Interfacing Condor and PVM to harness the cycles of workstation clusters

Jim Pruyne and Miron Livny

Department of Computer Sciences
University of Wisconsin–Madison
{pruyne, miron}@cs.wisc.edu

Abstract

A continuing challenge to the scientific research and engineering communities is how to fully utilize computational hardware. In particular, the proliferation of clusters of high performance workstations has become an increasingly attractive source of compute power. Developments to take advantage of this environment have previously focused primarily on managing the resources, or on providing interfaces so that a number of machines can be used in parallel to solve large problems. Both approaches are desirable, and indeed should be complementary. Unfortunately, the resource management and parallel processing systems are usually developed by independent groups, and they usually do not interact well together. To bridge this gap, we have developed a framework for interfacing these two sorts of systems. Using this framework, we have interfaced PVM, a popular system for parallel programming with Condor, a powerful resource management system. This combined system is operational, and we have made further developments to provide a single coherent environment.

1 Introduction

As it became apparent that heterogeneous clusters of workstations will be an important source of processing cycles, many researchers have engaged themselves in efforts to develop the technology needed to harness the aggregate resources of these clusters. Since the mid 80s, the problem of how to effectively and efficiently exploit the processing capacity of such clusters has been addressed by a wide range of research and development efforts. One group of researchers has approached the problem from the prospective of parallel processing. These researchers have been looking for ways to use the multiple resources of a cluster to solve a single, large problem in parallel. The development of Message Passing Environments (MPEs) has been the primary result of efforts guided by the parallel processing approach. Message passing allows multiple independent computations to proceed in parallel with partial results being passed via messages in order to form the desired complete solution. The most successful and most widely used MPE so far has been the Parallel Virtual Machine (PVM) system which was developed at the Oak Ridge National Laboratory [1]. PVM provides an easy, reliable message passing model, and is freely available.

Another approach to capturing the power of workstation cluster has been to develop distributed Resource Management (RM) systems. An RM system provides an interface which allows users to easily find resources on which to run their jobs and to monitor the state of the jobs as they run. Typically, RM systems use a batch interface which permits a user to submit a collection of jobs to be run whenever and wherever resources become available. Efforts that have been guided by this approach have been focusing on mechanisms and policies for identifying available resources in a cluster and allocating them to the queued jobs. A number of research and commercial systems of this type have been developed as a result of these efforts including Condor [2], DQS [3], LoadLeveler [4], and Utopia [5].

Condor is unique among these systems because it has, from its inception, been designed to be opportunistic, utilizing idle cycles on privately owned workstations. To do this, Condor must monitor every workstation within the pool it controls to determine when resources are in use, and when they have become idle. Successfully computing in such an environment also requires checkpointing mechanisms which enable a computation to be suspended when the owner of the workstation returns and to be later resumed on the same or a different machine. Condor is freely available and has been used in a wide range of production environments for almost ten years. It has proven to significantly improve the productivity of processing environments that consist of small as well as large clusters of workstations.

Unfortunately, thus far the development of MPEs and RM systems have been mostly independent of one another. Because of this, MPEs usually are ineffective in discovering resources on which to run. Likewise, RM systems are usually unable to bring multiple resources together in order to run a single, parallel application. This is unfortunate since the two play complementary roles. In this paper we present a framework for interfacing an MPE and an RM system, describe the Condor Resource Management Interface (CARMI) that we have developed to interface PVM and Condor, and report on our experience running PVM application in a production Condor pool of more than 100 workstations.

Our framework is guided by the observation that in order to fulfill the promises of high performance computing, parallel applications must be provided with effective resource management services. The Resource Manager of a system determines how long an application must wait for the CPUs it requires, how much memory will be allocated to it, and how balanced the computation will be. The resource manager is responsible for allocating resources among all jobs submitted to the system (Inter-Job RM), and binding resources to requests made by a single job (Intra-Job RM). Without effective inter-job RM, resources may be misallocated among users, placing some at an unfair disadvantage. Poor intra-job RM may cause users' programs to perform poorly because requests may be fulfilled using resources which are heavily loaded or do not closely match the application's requirements.

Using this framework, we have implemented the CARMI resource management system which provides RM services to PVM applications that run on a cluster controlled by the Condor batch system. CARMI utilizes the Resource Management protocols and tasks that have been supported by PVM since the 3.3 release. It has been operational for more than a year and has been used by a number of real-life applications.

The rest of this paper is organized as follows. The next section describes in more detail the difficulties of providing resource management services to parallel computations. Section 3 provides our proposed interface for combining RM and MPE systems, and describes its implementation under PVM. This is followed by a description of CARMI, a new API for parallel programming in an opportunistic environment. Section 5 describes additional support we have built on top of CARMI to make it easier to use, and section 6 presents some experience from using these systems. The final section presents conclusions and directions for future work.

2 Resource Management for Message Passing Environments

To fulfill the promises of high performance computing, parallel applications must be provided with effective Inter-Job and Intra-Job resource management services. Without them, users are burdened with resource control decisions which is likely to lead to poor utilization of resources. Resource management for parallel applications, however, is complex.

Inter-job RM is complicated by the necessity of allocating groups of possibly heterogeneous hosts to a single application. This decision may involve, for example, choosing between allocating a few resources to a number of jobs with small demands, or providing many resources to a single job with large demands. Intra-Job RM is also problematic. An example of an Intra-Job RM service which must be performed is starting processes as they are requested by the application. These processes must be placed so that each process is executed on appropriate hardware while maintaining a balanced load across all machines. A final complication is the continuous interaction between a parallel application and the RM system. Unlike sequential applications, parallel applications may be able to make use of new resources if they become available during the run. A resource manager for parallel applications must therefore continue to interact with the application during the course of the run to make changes in the state of the global system visible to running jobs.

The variety of message passing environments which are available pose additional complication in resource management for parallel applications. To get applications started, these systems generally provide simplistic RM functionality, but they are not as flexible as specialized RM systems. A specialized RM system is capable of providing additional functionality beyond what the MPE provides, but interfacing the MPE to an RM system currently requires extensive modifications to both systems.

To allow existing RM systems to be more easily interfaced with message passing environments, we propose that all RM functionality be removed from the MPE implementation. Instead, MPEs should provide a well defined interface for RM to be handled externally. This interface will provide the means for an application and a RM system to communicate. This approach has a number of advantages. First, adapting a new RM system to a MPE does not require any changes to the MPE code. Second, an RM system may customize or extend the set of RM functions defined by the MPE. Finally, a single RM system does not need to be changed greatly to support new MPEs. Only the communication primitives used need to be changed.

3 Building a framework for Resource Management

The first step in migrating resource management services out of a MPE is determining exactly which services will be handled by the RM processes and which will continue to be handled by the MPE. We view the resource manager as the decision maker in the system. Therefore, all service requests pertaining to host addition or deletion, task assignment, and queries about the state of the system should be in the domain of the resource manager. The MPE is responsible for communication and other services which help insulate the application and the RM from the underlying operating system and hardware. The MPE must be careful not to provide application tasks with access to services which circumvent the policies imposed by the RM. For example,

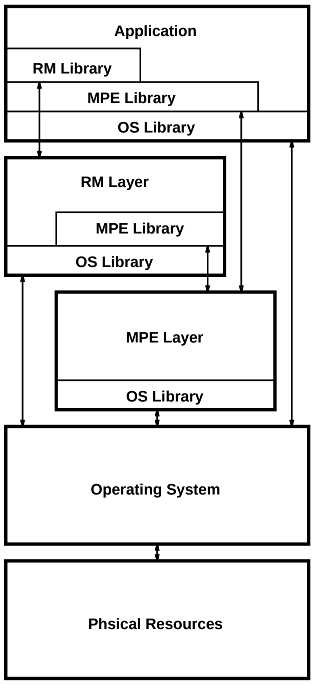

Figure 1: Layers of a system with distinct MPE and RM components

process creation is a desirable service for the MPE to provide, but this service should only be available to RM processes. By handling process creation, the MPE shields the RM developer from the diverse process creation primitives supplied by various operating environments. However, if application tasks are able to create processes directly they may defeat the efforts of the RM to maintain load balancing. Figure 1 shows the layering of service providers.

The distinction between resource management services and communication services is enforced by splitting them into separate libraries. 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 using this library directly. The RM library uses the communication library to send requests for service to the RM system. The RM system also uses the communication library to access services provided by the MPE for communicating with application tasks.

3.1 Processing RM requests outside the MPE

For resource management to be performed externally, applications must be able to communicate with the RM system to make requests. The easiest way to do this is to use the communication primitives provided by the MPE. The RM system can therefore appear to the application as another process within the communication domain of the MPE. RM requests are made by sending a message to this process. Figure 2 shows a generic Resource Management request exchange between an application task and the RM process. We believe that all run-time RM requests should be handled asynchronously. That is, when a process makes a request for a RM service, it should not be blocked 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.

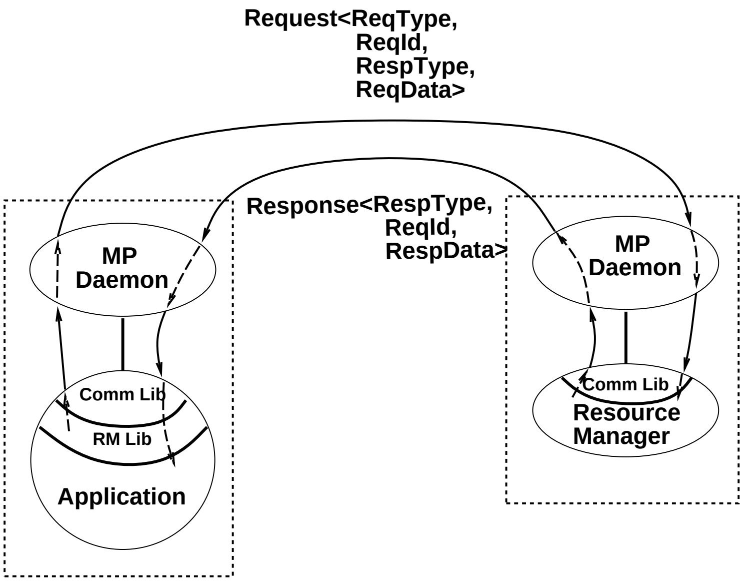

Figure 2:

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 in starting a group of new processes. Instead of making a series of requests to start individual processes, an application can have a number of start-process requests outstanding. These requests may be handled in parallel by the RM reducing the overall time required to start the processes.

Figure 2 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 ID and a response type field. The request ID field is the key to asynchronous handling of RM requests. The RM library locally generates a unique request ID for each request made by an application task, and returns its value to the caller immediately. The request ID is then passed to the RM task as part of the request message. When the resource manager composes a response to a service request, it includes the request ID. The application task then uses the request ID field in the response message to match the response with the corresponding outstanding request. By using unique request IDs, 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 RM when sending a response to this request. The system may provide a default value for this field based on the type of request if the application writer does not specify a value when making a request.

3.2 Resource Management for PVM

PVM implements this framework through the new function `pvm_reg_rm()` which was added in version 3.3 of PVM. This implementation is the result of joint work between our group and

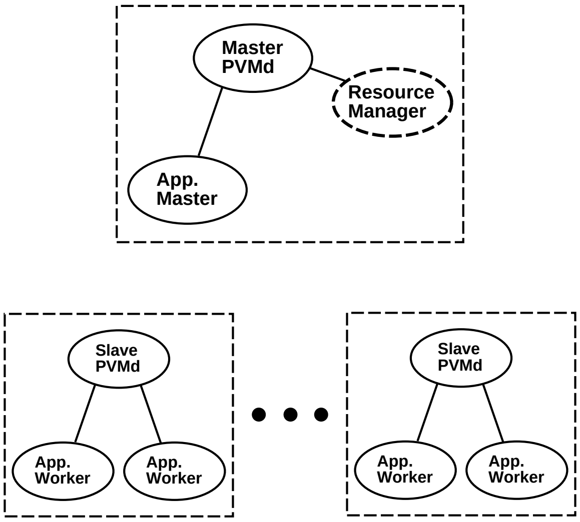

Figure 3:

the PVM development team at the University of Tennessee. We elected to start with PVM for a number of reasons. First, the PVM source code is available which makes experimentation possible. Second, PVM is widely used, and we hope that our work will benefit as many users as possible. Finally, unlike many other parallel programming environments, the PVM application programming interface handles dynamic addition and deletion of hosts and processes. This provides us with a basic set of operations to implement in the RM rather than having to define new primitives before gaining experience with the system.

To move the handling of Resource Management requests outside the PVM daemons, the first step was simply to introduce the notion of resource manager tasks which identify themselves via the new call `pvm_reg_rm()`. Each PVM daemon associates itself with a RM task. This RM task may or may not be on the same host as the daemon. Figure 3 shows the most basic RM configuration possible in which there is only one RM task in the system which is responsible for all of the hosts. When new hosts are added, the PVM daemon on the new host inherits the RM task of the master PVM daemon.

PVM daemons will perform some services for RM tasks which they will not perform for any other tasks within the system. The most important RM only service 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. When a process exits, the PVM daemon sends a message to the RM task containing the exit status and the resource usage of the process. This scheme helps to insulate the RM task from the underlying operating system. No matter what OS is running on a host, the RM can start a task and receive status information simply by sending and receiving messages. As more and more types of hardware (such as massively parallel machines) come to be supported by PVM this service becomes increasing useful in maintaining generality in resource managers.

As part of the process creation service, the PVM daemon passes the identity of its RM task to the new processes. All future requests for resource management services made by the new task will be sent to its RM task. An application may discover the identity of its RM task by calling the new PVM library function get_rm_id(). As in the general architecture described above, the RM portions of the PVM library have been separated into a new RM specific library. The RM library uses get_rm_id() to know where to send requests for service. Functions have been added to the PVM library which set and retrieve a new request identifier field in the message header. Because all PVM RM requests are synchronous, this field is not strictly required for normal PVM operation, but as new, asynchronous primitives are provided by more complex resource managers, it will be needed to match responses with requests.

4 The CARMI programming environment

After modifying PVM to allow resource management to be handled externally, we began using this interface with Condor. All resource allocation, and process creation decisions required by an application were taken over by Condor. It quickly became apparent that the Application Programing Interface (API) defined by PVM was not sufficient for the opportunistic environment provided by Condor.

The principle shortcoming of the PVM API was its synchronous nature. Although the 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, parallel applications must 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 is not sufficient in this environment. Finally, PVM provides 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, does maintain a great deal of information about the resources it controls in order to make scheduling decisions.

To better support parallel applications running under Condor, we have developed the Condor Application Resource Management Interface (CARMI). CARMI provides a set of RM services which are well suited to the opportunistic environment provided by Condor. CARMI still relies on the PVM primitives for application communication, and CARMI also continues to support the PVM resource management API to ease porting of existing PVM applications into CARMI.

4.1 CARMI Services

As suggested by the framework described previously, CARMI presents an asynchronous API in which services are requested via procedure calls which return a service *request identifier*. The return of the request identifier indicates that this 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. 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 likely require the application to poll continuously for each sort of event separately. The services provided by CARMI fall into three categories: request management, resource management, and task management.

4.1.1 Request Management

The request management portion of the CARMI API consists of only 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 query whether the request associated with a particular request identifier 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.

4.1.2 Resource Management

To perform any sort of resource management operations, a method of naming and describing resources is required. For PVM, a resource name simply corresponds to its host name, and the only description provided is a simple architecture string. These methods are not sufficient for CARMI because machine names mean little in an opportunistic environment. Because of this, more extensive descriptions are required. Fortunately for CARMI, Condor already presents a solution to these problems for the purposes of making resource allocation decisions.

Under Condor, every resource is represented by a *classified ad* which describes the resource in detail. A classified consists of an arbitrary collection of keyword, value pairs. Typical keywords which could be found in a classified are "HostName," "Architecture," "Operating System," "Memory," and "MIPS." The set of attributes advertised by a machine is extensible, and can be customized by a system administrator. Because Condor already collects this information, it seems natural to make this information available to applications via CARMI. Because classifieds contain so much information, applications may be able to use them to make decisions at run-time which may impact the overall performance of the application. For example, a data parallel program may assign a larger portion of an array to machines with larger memory or faster processors. The CARMI interface to classified ads allows an application to retrieve the classified for any resource allocated to it. Once it has retrieved the classified, the value associated with any keyword may be queried, or the list of all the keywords defined in the classified may be obtained.

For CARMI to allocate resources to an application, it must know what type of resources are needed. To do this, CARMI provides the resource class abstraction. All resources used by an application must be a member of a resource class which has been defined by the application. These definitions are made at the time the job is submitted, but in the future it will be possible to define new classes at run-time as well. For the purpose of allocation decisions, all resources within a class are considered to be the same.

Resource classes are defined using constraints in the same way that a user specifies the desired resource for a sequential job in Condor [6]. A resource is considered to be a member of a class if the constraint associated with the class evaluates to true when the keywords' values in the resource's classified are substituted. Sample constraints are shown in Figure 4. These constraints have proven to be very powerful for allocating resources to sequential jobs in Condor, and they are proving to be even more valuable when dealing with heterogeneous resources for a parallel application.

Resources are allocated to applications as the result of requests made at submit time or

```
(a) Arch == "ALPHA" && OpSys == "OSF1"

(b) (Arch == "HPPA" && OpSys == "HPUX9") &&
    (Memory >= 32 && Dedicated == TRUE)
```

Figure 4: Sample Resource Class Definitions. In example (a) any machine with architecture "ALPHA" running "OSF1" as its operating system will be a member of this class. A machine in class (b) must be an HP Precision- Architecture running version 9 of HP-UX. It also must have at least 32 Megabytes of memory, and have an attribute "Dedicated" set to TRUE.

at run-time. For each of the classes defined when the job is submitted, the user may specify a minimum and maximum count of desired resources in the class. CARMI will start the job when at least the minimum count is available, but will not allocate more than the maximum count before start-up. If the maximum count is not available when the job is started, CARMI will continue to try to allocate resources up to the maximum level while the job is running. An application can request that it be notified as these resources are added.

The CARMI API also provides a way for applications to request new resources at run-time. The user must specify the class identifier, and the number of new resources desired. When the resources have been allocated, the request is considered complete, and the application receives the notification message.

Finally, since CARMI runs in a dynamic (and often opportunistic) environment, services are provided for receiving notification when resources are lost, suspended or resumed ({`delete, suspend, resume`}`_notify()`). 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 user 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 and application processes which have requested notification will be notified.

4.1.3 Task Management

The last type of service provided by CARMI allows applications to make use of resources which have been allocated to them by creating processes to run there. 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 which is not available on a resource where the process will be executed is automatically transfered. When a process exits, a notification message will be sent to the process' parent giving the exit status of the process.

4.2 CARMI Implementation

Figure 5 shows the architecture of our current implementation of CARMI. 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 a process, called the *shadow*, on the machine where the job was submitted. A different shadow executable can be associated with different *universes* of jobs. CARMI is one of the Condor job universe, so the CARMI shadow process will be started by Condor. Condor also provides the shadow with the list of machines

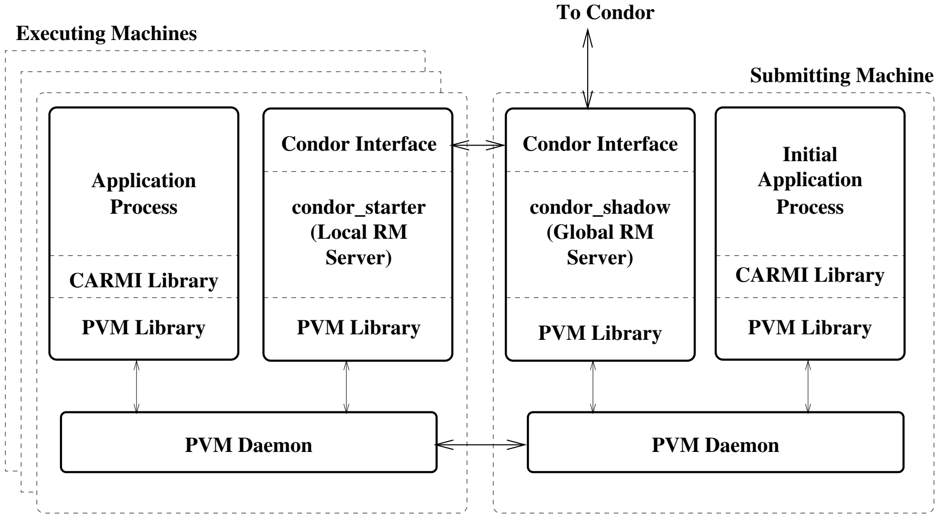

Figure 5: CARMI System Architecture

allocated to the job.

Each machine within a Condor pool runs a daemon called the *startd*. The startd is responsible for monitoring the state of the machine, generating its classified ad, as well as providing access to the machine for running applications. The shadow, therefore, contacts the startd on each allocated machine to request access. Assuming the startd still believes the machine is available, it will in turn create a *starter*. Like the shadow, the starter is specific to the job universe. The starter's responsibility is to start application processes on the machine, and monitor them for completion. Additionally, the starter, along with the shadow, will move executable files to the machine as needed.

PVM requires that a daemon process be running on every machine. To accomplish this, the shadow starts the daemon on the submitting machine, and requests each starter to create a daemon on their machines. The shadow also registers itself as the resource manager process for the daemon it starts. Likewise, the starters become the resource managers for their daemons. The shadow, though, is considered to be the global resource manager for the system. Next, the shadow starts the initial process for the parallel application which was specified by the user when the job was submitted to Condor.

The shadow services all RM requests and, after servicing the request, responds to the process from which the request originated. When a starter receives a RM service request, it forwards it to the shadow via a PVM message. When a starter receives a response, it simply forwards the response onto the local process which initially made the request. Resource allocation requires interaction with the Condor scheduler which is responsible for allocating machines to applications. The shadow translates run-time allocation requests into the format used to describe a Condor job's resource requirements. By translating the request into a format already understood by Condor, we completely avoided making changes to Condor in order to support CARMI applications.

5 Extensions to CARMI

While CARMI provides a powerful environment for parallel programming in an environment with dynamic resources, users still must explicitly handle resources coming and going at run-time. One approach to parallelism which works well in this 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 to process. 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 as they come and go. 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 [7] which also helps master-workers applications adapt to dynamic resources. Piranha, however, is restricted to working with the Linda [8] tuple-space based communication environment where as WoDi is intended to work in a general message passing environment.

An additional benefit to users is that WoDi can monitor the history of both the resources it is using and the work it is distributing to make intelligent work assignments. As we gain more experience with the system, WoDi will do a better job in making these decisions. By using WoDi, users benefit from this experience.

5.1 WoDi Services

To initialize a program using WoDi, the master process must provide WoDi with 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 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.

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 which in turn sends this message on to a worker process when it becomes available. When the worker completes the work step, it sends a result message back to WoDi. WoDi records that the work step has been completed, collects statistics, 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 available worker. In this way, WoDi guarantees that the result of every work step will be received by the master exactly once.

We have found that in some master-workers applications, work steps come in groups, and all of the results from one group must be calculated before the next group can be started. We refer to this group 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. WoDi applications may specify the beginning and ending of a work 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 performance data for each of the resources available to it. Finally, WoDi can send a benchmarking work step to each new worker process after it is created. The benchmarking approach provides the most accurate indication of a resource's performance because it is measured on the application itself. 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, this method is prefered.

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. Therefore, response time for a cycle cannot be smaller than the time required to compute the longest step. If the application requests, WoDi will run 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 user's initial request for a desirable resource level.

Figure 6 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 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 signalled when all cycles have been completed.

5.2 WoDi Implementation

Figure 7 shows the 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

```
main()
{
  buf = pack_initialization_data();
  num_classes = 2;
  class_needs[0] = 10;
  class_needs[1] = 5;
  wodi_init(buf, num_classes, class_needs,
            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 6: Sample WoDi master program

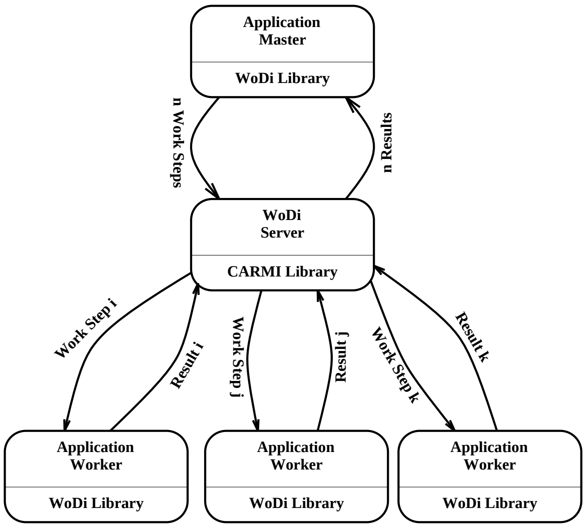

Figure 7: WoDi Architecture

WoDi server must be linked with the CARMI library. Application processes do not make any direct use of CARMI services.

Implementing WoDi in a server process, as opposed to entirely within a library, potentially increases 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 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 system scheduler to adapt to this situation. With one shared server no juggling of resources is needed leading to higher utilization.

5.3 CoCheck: Checkpointing for PVM

The principal mechanism used by Condor to insure jobs progress in spite of resource reclamations is checkpointing [9], [10]. By checkpointing sequential processes, Condor is able to restart an application from the point it was forced to vacate a resource due to its owner returning. Checkpointing could serve a similar purpose for parallel applications, but checkpointing of parallel applications is significantly more complicated. We have addressed the problem of checkpointing parallel applications, in collaboration with the Technical University of Munich, in a system called CoCheck (for Consistent Checkpointing) [11].

CoCheck was designed to provide checkpointing and process migration services to PVM programs. The principal problem to be overcome by CoCheck, or any checkpointing system for communication processes, is to insure that no messages are lost as a result of a checkpoint. CoCheck accomplishes this by first synchronizing all processes in a parallel application, and insuring that the communication system is drained of any messages. When CoCheck is sure that no more messages are outstanding, it independently checkpoints or migrates the desired process(es) using the single process checkpointing mechanism developed for Condor.

The checkpointing and migration facility of CoCheck has been integrated with CARMI, and new API calls have been provided which permit an application to checkpoint or migrate any of its processes. The need to checkpoint individual processes due to resources being reclaimed by their owner can also be detected, and checkpoints or migrations can be triggered without intervention of the application.

6 Experience

Our experience with CARMI has progressed as the system has been developed. Unfortunately, one of our biggest obstacles has been finding applications which can adapt to an opportunistic environment. Using purely CARMI and PVM primitives, we have been able to show the large amount of compute cycles available to such applications by running a synthetic test program. The development of WoDi helped remedy this situation because it made it much easier to run master-workers type applications in this environment. Using WoDi we've been able to perform some "real science."

6.1 Experience with CARMI

During development of CARMI, we used a synthetic application which attempts to acquire as many machines as possible while competing with owners and other jobs submitted to Condor. When a new host is received, a worker process which repeatedly loops for a number of iterations specified by the application master 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 1 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 which was not handled asynchronously by this application was creating a new process. 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. The suspended time column shows that hosts allocated to the computation spent slightly more than 15 hours in the suspended state. Accounting for the time lost due to suspends, the job was able to utilize 97% of the CPU time allocated to it. Finally, the last column shows that over the course of the run, the job was scheduled on 161 different hosts. This number corresponds to the number of machines which were reclaimed by owners during the run. An important result which table 1 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.

This run used machines in two classes: older, slower MIPS based machines and relatively newer and faster machines with Sparc processors. On our application, the Sparcs are a little more than two times faster than the MIPS machines. Unfortunately, we have fewer Sparc machines, and the demand for them is much greater than for the MIPS machines. To utilize as many Sparc

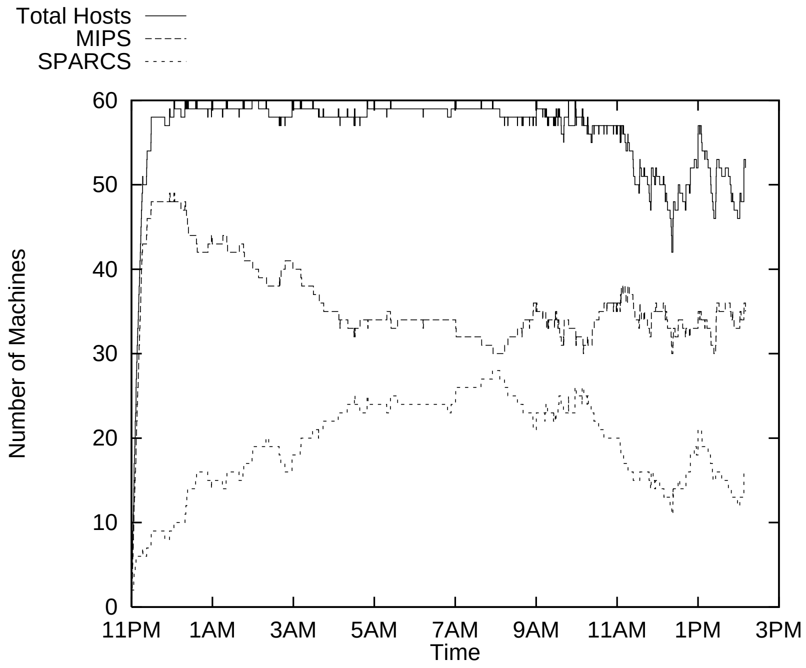

Figure 8: Host Occupancy

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 1: Resource utilization by a synthetic CARMI application

machines as it can get, 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, it will be replaced by the first available machine in either class. Because MIPS machines are plentiful in our pool, the application maintains nearly the maximum number of machines throughout the run. Figure 8 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, we lost Sparcs, but there were also no MIPS machines available to bring us back up to our desired level of sixty.

6.2 Experience with WoDi

Among the applications which we are running with WoDi is a materials science application which is designed to predict the properties of new materials based on first principles [12]. This application was originally written using stand alone PVM and a master-workers approach to parallelism. Because it was written using PVM, it assumed a constant allocation of resources. However, it proved quite easy to convert the PVM communication calls into WoDi requests, and by doing so the application was ready to run in a dynamic environment.

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

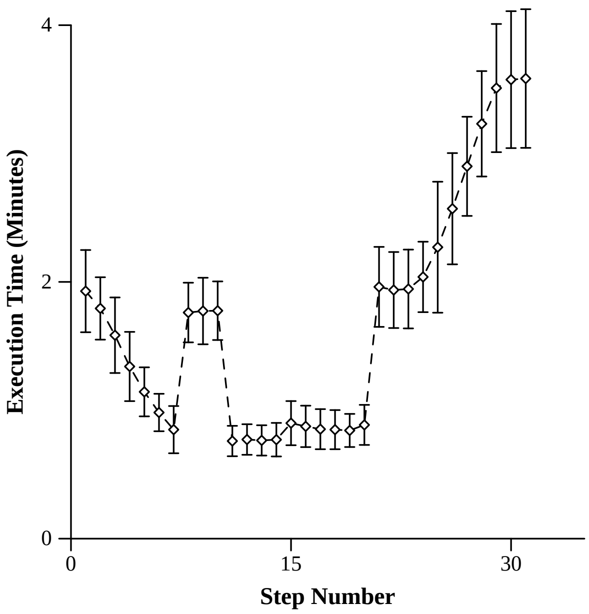

Figure 9: Average time and standard deviation by step number

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 9 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.

6.2.1 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. Occupancy is viewed as the cost of running an application. 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. 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 10 and 11 show the theoretical execution time and efficiency for different resource levels using the distribution from Figure 9. To generate these simulated results, we assume that all resources are identical and that none are lost during a cycle, and that work steps are given to processors using the greedy approach described in [13] in which the longest work steps are distributed first. Each step is considered to be of constant length equal to its average. These results show that in this ideal environment, there is no further benefit to execution time beyond 16 processors. In a real, dynamic resource environment, however, it is not possible to select such an ideal point because the behavior of resources will greatly influence performance. The user is therefore left with the challenge of the cost versus execution time

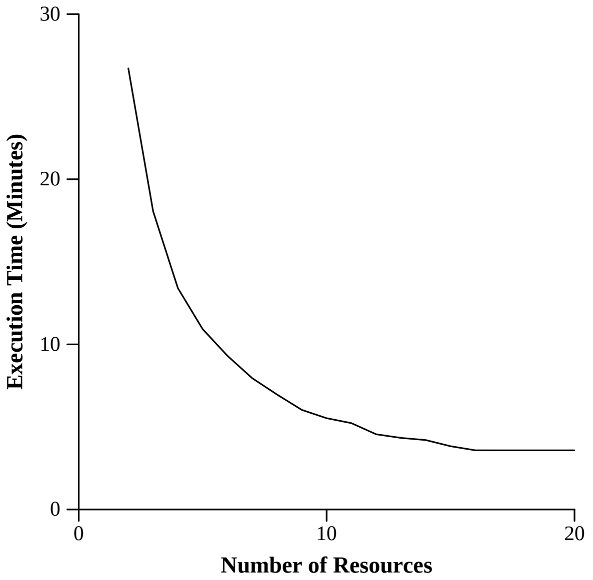

Figure 10: Execution Time for one work “cycle”

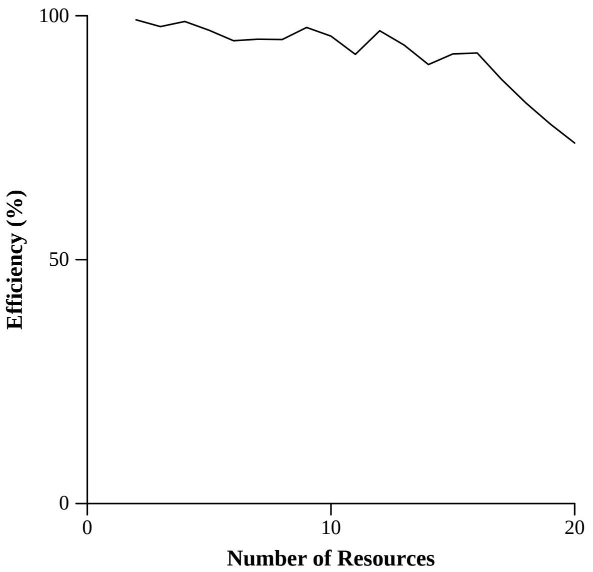

Figure 11: Efficiency for one work “cycle”

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 2: The effect of ordering

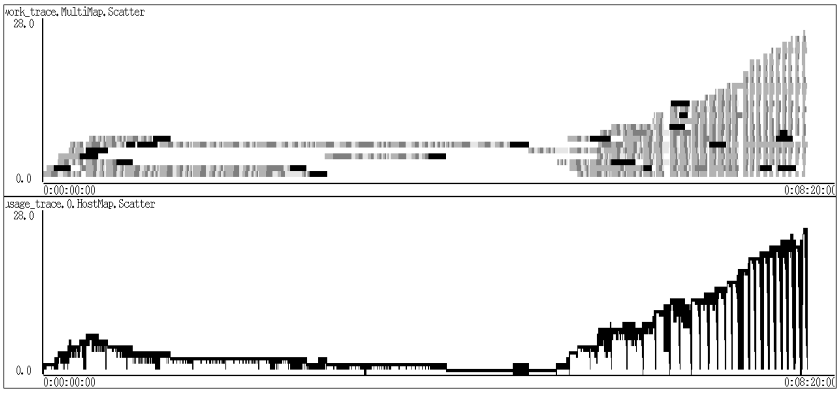

Figure 12: Resource utilization for one run of the materials science application

trade-off.

6.2.2 Work Step Ordering

WoDi's primary decision making responsibility is ordering of work steps. Previously, we assumed a greedy work step distribution algorithm [13], and this is exactly what WoDi employs. Table 2 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 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 roughly 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.

6.2.3 Adaptability

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. Figure 12 demonstrates how WoDi is able to adapt to changes in resource availability. The lower graph shows how many resources are available to the application, and the top graph shows how each processor is used. In both graphs, time goes along the horizontal axis, and the number of processors is on the vertical axis. 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

	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 3: Summary statistics for 61 runs of the materials science application

lots of black appears resources are being wasted. On the upper graph, each gray block indicates one work step being completed. White spaces occur either when resources are unavailable or when resources are idling due to synchronization at the end of a cycle. A black box indicates the loss of a resource. At start-up, the application was able to quickly acquire seven machines, but was then unable to acquire additional machines to replace those which had been reclaimed by their owners, so it 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.

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 users as well as owners for access to resources. During the day, competition from both sources often made it difficult to acquire machines even though our pool contains around 90 machines in the desired class. At night, competition from owners is very small, and we are only competing against other Condor users, so resources are more plentiful. Table 3 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 13 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, occuring 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

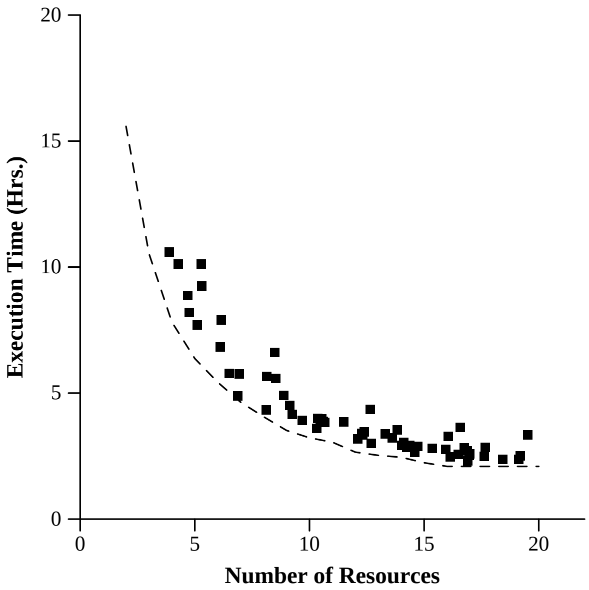

Figure 13: Theoretical and Actual Execution Time and Number of Resources

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 users.

7 Conclusions and future work

The advent of workstation clusters has encouraged researchers to find many different ways to exploit their power. Many innovative systems have been developed for this purpose, but no single system seems to provide all the needed services to fully utilize this environment. For this reason, we've developed a method of interfacing message passing and resource management systems with the goal of providing the best services of both environments to users. This interface has been implemented using PVM for communication, and Condor for resource management. Further refinement with this combined system lead to CARMI which provides a resource management programming interface which is appropriate to this environment. Using CARMI, we've been successful in using a large cluster of privately owned workstations for some types of parallel processing.

One of our primary goals for CARMI is to run a larger number and different types of applications. The ability to checkpoint and migrate applications via CoCheck appears to be a key tool for broadening the types of applications to be run with CARMI. Indeed, automatic detection of the need to perform migrations can be performed, so applications need not even be aware of resources being lost and regained. CoCheck provides new and interesting problems. First, checkpointing a parallel application requires saving a very large amount of state all at one time. We need to develop strategies for saving this large amount of data without permitting any bottlenecks to form. Further, an application which relies completely on CoCheck to insure progress will not fully utilize new resources which become available, or will not adapt to running on a smaller number of resources when the system is heavily loaded. Therefore, we need to investigate ways to use the services provided by CoCheck to help applications adapt.

Acknowledgements

We are grateful to the PVM research group at the University of Tennessee and Oak Ridge National Lab for being receptive to our ideas, and helping to make the the resource management framework a standard part of PVM. We also wish to thank William Shelton of Oak Ridge for providing us with the materials science application used with WoDi.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

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

[9] 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.

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

[11] G. Stellner and J. Pruyne, "Resource management and checkpointing for pvm," in *Proceedings of the 2nd European Users' Group Meeting*, Sept. 1995.

[12] 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.

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