Parallel Processing on Dynamic Resources with CARMI

Jim Pruyne Miron Livny

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

Abstract

In every production parallel processing environment, the set of resources potentially available to an application fluctuate due to changes in the load on the system. This is true for clusters of workstations which are an increasingly popular platform for parallel computing. Today's parallel programming environments have largely succeeded in making the communication aspect of parallel programming much easier, but they have not provided adequate resource management services which are needed to adapt to such changes in availability. To fill this need, we have developed CARMI, a resource management system, aimed at allowing a parallel application to make use of all available computing power. CARMI permits an application to grow as new resources become available, and shrink when resources are reclaimed. Building upon CARMI, we have also developed WoDi which provides a simple interface for writing master-workers programs in a dynamic resource environment. Both CARMI and WoDi are operational, and have been used on a pool of more than 200 workstations managed by the Condor batch system. Experience with the two systems has shown them to be easy to use, and capable of providing large numbers of cycles to parallel applications even in a real-life production environment in which no resources are dedicated to parallel processing.

1 Introduction

The principle goal of a parallel application is to execute as quickly as possible. To do this, the application must effectively exploit the capacity of the resources it can obtain from the system scheduler. It has to balance its urge to mobilize as many resources as possible with the cost associated with using them. In a multi-user, production environment, it is very unlikely that a parallel application can make a-priori assumptions on the availability of system resources before or during its execution. The number, type, and capacity of the resources that the scheduler of such a system is willing to allocate to a given application depend on the current job-mix in the system. Since both queuing time and execution time contribute to overall response time, a parallel application must be prepared to dynamically adjust to fluctuations in the availability of these resources. To do so, it needs means of interacting with the system scheduler at run time and utilizing resources that are allocated to it by the scheduler.

Heterogeneous clusters of workstations are an example of such a dynamic processing system. In recent years, clusters of workstations have become an increasingly popular source of computing resources for sequential and parallel applications. They provide a good cost performance ratio, have improving networking technology (e.g. ATM), and are supported by a range of parallel programming environments. In order to provide cycles to computationally intensive applications, workstations can either be dedicated to this task, or an opportunistic approach can be used. Opportunistic clusters strive to make the spare compute cycles on desktop machines available to sequential and parallel applications [1]. Large opportunistic clusters that consist of a heterogeneous collection of hundreds of desk-top workstations are commonly found in today's academic and industrial settings. By their very nature, these clusters are highly dynamic since they are based on a coexistence between the batch environment and the workstation owners who can regain control of their workstation by a single keystroke.

Whether a dedicated or opportunistic cluster is being used, it is vital that parallel applications have access to Resource Management (RM) services which allow them to dynamically allocate, exploit and query information about the system resources. Currently available parallel programming environments for workstation clusters, such as PVM [2] and P4 [3], do not provide adequate RM services. They either rely on the user to decide a-priori when and how to allocate resources or provide very simplistic algorithms for making RM decisions. To address this deficiency of parallel programming environments, we have developed a framework in which new RM services can be easily added to an existing parallel programming environment. 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 [4]. CARMI has been operational for more than six months and has been used by a number of real-life applications.

The remainder of the paper is organized as follows. In the next section, we describe CARMI and outline its main features. Section 3 discusses WoDi, a high-level interface based on CARMI designed for making writing master-workers parallel applications easy. Experience using CARMI and WoDi on a production cluster that consists of more than 200 workstations is described in section 4, and we make conclusions in section 5.

2 CARMI

The Condor Application Resource Management Interface (CARMI) provides services for writing parallel applications in an environment with dynamic resources. CARMI, therefore, allows an application to exploit new resources which become available at runtime, and aids an application in detecting and managing resource loss. These services can be used in a dedicated environment to utilize resources which are freed by other applications, or to allow a scheduling mechanism to revoke resources from a running application. In an opportunistic environment, CARMI permits an application to grow to new resources as they become available, and cope with resources being reclaimed by their owner.

The first implementation of CARMI uses the message passing capability of PVM [2] for communication among application processes, and between an application process requesting a service, and the CARMI implementation itself. PVM was selected for a variety of reasons. First, PVM is widely used, so we are able to support a large number of applications under CARMI. Second, the PVM source code is available which makes experimentation possible. Finally, PVM supports a dynamic resource environment. The existing PVM implementation already supported resource addition and deletion, and process creation at runtime. We have worked closely with the PVM research group during the development of CARMI, and all of the changes required are integrated into the standard release of PVM as of version 3.3. The approach used by CARMI to support PVM applications is also used by other resource management systems including LSF [5] and IBM's LoadLeveler [6].

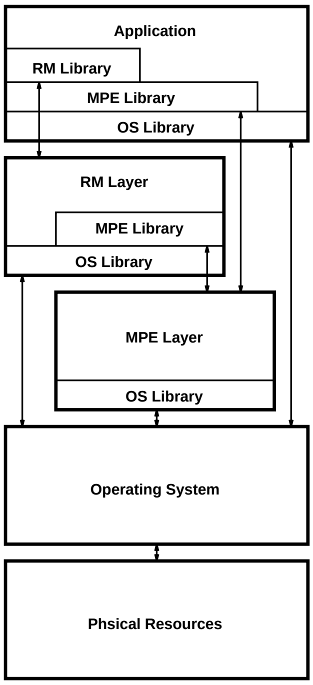

Figure 1: Layered Framework of CARMI

CARMI uses resource allocation and access services provided by the Condor distributed resource management system [4]. Condor is an opportunistic, batch system which schedules applications on idle workstations. When an owner returns to a machine used by Condor, it removes all processes it has started there. Since Condor can manage a large pool of machines, we have an opportunity to run highly parallel applications using CARMI.

2.1 System Design

CARMI is built using a general framework for implementing resource management services which is described in [7]. In general, this framework migrates handling of RM service requests out of a monolithic message programming environment (MPE), and into external RM server processes. These RM server processes, along with an RM library which is linked into each application process, are provided by the RM developer. Every application process is assigned to one of the RM server processes. All RM service requests made by a process are sent to this RM server. Systems built using this framework have a logical layering as shown in Figure 1.

By using this layered approach, we are able to provide support for new MPE's simply by changing the communication primitives used in the RM library and in the RM implementation layer. In general, this approach allows an arbitrary collection of RM services to be implemented by providing a custom RM library and RM implementation layer. The layered design also allows us to effectively leverage existing communication and resource management systems.

2.2 CARMI Services

The set of services available to an application are specified in an Application Programming Interface (API). The goal of a resource management API, and CARMI in particular, is to allow an application to allocate and use as many compute cycles as possible. When deciding what services to make part of an API, we must balance the desire to provide a powerful set of services and ease of use. A list of the services provided by CARMI are given in Appendix A. CARMI tries to achieve this balance by using an asynchronous interface.

CARMI presents an asynchronous API in which services are requested via procedure calls which returns 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 the same communication mechanism used among processes of the user's application. This notification carries a request identifier so the application can easily match the notification with the request it corresponds to. Using the same mechanism for RM service events and application events allows the application to use a single interface for dealing with events of either type. Using a separate notification mechanism for completion of RM service requests would likely require the application to continue polling for each sort of event separately.

The procedure call interface is familiar to application programmers, so they should be comfortable using it. The asynchronous completion, though, is unique and has a number of benefits. First, it allows an application to cope with the fact that the time required to service an RM request is unpredictable. For example, a request for new resources may be fulfilled quickly when resources of the desired type are available, but may take an arbitrary amount of time when all resources are in use. A synchronous interface to resource allocation services would therefore require blocking the application program for an unpredictable amount of time. Second, the asynchronous interface allows an application process to have any number of requests outstanding at one time. This allows complex operations to be composed of collections of simple requests which are outstanding simultaneously. For example, the asynchronous API allows an application which can use a variety of resource types to have requests for all types outstanding at the same time. Making this type of request with a synchronous API would require a single function which permits arbitrary resource collections to be requested at once. Such a function would likely be complex and difficult to use. Finally, an asynchronous API provides a natural way for applications to be notified of asynchronous events such as resources being revoked.

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

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

Figure 2: 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.

2.2.1 Resource Classes

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 to CARMI, 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 logical expressions in the same way that a user specifies the desired resource for a sequential job in Condor [8]. To determine if a given resource is a member of a class, this logical expression is evaluated against a set of attributes advertised by the resource. Examples of the attributes advertised by resources include the instruction set architecture (processor family), the operating system, the amount of memory, and performance measures (as calculated by the Dhrystone and C-LINPACK benchmarks). The set of attributes advertised by a machine is extensible, and can be customized by a system administrator. Sample expressions are shown in Figure 2. These expressions 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.

2.2.2 Resource Handling

Resources are allocated to applications as the result of requests made at submit time or 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 (using CARMI function `add_notify()`) 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 (function `addhosts()`). 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 a notification message.

Applications may also request detailed information about resources allocated to them (`get_host_info()`). This information includes all the attributes advertised by the resource which were used for matching the resource to a class. An application may use this information in a variety of ways. For example, two resources may qualify for the same class based on their instruction set architecture and operating system, but may have very different performance characteristics. Using the performance attributes of each resource allows the application to make load-balancing decisions.

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 killed. If the owner remains active, the resource must be vacated and application processes which have requested notification will be notified.

2.2.3 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 (`class_spawn()`). 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.

A useful service in a dynamic environment would be process checkpointing and migration. Unfortunately, supporting these services would require extensive modifications to existing MPEs. If and when these capabilities are available in MPEs, CARMI will support them. We are currently collaborating with a group at the Technical University of Munich to add such a capability to PVM.

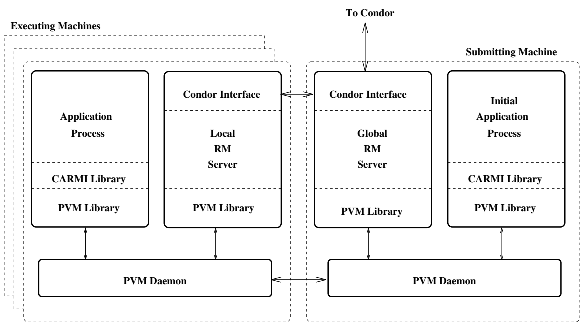

Figure 3: CARMI System Architecture

2.3 CARMI Implementation

Figure 3 shows the architecture of our current implementation of CARMI for PVM. Each machine is required to have a PVM daemon process which handles communication. CARMI starts this daemon on each machine when it is allocated to a job. CARMI also starts the initial application process which is specified at submit time on the machine where the job is submitted. Every machine has an RM server process which serves two functions. One is to handle the Condor related operations on the machine. These include transferring executable files to the machine, suspending, resuming and killing application processes due to owner activity on the machine, and cleaning after a machine is vacated. Second, it receives all RM service requests made by application processes on the host. When an RM server on an executing machine receives a request, it forwards it to the global RM server, which runs on the submitting machine, via a PVM message. The global RM services the request and sends a response back to the process from which it received the request. If the request came from a local RM server, the local server will forward the response on to the application process which originally made the request. Resource allocation requires interaction with the Condor scheduler which is responsible for allocating machines to applications. The global RM 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 avoided making changes to Condor in order to support CARMI applications.

3 WoDi

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 [9] which also helps master-workers applications adapt to dynamic resources. Piranha, however, is restricted to working with the Linda [10] tuple-space 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 and more experience with the system, WoDi will do a better job in making these decisions. By using WoDi, users benefit from this experience.

3.1 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 who 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 being used, WoDi will maintain a history of the computation times of all of 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 is used in place of the user's initial request for a desirable resource level.

Figure 4 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 send to each worker. Next, all of 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.

```
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 4: Sample WoDi master program

3.2 WoDi Implementation

Figure 5 shows the structure of a WoDi program. Like CARMI, WoDi is implemented as a combination of

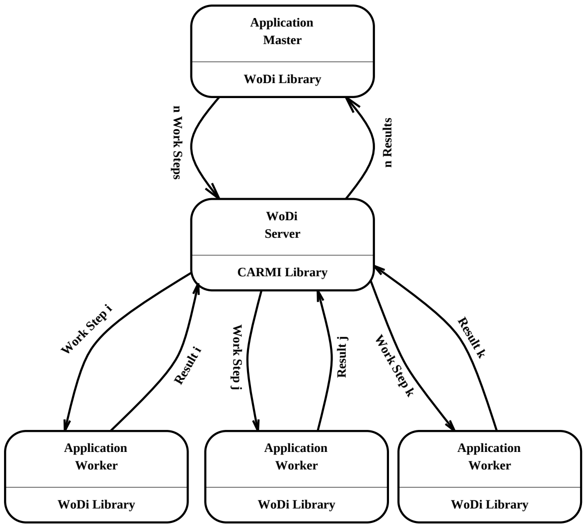

Figure 5: WoDi Architecture

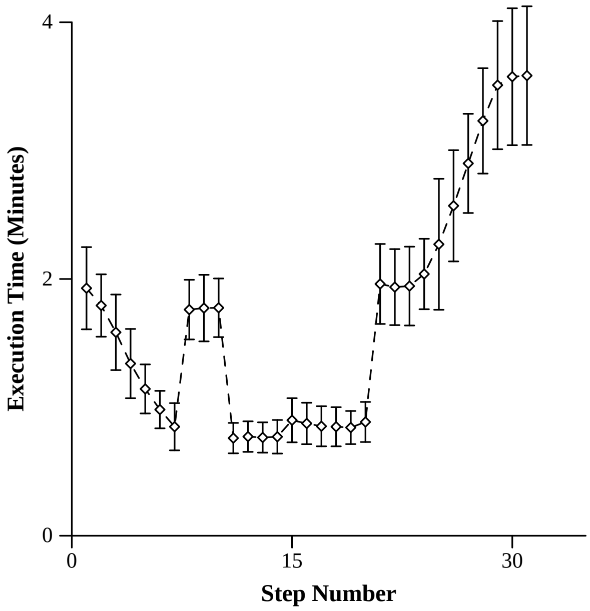

Figure 6: Average time and standard deviation by step number

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.

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 being generated, another might be generating a lot of work. With multiple independent WoDi servers, resources would have to be released by one and allocated to another by the system scheduler to adapt to this situation. With one shared server no juggling of resources is needed leading to higher utilization.

4 Experience

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 [11]. 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 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 6 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 varies relatively little across cycles.

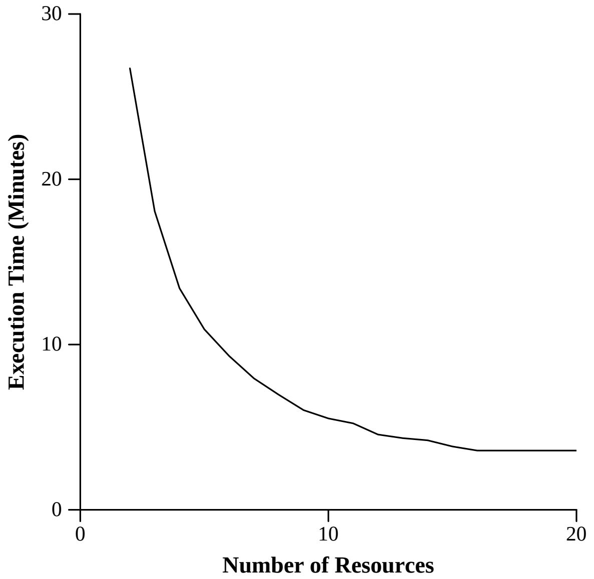

Figure 7: Execution Time for one work "cycle"

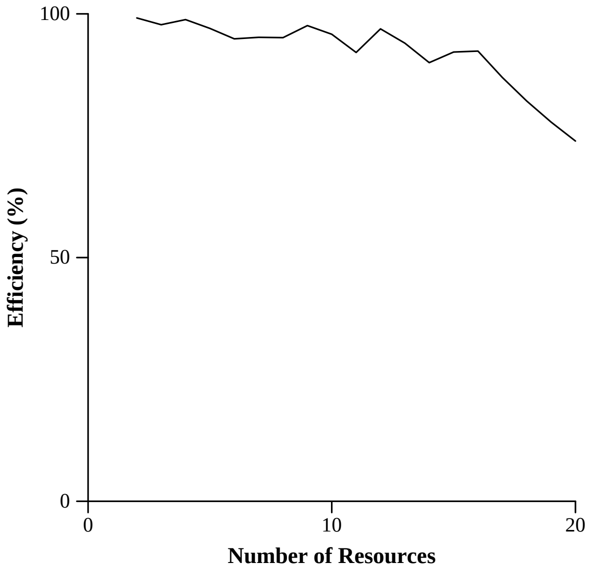

Figure 8: Efficiency for one work "cycle"

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 1: The effect of ordering

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. We do not include queuing time. Occupancy is the total amount of resource time allocated to the job. 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 7 and 8 show the theoretical execution time and efficiency for different resource levels using the distribution from Figure 6. To generate these 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 a greedy approach described in [12] 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 trade-off.

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 [12], and this is exactly what WoDi employs. Table 1 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

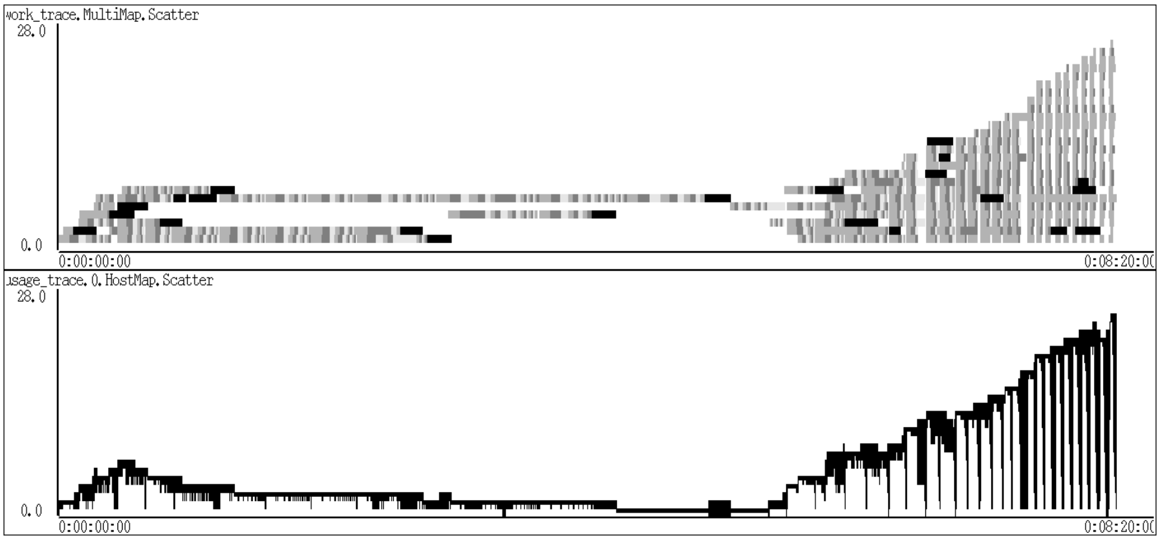

Figure 9: Resource utilization for one run of the materials science application

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.

4.3 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 9 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 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 it was able to get about the time it was started.

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

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 2 summarizes the results of 61 executions.

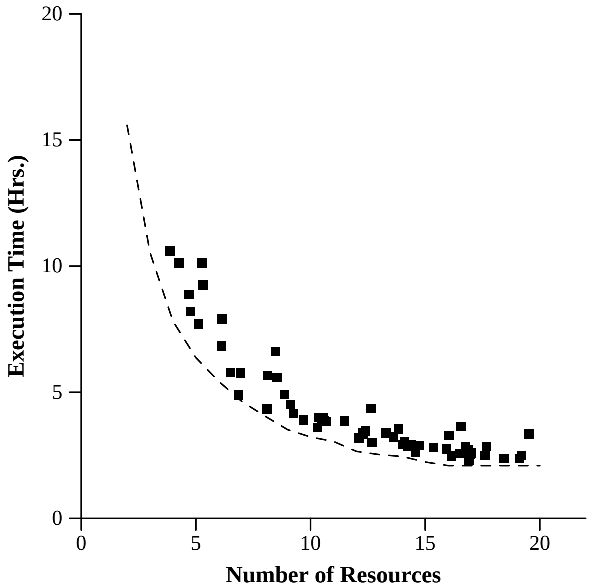

Figure 10: Theoretical and Actual Execution Time and Number of Resources

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 10 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 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.

5 Conclusions

While workstation clusters have become a popular source of cycles for parallel applications, they are not being fully exploited because of the lack of resource management services which allow applications to adapt to this dynamic environment. CARMI provides these needed services, but the user is still required to make their application able to adapt to these changes. WoDi uses the services provided by CARMI to hide the dynamic nature of resources, and makes writing master-workers style parallel programs extremely easy. Both systems are being used in a production environment, and they have shown to be a plentiful source of cycles for real parallel programs. Experimental results have shown that even when resources may be revoked by their owners at least some types of parallel applications can still perform well.

Both CARMI and WoDi are systems which are still under development. The ability to checkpoint and perhaps migrate processes under CARMI would be welcome, but doing so requires significant cooperation from the message passing environment. When MPEs begin to support these operations, CARMI will certainly provide services to make them useful to parallel programmers. CARMI currently can only allocate resources singly to applications, but the structure of some parallel programs causes them to perform significantly better at distinct resource levels. Services which provide resources in groups would therefore be desirable.

More work also needs to be done with WoDi in a number of areas. First, as new services are developed in CARMI, WoDi should exploit them. This includes using a potential checkpoint and restart facility or finding resource plateaus which are desirable for a given work distribution. Managing heterogeneity more intelligently is also a goal. This is particularly important because even in an environment consisting of machines from only one vendor, resources are still heterogeneous due to differences in processor speed, amount of memory and other characteristics. We must better understand how these differences influence performance and how to use heterogeneous resources well.

References

[1] M. Mutka and M. Livny, "The available capacity of a privately owned workstation environment," *Performance Evaluation*, vol. 12, pp. 269–284, July 1991.

[2] 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.

[3] R. Butler and E. Lusk, "Monitors, messages and clusters: The p4 parallel programming system," *Parallel Computing*, vol. 20, pp. 547–564, April 1994.

[4] 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.

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

[7] 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.

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

[9] 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.

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

[11] 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.

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

A CARMI API

The CARMI API adheres to the request protocol set-up by our resource management framework. 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.1 Request Management

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

Cancel the outstanding request with id. The cancel_request message returns false if no request with the given id is outstanding.

A.2 Resource Handling

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

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

```
RequestId  get_host_info(HostId id,
               ResponseTag resp_tag)
returns(CONTEXT machine_context)
```

This function returns all available information concerning a host. The type CONTEXT is borrowed from Condor, and it includes information which characterizes a particular resource. The CONTEXT structure is extensible, but the minimum content includes information such as processor and operating system type.

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

Addhosts requests new hosts in class class_name. Hosts will be added in groups of size increment until the total count hosts have been added. Multiple addhosts messages may be triggered from a single addhosts request as groups of size increment are obtained.

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

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

Delete_notify requests that messages be sent to the application when a host is lost due to an event other than an explicit 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 delhosts() request.

```
RequestId  {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 Task Management

```
RequestId  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)
```

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

Task_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.4 Class Management

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

This returns the definition of all existing classes. A ResourceClass definition is an expression which defines the characteristics a resource must have to be considered a member of a class.

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

DefineClass defines a new resource class, butdoes not request that any hosts in this class be added. The return value is false if a class with the same name already exists.

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

This function removes the class with the given name. RemoveClass fails if any hosts in this class are currently allocated to the application.