508696
TimeReceving	= 316
AvgReceiveBandwidth	= 667067
target.size < my.Disk - 1000	

Figure 23: Sample Checkpoint Server Classified Ad

its own type (`CheckpointServer`) and the type of its desired match. In this case, the checkpoint server is paired with actual checkpoint files. Next, we are told what workstation this checkpoint server is actually running on, and the amount of free disk space on the machine for storing checkpoints. The checkpoint server's class ad also contains the subnet on which it is located. As we have seen, relative network position plays an important role in determining the time required to create a checkpoint. By providing the server's subnet in the attribute list, constraints can be written to find a checkpoint server on the same subnet as the checkpointing process. Next, the Internet Protocol address of the checkpoint server is displayed, so that the other processes know how to get in contact with the checkpoint server. Finally, some summary data about activity on the checkpoint server is given. In this case, the additional information is primarily for monitoring the performance and load on the server rather than for matching. The constraint simply states that the size of the checkpoint file must not fill the local disk.

Because checkpoint servers advertise themselves to the global resource allocator like all other resources, the process of allocating a checkpoint server when required is straight forward. In fact, it is essentially identical to the process of allocating any new resource to an application. The first step is simply to detect that a checkpoint needs to be done. This may be done either by the resource access control component when a policy decision forces a checkpoint, or on demand of the application using CARMI's checkpointing API which is described later in this chapter. In either case, initiating a checkpoint is simply a RM service request just like any other, so it is sent to the application RM process. To find an appropriate checkpoint server, we have the application RM consult the global resource allocator. It does this by creating a classified ad which represents the checkpoint request. This classified ad has a constraint specifying the required amount of disk space for the checkpoint, and attributes with information about where the checkpoint is being generated, such as the sub-net where the checkpointing process is located. The checkpoint file classified ad is then run through the normal matchmaking mechanism to determine the best checkpoint server. When the application RM has been allocated a server, it contacts the server to request a checkpoint operation. The server responds with a World Wide Web style Universal Resource Locator (URL) which has a service prefix (e.g. “http:” or “ftp:”) which is specific to our checkpoint server. The application RM simply passes this URL on to the application itself which writes its checkpoint to the server.

6.4 CoCheck

Checkpoint has proved to be an extremely powerful resource management tool for utilizing opportunistic resources in Condor. Building on this success, we wanted to provide checkpointing services to parallel programs as well. The result is CoCheck (for Consistent Checkpointing) which provides checkpointing services for parallel programs which communicate via a Message Passing Environment (MPE). It has been developed in collaboration with researchers at the Technical University of Munich. Building on our existing experience with PVM, we chose it as the first MPE for development of CoCheck. Work is ongoing in Munich to extend CoCheck to support MPI [55].

We started with a number of important design goals when developing CoCheck. The first goal of CoCheck was to remain portable. That is, although the first implementation was done on top of PVM, the concepts used in CoCheck should be applicable to any MPE. Second, CoCheck must not require modifications to the MPE implementation. This helps with portability both across different message passing systems, and for maintaining compatibility with updated releases of a single system. It also permits us to implement CoCheck on systems for

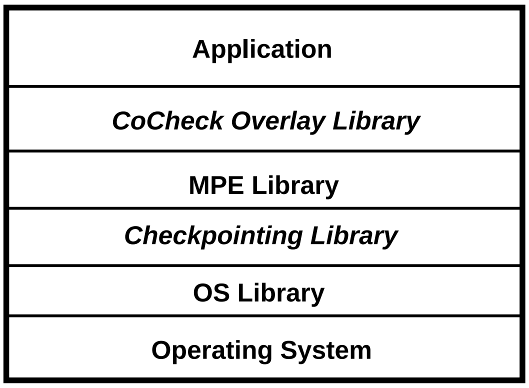

Figure 24: Layering of the CoCheck components

which source code is not available. Third, we required flexibility in the degree of checkpointing to be performed. For example, it may be desirable to create a global checkpoint of all processes in a parallel application, or it may only be necessary to checkpoint a single process to perform a migration. The degree of checkpointing to be performed is placed under the control of the application and the RMS. Finally, CoCheck must have no *residual dependencies* on resources after a checkpoint is complete. This is to say that a resource cannot be required to participate in the life of a parallel program after all processes running there have been checkpointed. This requirement provides us with fault tolerance, and allows us to completely evacuate a resource when the resource utilization policy requires it.

6.4.1 CoCheck Components

A checkpoint requires us to save the entire state of a running program. The state of a message passing parallel program at any given time consists of the state of each process in the application as well as the condition of the communication network which may be carrying or buffering messages in transit. To capture this state, and to meet our design goals, we have developed CoCheck in three components: an overlay library for the message passing API, a single process checkpointing library, and a RM process which coordinates the checkpointing protocol. The two libraries are linked into every application process generating a service layering as shown in figure 24. The overlay library is the key to doing checkpointing without modification to the underlying MPE. This library provides a stub for every function defined by the MPE. These stubs trap all application calls to the MPE, and perform communication identifier or other translations which must be made as a result of previous checkpoints and restarts. In most cases, the stub will in turn call the original MPE function to get the actual communication service performed. The overlay library also implements the protocol to capture the network state which is described below.

Single process checkpointing libraries have existed for quite some time. CoCheck utilizes the checkpointing library which was developed as part of Condor [56, 57] which, among others, provides this functionality without any modifications to the operating system on which it runs. The technique used for performing single process checkpointing is similar to the message passing overlay library described above (indeed, the techniques used in single process checkpointers were an inspiration for CoCheck's overlay approach). The state of a single process includes its memory (the bounds of its address space), the state of the processor registers, and any state within the operating system kernel such as the set of open files and their current seek position. Determining the bounds of the address space and saving the registers of a process are typically easy to perform. Either operating system calls or well defined variables provide this information. However, the increasing use of techniques such as dynamically loaded libraries have made address space lay-outs more complex making this a more difficult task. The overlay functions in a checkpointing library catch calls to the kernel which modify the kernel state of a process (for example, opening a file), and record this information so that it can be saved in the checkpoint and restored upon restart. Not all of the state of a process can be saved. For example, the parent-child relationship of processes following a `fork()` system call, or inter-process communication outside the scope of the MPE (e.g. pipes or sockets) cannot be retained. The Condor checkpointing library therefore disallows these system calls by trapping them and returning an error.

The final component of CoCheck, the resource manager process, is the coordinator for the entire system. An RM process which is an extension of the stand alone PVM RM process [58] has been developed. We have also extended CARMI's RM process to support CoCheck. In either case, the application RM process receives requests for checkpointing services, and initiates the CoCheck protocol between itself and the overlay library of each the application processes to perform these services. The stand-alone RM also writes a meta-checkpoint

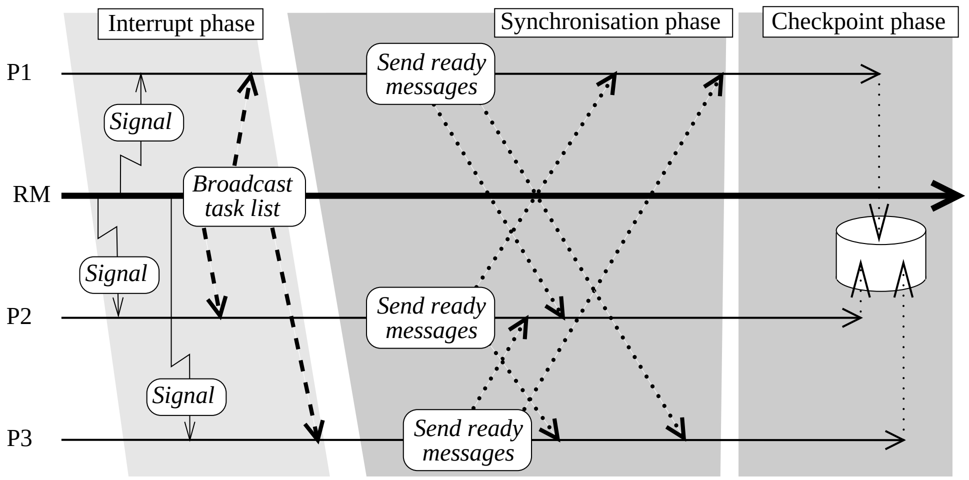

Figure 25: CoCheck's network cleaning protocol

file which can later be re-read by a new instance of the RM to provide the information needed to restart the entire computation. By using these three components, we have been able to meet our design goals, and at the same time leverage much pre-existing technology.

6.4.2 CoCheck Protocols

The CoCheck network cleaning protocol (shown in figure 25) is responsible for ensuring that the entire state of the network is saved during a checkpoint, and to insure that communication can be resumed following a checkpoint. The CoCheck protocol begins when the RM determines that a checkpoint is required due to an application request, a change in the state of a resource or a RMS policy decision. The RM begins by sending a Unix signal and a message to each of the application processes. The combination of signal and message is required because each process may be either computing or communicating. The signal will interrupt a process which is computing causing it to enter the CoCheck library to participate in the checkpoint protocol. The overlay library of a process which is communicating will simply see the checkpoint request message, and interpret it as a request to begin checkpointing.

The checkpoint request message sent from the RM to each process contains two pieces of information. The first is how the process should participate in the checkpoint. The most common alternative is for the process to checkpoint itself. In this case, the message contains the URL to which the process should write its checkpoint file. The message may instead tell the process not to checkpoint at all, or it may request that the process generate a new, CoCheck specific, URL from which another process may read its checkpoint. This last alternative provides a means of performing a direct process migration without the need to create an intermediate checkpoint file. After requesting a migrating process to generate a URL, the application RM can start a new process, and provide it with this URL as the source of its checkpoint file. When the new process reads the checkpoint from the original process, it will have effectively performed a migration without the checkpoint ever being stored on disk.

The second piece of information the checkpoint request message contains is a list of communication identifiers of processes which are also checkpointing. This list provides each process the information needed to insure that the network has been fully cleaned. The checkpointing processes each send a *ready* message to

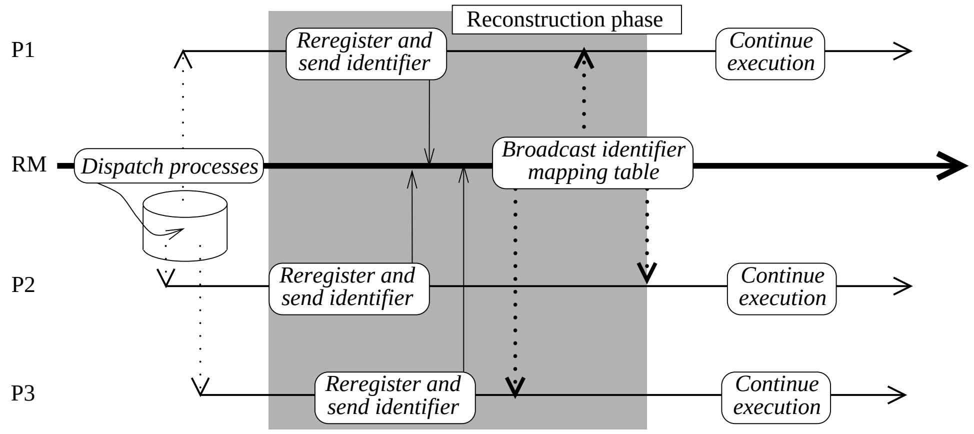

Figure 26: CoCheck's restart protocol

each of the other processes in this list, and then waits for ready messages from all the other processes. Any other messages which are received while waiting for readys are assumed to be part of the application's communication, and are buffered in the process' address space. Since they are stored within the address space they become part of the process' checkpoint, and they can be retrieved from memory after restart. With the provision that messages are delivered in order between any two processes, it can be assumed that the network has been drained when all of the ready messages are received. At this point, it is safe for each process to disconnect from the MPE (if the MPE requires it), and to invoke the Condor checkpointing library to save its state.

Restarting requires each process to read its checkpoint file, then establish communication identifier mappings to hide changes in the communication environment from the application. The restart protocol is shown in figure 26. As each process is restarted, it is provided with a URL from which to read its individual checkpoint by the RM. Prior to performing a restart operation, the process connects itself with the MPE to establish a new communication identifier. It next invokes the single process restart mechanism within the Condor checkpointing library while preserving its new communication id. When the Condor library finishes reading the checkpoint file, the entire state of this process has been restored. To resume communication the processes send both its new and old communication ids to the RM process. The RM collects all of these ids from the restarting processes, and sends each process a mapping of old ids to new ids. When the mapping is received, it is installed in the CoCheck library, and all future communications will go through this mapping before passing into and out of the MPE implementation. In this way, processes are able to continue to use the original communication ids which were in use prior to the checkpoint. After the new mapping is installed, the application processes resume from the point at which the initial checkpoint notification was received.

6.4.3 CARMI CoCheck API

We designed CoCheck to be flexible in the number of processes to be checkpointed and where the checkpoint is to be written. To leverage this flexibility, the interface to CoCheck must also be flexible. The basic interface to CoCheck is with the `carmi_ckpt()` function. `carmi_ckpt()` takes arguments for specifying three groups of processes: those that should be checkpointed, those that should block while the processes remain checkpointed in order to maintain a consistent communication identifier space, and those processes which should neither checkpoint nor block. This last group simply runs the network cleaning protocol with each of the checkpointing processes, then continues. This last group, therefore, can be assured that no messages are lost with the checkpointing processes, but they cannot continue to use the same communication identifiers. Processes may be placed in this "clean only" group when the application deems it inappropriate for these processes to remain blocked while the other processes are checkpointed. The application, however, will also have to insure that any future communication between the checkpointed processes and the "clean only" processes uses newly updated communication identifiers. Processes which request a checkpoint commonly put themselves in the "clean only" group so that they will not be blocked and can receive confirmation that the checkpoint is complete. The complement to `carmi_ckpt()` is `carmi_restart()`. This function takes a list of checkpointed processes and a list of blocked processes which are waiting for new communication identifier mapping tables as arguments. The checkpointed processes are restarted, and the blocked processes resume after receiving the new communication identifiers for the restarted processes.

Like all other CARMI calls, the checkpoint and restart functions are asynchronous remote procedure calls handled by the application RM process. Therefore, a process requesting a checkpoint need not block while the CoCheck protocol is running, and while the individual checkpoints are being stored. It is legal, though, for a process to include itself in the list of processes to be checkpointed. When a checkpoint or restart is complete, the application RM sends a completion notification message to the requesting processes. A process which requests its own checkpoint will not, however, receive the completion notification message because it is not active at the time the checkpoint completes.

The last function in CARMI's CoCheck support API is `carmi_migrate()`. The purpose of this function is to simply move a process from one resource to another without creating an intermediate checkpoint file. The arguments to `carmi_migrate()` specify a single process to be migrated, and the identifier of a resource to which it should be migrated. A list of processes which should run the cleaning protocol to maintain consistent communication identifier mappings with the migrated process must be provided when calling this function. No "clean only" group is permitted because when the migration operation is complete, all of the processes in the consistent group also continue to run. They do not remain blocked indefinitely as the do with `carmi_ckpt()`.

An important CARMI service which is related to checkpointing is the ability to receive a notification about an event at a resource, and perhaps most importantly notification when a resource must be vacated. Often, applications will request notification when a resource must be vacated, and respond to this notification by either checkpointing or migrating processes running there.

The basic CARMI CoCheck functions are intentionally designed to be very general. Using these functions as a basis, though, a variety of more special case, and easier to use, checkpointing functions can be developed. For example, it is easy to design calls which checkpoint an entire parallel application or a single process. It is also possible to generate watchdog daemons which use these services to automatically checkpoint or migrate processes in the event of resource revocation.

6.4.4 Comparison with other PVM checkpointing systems

The advantages of checkpointing and migrating processes for MPEs and PVM in particular have been noted by a number of research groups. As a result, there have been a number of other efforts to provide these services for PVM including MPVM [59] (for "migrateable PVM") and DynamicPVM [60]. The approaches taken by these groups have been very much the same, but they did not meet the requirements of our environment. Therefore, we proceeded to develop CoCheck.

Both MPVM and DynamicPVM implement checkpointing by directly modifying the PVM implementation. In particular, they make changes to the PVM daemon processes. Following a checkpoint, the PVM daemon on the process' original host becomes responsible for forwarding or storing messages for a migrated or checkpointed process. This did not satisfy two of our design goals: it is not portable (even a new release of PVM requires re-writes of these systems), and it produces residual dependencies. Because forwarding is carried out by the PVM daemons, all daemons must remain alive, even after all processes on a host have been migrated away. In our target environment of privately owned workstations, this was not acceptable. This technique can, however, lead to higher performance. CoCheck requires that all application processes be stopped during a checkpoint or migration of only one process. By using the daemon processes to handle message consistency, the application processes need not be halted.

6.4.5 WoDi extensions to exploit checkpointing

As checkpointing became available in CARMI, we looked to see how WoDi could be extended to take advantage of these services. The basic WoDi model assumes that every worker is stateless, and that any work step may be given to any worker. Because of this, applications cannot be written which place any dependency on which work steps will go to which workers, and workers cannot reliably accumulate more state as a result of completing work steps. Likewise, very long running work steps are not well supported by WoDi because a resource failure or revocation during the computation of a work step requires that the work step be restarted from the beginning. When work steps are long, it becomes decreasingly likely that the work step will ever find a resource which is stable enough for it to complete. The inclusion of checkpointing services in CARMI allows WoDi to break some of these restrictions.

To support stateful and long running workers, we have extended WoDi to include the notion of *worker classes.* A worker class is a collection of worker processes which share certain characteristics, namely whether they can be checkpointed, and whether they maintain state between work steps. The basic WoDi model assumed that workers were neither stateful nor checkpointable, so every worker died when its resource was revoked. Even in this model, worker classes provide additional benefit to applications by allowing them to specify different types of work steps which go to different types of workers. Workers which maintain state are no longer anonymous. Each worker is differentiated from the others by its current state. Therefore, each worker within a class is assigned a unique instance value. To allow applications to exploit this feature, WoDi permits work steps to be directed to particular instances when they are sent, and the application can discover which instance computed a particular result when it is received. If a worker is checkpointable, WoDi has additional flexibility in dealing with resource revocation and in scheduling of work steps.

With checkpointable workers, WoDi need not allow workers to die when a resource is revoked by its owner. Instead, workers that can be checkpointed will be checkpointed if they maintain state between work steps or are engaged in computing a result. If the worker is not stateful and idle, there is no need to checkpoint it when the revocation occurs because no state and no computation will be lost.

WoDi can also generate checkpoints to better utilize its resources. For example, consider a case in which there are five workers in a class and currently four resources on which they can run. One of these workers must be checkpointed since WoDi does not permit a single resource to be shared among workers. If a work step arrives for the checkpointed worker and another worker is idle, the idle worker can be checkpointed to make room to restart and schedule this new work step on the appropriate worker.

DiViS - Distributed Visualization of Simulations

A common method of utilizing a resource management package such as Condor is to submit a group of simulation runs, each with a different set of parameters, as a batch. The result of this batch generates a curve or other result for the simulation collection. This process is cumbersome because the customer is required to generate the input data sets for each run, and has no way of knowing for sure if the proper data points have been selected until the batch is complete. With DiViS we attempt to alleviate these difficulties by using the checkpointing features of WoDi. We treat each simulation as a stateful and checkpointable WoDi worker process. The DiViS master process keeps each of the simulations running by sending it work steps representing the amount of simulation time which should be run. In addition, DiViS provides a real-time visualization of the results of all of the runs, and the ability to steer the computation by adding or eliminating any particular run in progress.

In addition to WoDi, DiViS utilizes a discrete event simulation library called *MiMic*. MiMic is based on C++, and provides a number of useful functions for

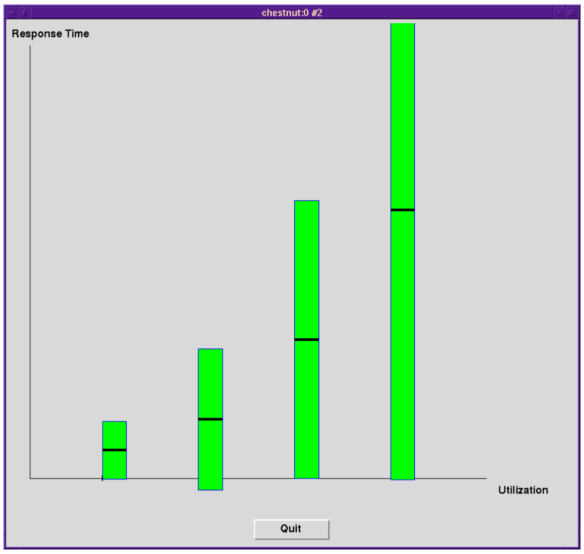

Figure 27: Sample output of DiViS visualization

writing simulations such as threads, scheduling of simulation events, gathering of statistics, and generation of random numbers with various distributions. Modifying an existing MiMic simulation for use with DiViS requires only calling a DiViS initialization function prior to other processing in the simulation. This initialization function returns a float value which is considered to be the input parameter for this run of the simulation. This initialization function installs a periodic event to occur within the simulation. Each time this event is triggered, the current value of a MiMic statistics gathering “probe” object is sent back to the DiViS master process. The input parameters for each of the simulations as well as the name of the probe object to be displayed are provided by the customer as input to DiViS master process which is submitted to CARMI.

Figure 27 shows an example of the visualization provided by DiViS. In this example, there are four simulations of a queueing network running. The load on the queueing network is the input parameter, and is shown on the horizontal access, and the average response time for each of the models is shown on the vertical. The height of the bars represents the confidence interval for that simulation. In a typical simulation, longer running times generate tighter confidence intervals. Being able to visualize these confidence intervals while the the simulations run is another advantage of DiViS. For many simulations, it is difficult to predict how long to let a simulation run in order to get the desired level of confidence. Because DiViS shows this graphically, the customer can simply let the simulation run, and stop it when the interval is satisfactory. Customers can also click a mouse button on one of the result bars to get exact result values. In addition, DiViS' graphical user interface provides the customer with the ability to begin new simulations and to halt or delete simulations which are already running. By providing this visual feedback, we provide customers with far greater power in running a collection of simulations than would be found in a standard batch environment.

6.5 Summary

Checkpointing is a powerful resource management service, but to be utilized effectively, it requires planning throughout the entire RMS. The inclusion of servers dedicated to checkpointing allows the RMS more control over how checkpoints are stored, and leads to significantly better performance than relying on existing technologies such as distributed file systems. Extending checkpointing to parallel environments is not a new idea, but also not commonly provided. With the checkpointing mechanism in place applications can be provided with services for invoking this mechanism to assist in fault tolerance and load balancing. Exploiting checkpointing within WoDi has permitted us to support new types of applications which require stateful or long running workers. As an example, we have developed DiVis which uses WoDi to run collections of related simulations and provide customers with a graphical representation of the simulations' results.

Chapter 7

Conclusions and Future Research Directions

Many scientists today have a nearly unlimited appetite for computing power. They will use any resource to which they can gain access, but accessing these resources is not always practical. The complexity of the environment and ownership concerns present barriers to those needing the compute power, even if it is otherwise not being used. It is the job of resource management services to tame the complexity of the environment while satisfying owners' policies on individual resources. When RM services are available, their customers can focus all available computing resources toward solving their problems.

Customers of RM services require that they be available continuously, 24 hours-a-day, seven days-a-week. Without this continuous availability, the systems which provide RM services will be considered unstable, and will therefore

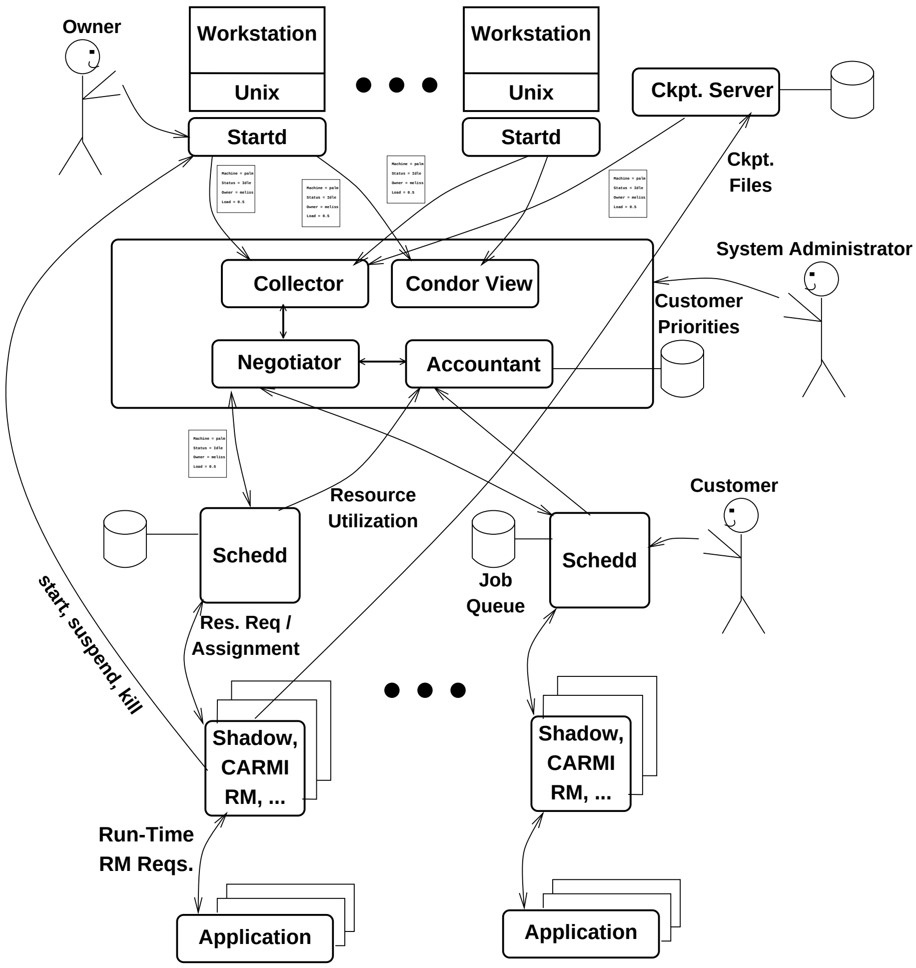

Figure 28: The components of Condor

never succeed in a production environment. To provide a stable environment, we have developed a layered approach to implementing a RMS. Figure 28 shows how Condor has been implemented using these layers. With this approach, we have achieved stability in the system by maintaining well defined roles and interfaces for each of the system's layers. The fundamental problem in developing an RM system is how resources and customers will be represented. In chapter 3, we described the classified ad structure for representing both of these. Classifieds are versatile enough to represent both resources and customers, and provide a symmetrical method for matching the two together. Matchmaking is performed by evaluating the boolean constraint in each classified. Because the constraint is a boolean expression, it can be used to represent a large class of policies or resource requirements. When a match is made by the RMS, it informs the customer that a resource has been granted. Classifieds are used in Condor to represent individual resources such as workstations and checkpoint servers, and to represent customers' requests.

As described in chapter 4, Condor's ability to use different application RM server processes for different "universes" of jobs has allowed us to develop a powerful parallel programming environment consisting of PVM, CARMI and Condor. As a first step in the development of CARMI we created an interface to an existing parallel programming environment. By using an existing system, we avoided redeveloping communication services, and provided a foundation of familiar services to programmers using CARMI. CARMI supplies new services to applications through an asynchronous remote procedure call interface which permits a single application process to have multiple requests outstanding at one time. This increases parallelism in the system, and prevents the process from having to block while long running services are being handled. CARMI's services include allocating and gathering information about resources, and process creation facilities. We implemented CARMI as a new application "universe" in Condor. This allows CARMI applications to utilize all of the resources in a Condor pool for use by customers running parallel applications.

Building a layer above CARMI, we determined that applications can benefit from services which provide more abstractions, and internally manage individual resources for an application. The result, described in chapter 5, is WoDi which has a master-workers style parallel programming interface. WoDi has been applied to a number of applications, and in one case has improved performance for an application previously written in a lower level parallel programming system.

Checkpointing is a RM service that is valuable for a RM system to provide, and that spans all levels of a RM system implementation. Checkpoint files, and the locations where these files are stored are themselves resources to be managed. To permit control, and improve performance of these services, we developed a checkpoint server which Condor can allocate just as it does other resources in the system. With these resources in place, we developed CoCheck which is a system for checkpointing parallel applications. CoCheck has been integrated with CARMI to provide services for accessing CoCheck's facilities, which in turn enabled WoDi to run applications which cannot withstand loss of individual workers.

7.1 Future Research Directions

Condor and CARMI provide a very solid foundation for continuing to develop new sorts of RM services. Because of the general nature in which Condor represents resources, it is possible to look at other types of resources in the computing environment which can benefit from management services. One example is network bandwidth which Condor presently does not handle. Condor could, however, be extended to measure the bandwidth on various network links, and represent them with classified ads. Once this information has been gathered, it could be used in a variety of ways. Clearly, this information would be useful in making checkpoint server allocation decisions. As was shown in chapter 6 checkpointing bandwidth can vary greatly even within one local area network, so actual measurements are beneficial in scheduling decisions. When the resource pool expands to a wide area network, such as via Condor flocking, the huge variety in network links can have enormous impact on the time required to write a checkpoint. It is therefore extremely important to make a good selection for the checkpoint's destination. In addition, network performance information can be used internally by parallel applications when making communication decisions. For example, others have shown [61] that by using information about the performance of network links, parallel applications can improve the performance of collective communication operations such as a broadcast.

Another sort of resource which RM systems can increase customers ability to access is data. Today, there is a wealth of information scattered about the Internet and the World Wide Web. There are many methods, such as browsers, for humans to view this data, but using this for computation is not easy. Typically, the data must be copied to a local site in order to perform computations on it. This has disadvantages of requiring local storage for potentially enormous data sets, and the loss of consistency if the source data is updated. As an example, Condor's checkpointing library is already able to access checkpoint files stored anywhere on the Internet because it utilizes the World Wide Web's URL naming scheme. RM services can be created which make URLs available to applications in the same way in which local files are named today.

Because the focus of the work on CARMI has been resource management, we specifically did not want to address communication services. Therefore, we chose the most popular MPE of the time, PVM, and developed the interfacing methodology using it. Today, the Message Passing Interface (MPI) is becoming increasingly popular, so it would be desirable to support the MPI communication primitives with CARMI RM primitives. In addition, because MPI is intended to a be a written standard, it has spawned numerous implementations. Making the MPE to RM system interface a part of this standard would potentially allow CARMI or other RM service providers to support a variety of MPI implementations with no changes. The MPI standard group is also working to establish a minimal set of RM services as part of the standard. In particular, process creation facilities will apparently be included in the next release of the standard. Just as CARMI has been able to support the RM functionality defined by PVM, CARMI should also be able to support whatever is specified by MPI.

As shown in chapter 5, providing higher level services on top of CARMI eases the job of programmers, and can yield good performance. We have so far supported the master-workers programming paradigm, but it would be valuable to have high level services to support other approaches. One popular approach is domain decomposition in which a large data structure is broken across a number of processors to aggregate both memory and compute power. This approach does not lend itself to adaptation to resource changes as easily as master-workers, but the inclusion of checkpointing services makes it feasible. This higher level set of services could aid the programmer in partitioning the data set, and could re-partition the data as needed due to changes in available resources. In addition, other programming paradigms, such as Cilk [31], are defined such that the generated programs are malleable. Using CARMI to provide the run-time environment for an approach such of Cilk would make all of the resources managed by Condor available to these applications.

Bibliography

[1] W. Ludwig, *Algorithms for Scheduling Malleable and Nonmalleable Parallel Tasks.* PhD thesis, University of Wisconsin – Madison, 1995.

[2] E. Dijkstra, "The structure of the "THE"-multiprogramming system," *Communications of the ACM*, vol. 11, pp. 341–347, May 1968.

[3] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor: A hunter of idle workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 104–111, June 1988.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A Users' Guid and Tutorial for Networked Parallel Computing.* Cambridge, MA.: The MIT Press, 1994.

[5] M. P. I. Forum, "MPI: A message-passing interface standard," May 1994.

[6] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for UNIX development," in *Proceedings of the Summer 1986 Usenix Conference*, pp. 93–112, 1986.

[7] D. R. Cheriton and W. Zwaenepoel, "The distributed V Kernel and its performance on diskless workstations," in *Proceedings of the Ninth Symposium on Operating Systems Principles*, pp. 129–140, ACM, October 1983.

[8] A. S. Tanenbaum, R. V. Renesse, H. Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. Rossum, "Experiences with the Amoeba distributed operating system," *Communications of the ACM*, vol. 33, pp. 46–63, December 1990.

[9] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite network operating system," *Computer*, pp. 23–26, February 1988.

[10] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A network transparent, high reliability distributed system," in *Proceedings of the Eigth Symposium on Operating Systems Principles*, pp. 169–177, ACM, December 1981.

[11] S. Zhou, J. Wang, X. Zheng, and P. Delisle, "UTOPIA: A load sharing facility for large, heterogeneous distributed computing systems," Tech. Rep. CSRI-257, Computer Systems Research Institute, University of Toronto, April 1992.

[12] R. H. Arpaci, A. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, "The interaction of parallel and sequential workloads on a

network of workstations," Tech. Rep. CS-94-838, UC Berkeley, November 1994.

[13] D. Duke, T. Green, and J. Pasko, "Research toward a heterogeneous networked computing cluster: The Distributed Queuing System version 3.0," tech. rep., Supercomputer Computations Research Institute, Florida State University, March 1994.

[14] IBM Corporation, *IBM LoadLeveler: User's Guide*, 1993.

[15] R. Henderson, "Job scheduling under the Portable Batch System," in *Job Scheduling Strategies for Parallel Processing* (D. G. Feitelson and L. Rudolph, eds.), vol. 949 of *Lecture notes in Compter Science*, Springer-Verlag, 1995.

[16] D. Ferguson, Y. Yemini, and C. Nikolau, "Microeconomic algorithms for load balancing in distributed systems," in *Proceedings of the 8th International Conference on Distributed Computer Systems*, pp. 491–499, IEEE, 1988.

[17] J. Kay and P. Lauder, "A fair share scheduler," *Communications of the ACM*, vol. 31, pp. 44–55, January 1988.

[18] M. Mutka, *Sharing in a Privately Owned Workstation Environment.* PhD thesis, University of Wisconsin – Madison, 1988.

[19] D. G. Feitelson, "A sureney of scheduling in multiprogrammed parallel systems," Tech. Rep. RC 19773, IBM T. J. Watson Research Center, 1994.

[20] D. G. Feitelson and L. Rudolph, eds., *Job Scheduling Strategies for Parallel Processing*, vol. 949 of *Lecture notes in Compter Science*, Springer-Verlag, May 1995.

[21] D. G. Feitelson and L. Rudolph, eds., *Job Scheduling Strategies for Parallel Processing*, 1996.

[22] J. Flower, A. Kolawa, and S. Bharadwaj, "The Express way to distributed processing," *Supercomputing Review*, pp. 54–55, May 1991.

[23] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3 user's guide and reference manual," Tech. Rep. ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[24] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-passing Interface.* Scientific and Engineering Computation Series, Cambridge, MA.: The MIT Press, 1995.

[25] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference.* Scientific and Engineering Computation Series, Cambridge, MA.: The MIT Press, 1996.

[26] R. Butler and E. Lusk, "Monitors, messages and clusters: The P4 parallel programming system," *Parallel Computing*, vol. 20, pp. 547–564, April 1994.

[27] R. J. Harrison, "Portable tools and applications for parallel computers," *International Journal of Quantum Chemistry*, vol. 40, pp. 847–863, 1990.

[28] G. D. Burns, R. B. Daoud, and J. R. Vaigl, "LAM: An open cluster environment for MPI," in *Proceedings of the Supercomputing Symposium '94*, (Toronto, Canada), June 1994.

[29] M. P. I. Forum, "MPI-2: Extensions to the Message-Passing Interface," July 1996.

[30] D. Gelernter, "Generative communications in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 80–112, January 1985.

[31] C. F. Joerg, *The Cilk System for Parallel Multithreaded Computing.* PhD thesis, Massachusetts Institute of Technology, January 1996.

[32] D. Gelernter and D. Kaminsky, "Supercomputing out of recycled garbage: Preliminary experience with Piranha," in *Proceedings of the ACM, International Conference on Supercomputing*, July 1992.

[33] B. Kingsbury, *The Network Queueing System.* Sterling Software, Palo Alto.

[34] W. Cai, M. Livny, and J. Pruyne, "Using classified advertisements for resource management," tech. rep., Dept. of Computer Sciences, University of Wisconsin–Madison, In preparation.

[35] J. Siegel, *CORBA - Fundamentals and Programming*. New York, NY: John Wiley & Sons, 1996.

[36] A. Bricker, M. Litzkow, and M. Livny, "Condor technical summary," Tech. Rep. 1069, Computer Sciences Department, University of Wisconsin–Madison, January 1992.

[37] D. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne, "A world-wide flock of Condors: Load sharing among workstation clusters," *Future Generation Computer Systems*, vol. 12, pp. 53–66, May 1996.

[38] J. Pruyne and M. Livny, "Parallel processing on dynamic resources with CARMI," in *Job Scheduling Strategies for Parallel Processing* (D. G. Feitelson and L. Rudolph, eds.), vol. 949 of *Lecture notes in Compter Science*, Springer-Verlag, 1995.

[39] J. Pruyne and M. Livny, "Interfacing Condor and PVM to harness the cycles of workstation clusters," *Future Generation Computer Systems*, vol. 12, pp. 67–86, May 1996.

[40] N. Starling, "A dynamic, self-configuring, distributed computing facility." `http://www.dur.ac.uk/ dit0nas/jisc.html`, June 1995.

[41] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan, "Dome: Parallel programming in a distributed computing environment," in *Proceedings of the 10th International Parallel Processing Symposium*, (Honolulu, HI), pp. 218–224, IEEE, April 1996.

[42] V. Herrarte and E. Lusk, "Studying parallel program behavior with upshot," Tech. Rep. ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, August 1991.

[43] M. T. Heath, "Recent developments and case studies in performance visualization using paragraph," *Performance Measurement and Visualization of Parallel Systems*, pp. 175–200, 1993.

[44] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn parallel performance measurement tools," *IEEE Computer*, vol. 28, November 1995.

[45] M. Livny, R. Ramakrishnan, and J. Myllymaki, "Visual exploration of large data sets," in *Proceedings of the IS&T/SPIE Conference on Visual Data Exploration and Analysis*, January 1996.

[46] W. A. Shelton, G. M. Stocks, R. G. Jordan, Y. Liu, L. Qui, D. D. Johnson, F. J. Pinski, J. B. Staunton, and B. Ginatempo, "First principles simulation of materials properties," in *Proceedings of SHPCC '94*, pp. 103–110, May 1994.

[47] R. L. Graham, "Bounds on multiprocessing timing anomalies," *Siam Journal of Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[48] C. Young and D. Wells, *Ray Tracing Creations, Second Edition*. Waite Group Press, November 1994.

[49] R. Ramakrishnan, D. Srivastava, and S. Sudarshan, "Coral - control, relations and logic," in *Proceedings of the International Conference on Very Large Databases*, 1992.

[50] M. Squillante, "On the benefits and limitations of dynamic partitioning in parallel computer systems," in *Job Scheduling Strategies for Parallel Processing* (D. G. Feitelson and L. Rudolph, eds.), vol. 949 of *Lecture notes in Compter Science*, Springer-Verlag, 1995.

[51] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 63–75, Feb. 1985.

[52] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the Sun network file system," in *Proceedings of the Summer Usenix Conference*, pp. 119–130, 1985.

[53] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance

in a distributed file system," *ACM Transactions on Computer Systems*, vol. 6, pp. 51–81, February 1988.

[54] J. Gerner, "Input/output on the IBM SP2–an overview." `http://www.tc.cornell.edu/SmartNodes/Newsletters/IO.series/intro.html`.

[55] G. Stellner, "CoCheck: Checkpointing and process migration for MPI," in *Proceedings of the 10th International Parallel Processing Symposium*, (Honolulu, HI), pp. 526–531, IEEE, April 1996.

[56] M. J. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the Unix kernel," in *Proceedings of the Winter Usenix Conference*, (San Francisco, CA), 1992.

[57] T. Tannenbaum and M. Litzkow, "The Condor distributed processing system," *Dr. Dobb's Journal*, pp. 40–48, February 1995.

[58] J. Pruyne and M. Livny, "Providing resource management services to parallel applications," in *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing* (J. Dongarra and B. Tourancheau, eds.), SIAM Proceedings Series, pp. 152–161, SIAM, May 1994.

[59] J. Casa, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "Mist: PVM with transparent migration and checkpointing,"

in *Proceedings of the 3rd PVM Users' Group Meeting*, (Pittsburgh, PA), May 1995.

[60] B. J. Overeinder, P. M. A. Sloot, R. N. Heederik, and L. O. Hertzberger, "A dynamic load balancing system for parallel cluster computing," *Future Generation Computer Systems*, vol. 12, pp. 101–115, May 1996.

[61] B. Lowekamp and A. Beguelin, "Eco: Efficient collective operations for communication on heterogeneous networks," in *Proceedings of the 10th International Parallel Processing Symposium*, (Honolulu, HI), pp. 399–405, IEEE, April 1996.

Appendix A

CARMI User's Guide

A.1 Submitting CARMI jobs via Condor

Submitting a job to CARMI is much the same as submitting any other job to Condor. The biggest difference is the requirement that the Condor job universe must be set to PVM. Second, the number of machines desired must be specified. This is done with a `machine_count` statement. A `machine_count` is of the form: `machine_count = n..m` where `n` is the lower bound on number of machines required, and `m` is the upper bound on the machines desired. A job which can use as many machines as it can get can simply specify an extremely large upper bound. A sample CARMI submit file is shown in figure 29.

This submission file assumes the job can use between one and five machines of the same type as the one the submission command is run on. Getting machines of a type other than the submitting machine requires explicitly writing

```
universe = PVM
executable = carmi_program
arguments = -foo -bar 2
output = out_file
error = err_file
# This job needs between 1 and 5 machines.
machine_count = 1..5
queue
```

Figure 29: Sample Condor submission file for a CARMI job

a requirements expression in the submission file, prior to the queue statement.

One job can request machines of different types by alternating requirements and queue statements as in figure 30. In this example, Condor will allocate between one and three workstations running Ultrix4.3 to the job, exactly four workstations running HPUX, and between one and one hundred machines running SunOS to the job prior to starting the job. Each requirements expression defines another CARMI resource class. These classes are named by number, starting with "0" for the first `requirements` statement, and increasing by one for each queue statement. Therefore, the Ultrix machines will be in CARMI class "0," the HPUX machines in CARMI class "1," and the SunOS machines in class "2." As will be seen later, CARMI provides services for defining new classes, and retrieving these class definitions at run-time.

```
machine_count = 1..3
requirements = OpSys == "Ultrix4.3" \&\& Arch == "MIPS"
queue
machine_count = 4..4
requirements = OpSys == "HPUX9" \&\& Arch == "HPPAR"
queue
machine_count = 1..100
requirements = OpSys == "SunOS4.1.3" \&\& Arch == "sun4m"
queue
```

Figure 30: Submitting a Multi-class CARMI job

A.2 Differences in PVM under CARMI

CARMI uses PVM for the basic set of application communication primitives. Therefore, there are no changes in how PVM application processes communicate when running under CARMI. CARMI also supports the PVM resource management API (e.g. `pvm_spawn()` with a few important exceptions.

First, where ever PVM uses an architecture name (for example "SUN4"), CARMI replaces it with a class name. Any place PVM uses an architecture as a parameter (e.g. `pvm_addhosts()`), or returns it in a function call (e.g. `pvm_config()`), CARMI replaces the architecture with a class name.

CARMI also does not use the directory structure normally used by PVM for storing binaries. CARMI assumes that all binaries are stored in the same directory from which the job was submitted. Therefore, any processes to be created via the `pvm_spawn()` function should have their binaries stored in the same directory from which the user submits the CARMI job to Condor.

A.3 CARMI API

The CARMI API adheres to the request protocol set-up by our resource management framework [39]. That implies that every CARMI function immediately returns a request identifier, and that every function requires an argument which specifies the tag to be used on the request response message. The function definitions below are followed by a returns(...) which specifies the data types in the response message. All of these messages are assumed to begin with a request identifier field, which is not shown here.

A.3.1 Request Management

```
RequestId  carmi_cancel_request(RequestId id,
            ResponseTag resp_tag)
returns(BOOLEAN success)
```

carmi_cancel_request the outstanding request with id. The response message returns false if no request with the given id is outstanding. The side effects of canceling a request are dependent on the type of request which has been deleted.

A.3.2 Resource Handling

```
RequestId  carmi_config( ResponseTag resp_tag)
returns(int nhost, HostId hosts[nhost])
```

carmi_config returns the number of hosts currently available to the application, and a list containing the host identifiers.

```
RequestId  carmi_get_class_ad(HostId id, ResponseTag resp_tag)
returns(ClassAd machine_ad)
```

This function returns all available information concerning a host. The type "ClassAd" is borrowed from Condor, and it includes information which characterizes a particular resource. The ClassAd structure is extensible, but the minimum content includes attributes such as processor and operating system type. A ClassAd is a C++ object, and includes methods for discovering all the attributes stored in the ad, and retrieving the value associated with any attribute.

```
RequestId  carmi_addhosts(char *class_name, int count,
              ResponseTag resp_tag)
returns(int count, HostId new_hosts[count])
```

carmi_addhosts requests new hosts in class class_name. Hosts will be added, one at a time, until count total have been added. Multiple response messages to the addhosts functions may be generated until the sum of the count in all of the responses equals the count in the initial request.

```
RequestId  carmi_delhosts(int count, HostId hosts[],
              ResponseTag resp_tag)
returns(int count, HostId host_id[count])
```

This function requests that hosts be deallocated from the application. The response message contains the host identifiers of the machines successfully removed. In a successful execution, this list will exactly match the list given in the request.

```
RequestId  carmi_add_notify(char *class_name,
             ResponseTag resp_tag)
returns(SAME as addhosts above)
```

carmi_add_notify requests that messages be sent to the application when new hosts are added to the system due to the original set of requests in the submission file (as opposed to a run-time addhosts() request). The response messages are the same as those returned in response to an addhosts request. This request is outstanding until explicitly canceled (though after all resources specified by the submit file have been granted it will never be triggered).

```
RequestId  carmi_delete_notify(int count, HostId host_ids[],
             ResponseTag resp_tag)
returns(SAME as delhosts above)
```

carmi_delete_notify requests that messages be sent to the application when a host is lost due to an event other than an explicit carmi_delhosts() request (e.g. hardware failure, or host being reclaimed by its owner). The format of these notification messages is the same as the response to a carmi_delhosts() request.

```
RequestId  carmi_{suspend, resume}_notify(int count, HostId
             host_ids[], ResponseTag resp_tag)
```

`returns(int count, HostId host_ids[count])`

These two functions request that the application be notified whenever a host in the specified host list is suspended or resumed by Condor.

A.3.3 Task Management

```
RequestId  carmi_class_spawn(char *executable, char **argv, char
              *class_name, ResponseTag resp_tag)
returns(ProcessId id)
on process exit:  returns(ProcessId id, int status, int
CPU_usage)
```

carmi_class_spawn triggers the creation of a new process. The new process will be started on a host within the class specified. When the process has successfully been created, a response message containing the identifier of the next task is returned. When that process exits, the application receives a message containing exit status information and processor utilization.

```
RequestId  carmi_process_info( ResponseTag resp_tag)
returns(int nprocs, struct procinfo proc_list[nprocs])
```

carmi_process_info returns information on the process running in the system. Information included in the procinfo structure includes the ProcessId, the HostId where the process is running, the ProcessId of the process' parent, and the name of the executable file being run.

A.3.4 Class Management

```
RequestId  carmi_class_definitions( ResponseTag resp_tag)
returns(int count, ResourceClass list[count])
```

This returns the definition of all existing classes. A ResourceClass definition is a string containing an expression which defines the characteristics a resource must have to be considered a member of a class. The string will be the same as the requirements expression in the submit file, or provided to carmi_define_class described below.

```
RequestId  carmi_define_class( ResourceClass,
             ResponseTag resp_tag)
returns(BOOLEAN success)
```

carmi_define_class defines a new resource class, but does not request that any hosts in this class be added. The return value is false if a class with the same name already exists. The string provided in the ResourceClass parameter is of the same form as a requirements expression in the Condor submit file.

```
RequestId  carmi_remove_class( char *class_name,
             ResponseTag resp_tag)
returns(BOOLEAN success)
```

This function removes the class with the given name. carmi_remove_class fails if any hosts in this class are currently allocated to the application. Therefore, all hosts in the class must be deallocated via carmi_delhosts prior to calling carmi_remove_class.

A.3.5 Checkpointing

```
RequestId  carmi_ckpt(int ckpt_cnt, ProcessId *ckpt_list, int
             consistent_cnt, ProcessId *consistent_list, int
             flush_cnt, ProcessId *flush_list)
returns(BOOLEAN success)
```

This function provides a general interface to checkpointing services provided by CoCheck. carmi_ckpt specifies three groups of processes. The first group has ckpt_cnt members with task identifiers in the vector ckpt_list. The processes in this group are checkpointed. The next group, specified by consistent_cnt and consistent_list will be suspended until the processes in ckpt_list have been re-started. Because they are held suspended, CoCheck is able to install communication identifier maps in the processes, so that they will continue to use the same communication identifiers after the checkpoint as they did before the checkpoint. The last, flush, group will simply have the network cleared of any messages in transit to the checkpointing processes. These processes will continue immediately, but will not be able to continue to use the same communication identifiers for the processes in ckpt_list following a restart.

```
RequestId  carmi_restart(int restart_cnt, ProcessId *restart_list,
             int consistent_cnt, ProcessId *consistent_list)
returns(int count, ProcessId[count])
```

carmi_restart is used to restart processes checkpointed previously via carmi_ckpt. The restart_list specifies the processes to be re-started. Typically, these will be the same as the ckpt_list group in the call to carmi_ckpt. The processes in consistent_list are assumed to be blocked waiting for these processes to restart as a result of being in the consistent_list of a previous call to carmi_ckpt.

```
RequestId  carmi_migrate(ProcessId target, HostId host, int
               consistent_cnt, ProcessId *consistent_list)
returns (ProcessId new_id)
```

This function performs a migration of the process specified by target to the machine specified by host. If the machine is no longer available, or of a different class than where target is initially running, an error is returned. The processes in consistent_list will be blocked during the duration of the migration, and will be able to continue communicating with the migrated process using the identifier specified in target. A new ProcessId is returned which can also be used for communicating with the migrated process.