Just-in-time Transparent Resource Management in Distributed Systems

TR1998-762
Department of Computer Science
New York University

Arash Baratloo Ayal Itzkovitz* Zvi M. Kedem
Yuanyuan Zhao
{baratloo,ayali,kedem,yuanyuan}@cs.nyu.edu

March 24, 1998

Abstract

This paper presents the design and the implementation of a resource management system for monitoring computing resources on a network and for dynamically allocating them to concurrently executing jobs. In particular, it is designed to support *adaptive parallel computations*—computations that benefit from addition of new machines, and can tolerate removal of machines while executing. The challenge for such a resource manager is to communicate the availability of resources to running programs even when the programs were not developed to work with external resource managers. Our main contribution is a novel mechanism addressing this issue, built on low-level features common to popular parallel programming systems.

Existing resource management systems for adaptive computations either require tight integration with the operating system (DRMS), or require an integration with a programming system that is aware of external resource managers (e.g. Condor/CARMI, MPVM, Piranha). Thus in each case, their support is limited to a single type of programming system. In contrast, our resource management system is unique in supporting several unmodified parallel programming systems. Furthermore, the system runs with user-level privilege, and thus can not compromise the security of the network.

The underlying mechanism and the overall system have been validated on a dynamically changing mix of jobs, some sequential, some PVM, some MPI, and some Calypso computations. We demonstrate the feasibility and the usefulness of our approach, thus showing how to construct a middleware resource management system to enhance the utilizations of distributed systems.

---

[*] New York University & Technion

Contents

1 Introduction **3**

2 Design Goals **4**

3 Architecture **5**

4 User, Job, and Resource Manager Interactions **6**

4.1 Interaction of Users with the Resource Manager 7

4.2 Interaction of Jobs with the Resource Manager 8

5 Mechanisms **9**

5.1 Default Behavior 10

5.2 Using External Modules 11

6 Experiments **12**

7 Brief Overview of Selected Resource Management Systems **15**

8 Conclusions **16**

Acknowledgements **16**

References **17**

1 Introduction

The number of networked machines in many organization is rapidly growing, while frequently many are underutilized or even idle [29, 34, 10, 13, 27]. Such environments are therefore potentially excellent platforms for executing compute-intensive parallel jobs as "guests," in addition to the regular applications for which the machines are designated. A parallel computation most benefitting from such platforms will exhibit one or both of the two characteristics: (1) it can ask statically (at start time) and dynamically (during the execution) for machines satisfying specified characteristics and will incorporate such machines when given to it, and (2) it can continue executing if some machines are taken away from it. We refer to such computations as *adaptive*. Most master-slave PVM [18] programs, self-scheduling MPI [20] programs, and all Calypso [5] programs are adaptive. Indeed, such programs can run on a varying number of machines (as opposed to a fixed number), and remote processes can be added/removed at any time during the computation. In the case of PVM and MPI, the programs must be written such that they are able to tolerate machine removals, whereas for Calypso, this service is provided by its runtime layer.

In order to find appropriate available machines, to allocate them to computations, to help running computation incorporate them, and to remove them as needed, a *global resource management system* (or *service*) is needed. In addition, to be most effective, such a system should not require programmers to be aware of it during application software development—the system should be an invisible service optionally provided to pre-compiled binaries. Furthermore, it should be able to simultaneously manage a variety of computations written using diverse programming systems.

Existing resource management systems such as Condor [25], Utopia/LSF [36, 32], DQS/CODINE [14], PBS [21], and IBM's LoadLever [22] were initially developed to manage sequential jobs. In a recent study performed at NASA Ames Research Center [23], six commercial and research resource management systems were evaluated and it was reported that "the bad news is the confirmation of continuing lack of JMS [Job Management System] support for parallel applications, parallel systems, and clusters of machines." In such systems, the allocation of machines is static and only done at the start of the job execution; and there is no support for adaptive allocation and reallocation of machines to running jobs.

Resource management systems developed for adaptive programs such as Piranha [7], MPVM [8], Condor/CARMI [33], and DRMS [26] are system specific. DRMS uses implementation-specific mechanisms to modify MPI routing tables; others, such as Piranha, MPVM, and CARMI, rely on system-specific interfaces to manage adaptive programs. For example, CARMI depends on the job management API defined by PVM to provide its services. This approach is limited to programming systems that are aware of external resource managers with well-defined interfaces, and requires explicit integration for each new programming system. As we sketch in Section 7, no existing resource management system supports adaptive programs written with different types of parallel programming systems. We present the first such system.

Our main technical contribution is a general mechanism to manage a pool of resources used by a dynamic set of computations, especially benefitting adaptive programs. It is based on interception and redirection of a set of low-level actions common to many popular parallel programming systems. Thus, the approach is not tied to any particular programming system. We built a simple resource management system to validate the feasibility and the utility of the approach.

Our system is unique in providing a set of features previously not available together. These features include:

- **Support for adaptive programs:** Resources are allocated dynamically to parallel jobs. The system provides resources just-in-time if the job's needs grow, and it can reallocate resources among multiple executing jobs to distribute the workload to adhere to the externally specified policies and constraints. Of course, "standard" non-adaptive programs are supported too.

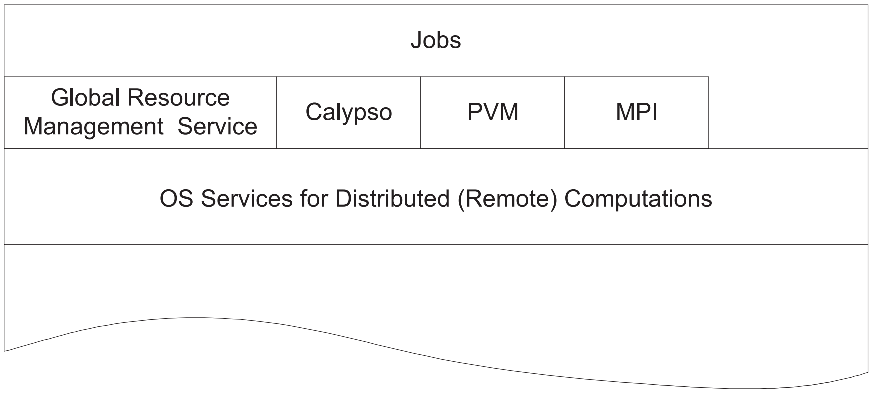

Figure 1: Structure of a global resource manager as a user-level service.

- **Simultaneous support for diverse programming systems:** Support is provided for a mix of computations including sequential, PVM, LAM [30] (an implementation of the MPI standard), and Calypso, running concurrently on a heterogeneous network.
- **System independence:** We treat parallel programming environments as COTS, and do not make any modifications to them. The system transparently supports existing binary executables.
- **User privilege:** The system does not require super-user privileges to run, and the only privilege required for its installation, configuration, and use is `rsh` access to remote machines. Thus, it does not compromise the security of the enterprise.
- **Negligible overhead:** The underlying mechanism imposes a very small overhead while a machine is added to a computation or removed from it. It imposes no overhead at other times.

In the next two sections we describe the goals and then the architecture of our system. The architecture supports interaction of users and running jobs with the resource manager. The specification for this interaction (with parts influenced by the Globus project [11]) is presented in Section 4. Section 5 describes the mechanism our resource manager utilizes to dynamically change the allocation of resources to running jobs. Experimental results are presented in Section 6. We conclude by succinctly putting our work in context of related work.

2 Design Goals

Our goal is to provide a comprehensive resource management system able to dynamically allocate, deallocate, and reallocate machines among multiple competing computations written using different parallel programming systems, such as PVM, MPI and Calypso. We list some key objectives met in our system:

- The use of our resource management system is optional. To emphasize its unobtrusive availability, we sometimes may refer to it as "service," and call it a *Global Resource Management Service*, see Figure 1. A user is free to use or not to use the services at each invocation of a job.
- The service runs on top of completely standard operating system. The current implementation runs on top of (unmodified) Linux 2.0.29. The mechanisms are general and are applicable to other Unix operating systems.

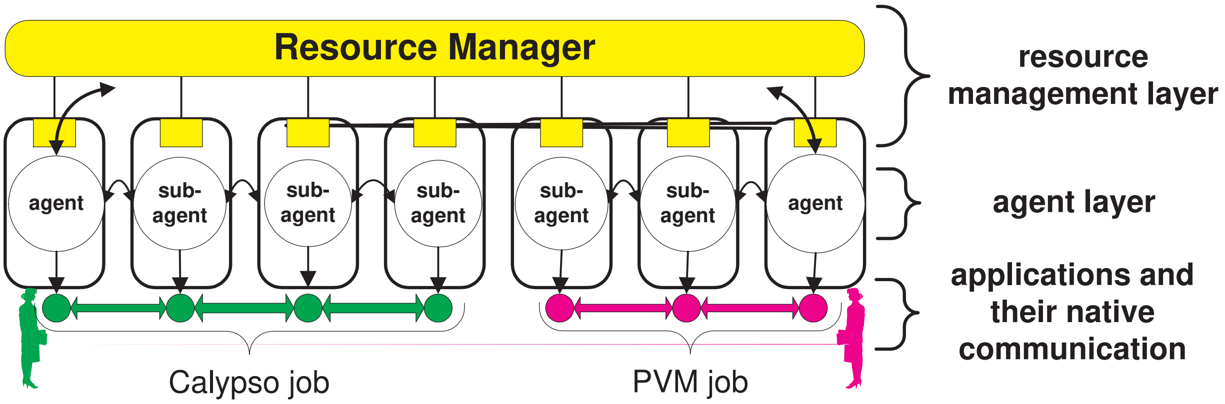

Figure 2: The components comprising the resource management layer and the agent layer

- It treats programming parallel environments as COTS. Unlike others, we do not expect parallel programming environments to be modified or adapted to take advantage of our service. The service can manage executables that were compiled and linked for an environment with no global resource management service. Thus the service is backward-compatible with existing binary executables.
- For jobs that are able to request additional machines during the execution, or able to tolerate removal of machines (fault tolerant jobs), the service can dynamically reallocate resources among executing jobs, thus providing resources just-in-time to jobs.

Of course, in general the policy employed by the resource manager in allocating and deallocating resources to computation needs to be clearly specified. We will not address this here, as we focus on the mechanisms underlying the overall system. However, in order to test the mechanisms we established a policy which, while oversimplifying, can be briefly described. User jobs are divided into two classes, fault-tolerant (such as Calypso), and others. The resource manager tries to "fairly" satisfy the requests of all the jobs, removing machines from fault-tolerant jobs, if necessary. (A very simple example of this will be seen in our performance evaluation experiments, described in Section 6.)

3 Architecture

Our system consists of two weakly coupled layers, the *resource management layer* and the *agent layer*, see Figure 2. Our current research stage focuses on the agent layer and the mechanisms providing its functionality. However, to validate and test these mechanisms and the overall approach we built a rudimentary resource management layer.

The *resource management layer* is comprised of a network-wide resource manager process, and monitoring daemon processes, one on each machine. The per-machine daemons are responsible for monitoring items such as the CPU status, users logged on, the number of running jobs, and the status of the keyboard/mouse. This information is periodically reported to the resource manager. The resource manager initially starts the daemon processes on each participating machine, and restarts them in an event of failure. It is also responsible for deciding how to allocate/deallocate available machines to running jobs. This decision-making process in the current prototype is conventional (is the machine pre-emptable, how long ago was it used by the owner, etc.) and outside the scope of this paper. It is important to note here that although processes

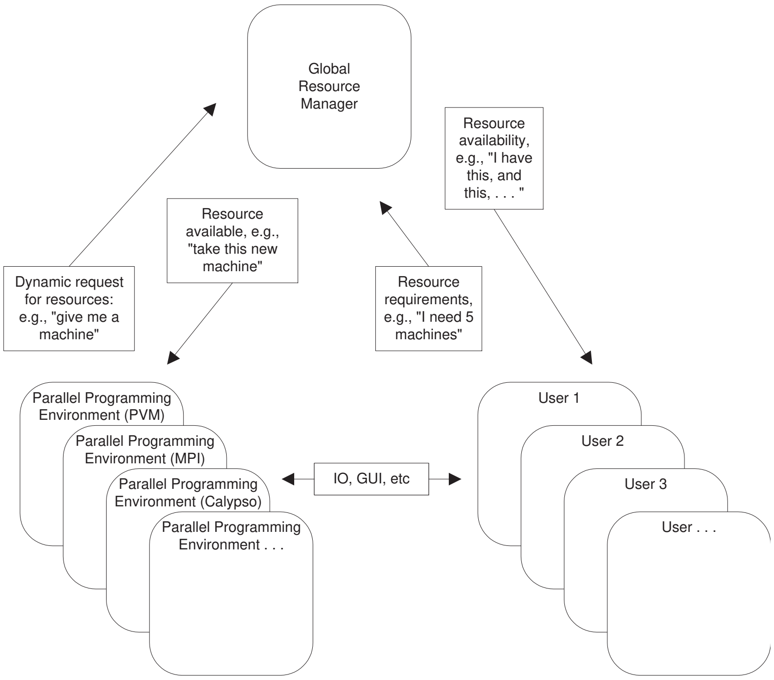

Figure 3: The entities involved in every job execution: resource manager, users, and jobs.

that comprise the resource management layer run at user-level privilege, they are able to manage other user's jobs. We explain the mechanism in Section 5.

The *agent layer* consists of sets of agent processes, each associated with a user job. A user submits a job through a process called the *agent*; a *subagent* runs on each of the remote machines, see Figure 2. In this way, the *agent layer* consists of a dynamically changing set of agent processes. Agents provide to the resource manager the means to monitor and communicate with a running job and also to actively intervene in the execution of a job. In the broader sense, they act as a broker between the resource manager and a job. An important feature of our system is its ability to interact with, and thus support, multiple parallel programming environments through external modules. These modules will be discussed in Section 4.

4 User, Job, and Resource Manager Interactions

In the presence of a system-wide resource management service, one might consider three types of entities in every job invocation: (1) the resource manager, (2) the user, and (3) the job submitted for execution. Figure 3 depicts the three types of entities and the interaction among them. The interaction between users and running jobs, such as IO, is outside our interests here. In contrast, the interactions between the resource manager and the users, and between the resource manager and the jobs are of interest and will be discussed next.

```
specification := request
request := multirequest | conjunction | disjunction | parameter
multirequest := "+" request-list
conjunction := "&" request-list
disjunction := "|" request-list
request-list := "(" request ")" request-list | "(" request ")"
parameter := parameter-name op value
op := "=" | ">" | "<" | ">=" | "<=" | "!="
parameter-name:= "adaptive" | "start_script" | "module" |
                 "count" | "max_time" | "start_time" |
                 "arch" | "environment"
value := ([a..Z][0..9][_])+
```

Name	Default	Semantics
adaptive	false	whether the job is adaptive
start_script	NULL	script to execute after allocation of resources and before invocation of the job
module	NULL	module name to use in communication to the job

Figure 4: Specification language (adopted from Globus) for users to describe resource requirements to the resource management system, and three new parameters to describe adaptive programs.

4.1 Interaction of Users with the Resource Manager

The interaction between a user and the resource management service could be, for example, a specification provided by the user describing the number and the types of resources required for the job. We adopted the Resource Specification Language of Globus [11], and extended the specification to support adaptive programs. See Figure 4 for the specification grammar. Specifically, we added `adaptive`, `start_script`, and `module` parameters for users to describe the nature of their adaptive jobs. For example,

```
& (count<=8) (adaptive=1) (arch="i86Linux") (start_script="my_script")
```

is a request to execute an adaptive (as specified by `(adaptive=1)`) program on up to 8 Linux machines. The `(start_script="my_script")` requests the execution of `my_script` after the resources have been allocated, but before the job is started. Starting scripts are passed the allocated host names as arguments. A typical script could be

```
#!/bin/sh
echo add $* > $HOME/hosts
```

to initialize a programming-system specific hosts file. As an additional example,

```
& (count=4) (module="pvm")
```

is a request to execute a PVM program on exactly 4 machines. When job is submitted along a `module` option, as in (`module="pvm"`), our system assumes the existence of three executables to carry out some of its responsibilities. Figure 5 contains the set of PVM modules.

pvm_grow	pvm_shrink	pvm_halt
`#!/bin/sh` echo add $1 > $HOME/.pvmrc echo quit >> $HOME/.pvmrc pvm >& /dev/null rm $HOME/.pvmrc	#!/bin/sh echo delete $1 > $HOME/.pvmrc echo quit >> $HOME/.pvmrc pvm >& /dev/null rm $HOME/.pvmrc	#!/bin/sh echo halt > $HOME/.pvmrc echo quit >> $HOME/.pvmrc pvm >& /dev/null rm $HOME/.pvmrc

Figure 5: PVM specific module scripts.

4.2 Interaction of Jobs with the Resource Manager

The interface between the resource management service and the job is complex since the two software components are typically developed independently. Resource management systems intended for sequential and batch-like jobs (e.g. Condor, Utopia/LSF, DQS/CODINE, PBS, and LoadLever) do not interact with the job. Thus, to use such systems *the user becomes the interface*: the user first requests resources from the resource management service, then explicitly starts the job on the allocated resources. The set of allocated resources remains fixed for the duration of a computation.

Parallel programming systems such as Linda, PVM, MPI, and Calypso, have a built-in resource manager, though of limited functionality. It is common for this type of resource managers to support (1) the scheduling of parallel tasks among the (already) allocated machines, and (2) interfaces for a job (or a user) to add an *explicitly-named* a machine to the computation. We refer to such functionality as *intra-job resource management*. In contrast to the resource management provided by parallel processing environments, we will use the term *inter-job resource management* to refer to the management of *all the machines* among *all the executing jobs*. The concept of the separation of the two management services was discussed in work by Benevolent Bandit Laboratory [16].

Resource managers for adaptive programs, such as Piranha, MPVM, Condor/CARMI, have *integrated* intra- and inter-job resource managers. The interaction between a PVM job and the CARMI resource manager, for example, is implemented through a system dependent API and communication libraries. This approach, however, will not work for MPI and Calypso programs since they do not support a well-defined job management API.

We chose a novel design to interface intra- and inter-job resource managers, and hence, support multiple parallel programming systems. Simply put, we rely on a set of common features to achieve this goal. We next describe the two interfaces: between the job and the resource manager, and between the resource manager and the job.

Job to Resource Manager Interface

Note that `rsh/rexec` is a common *underlying low-level* command used by various programming systems for adding machines to a computation, though the actual command available to the programmer is likely to be higher-level. For instance, PVM uses the command `add`, see Figure 5. It is easy to detect when a program issues such a low-level command. Thus, we rely on intercepting the `rsh` command issued by the parallel program as a signal to our system that the job is requesting additional resources. The UNIX `rsh` command takes a machine name as an argument. To use our resource management system, a program simply needs to issue an `rsh` command with a *symbolic* host name, indicating a request for additional resources. Our system in turn analyzes the symbolic name for the requirements, (for example, `any_linux` could represent a Linux machine) and allocates an appropriate machine on the network.

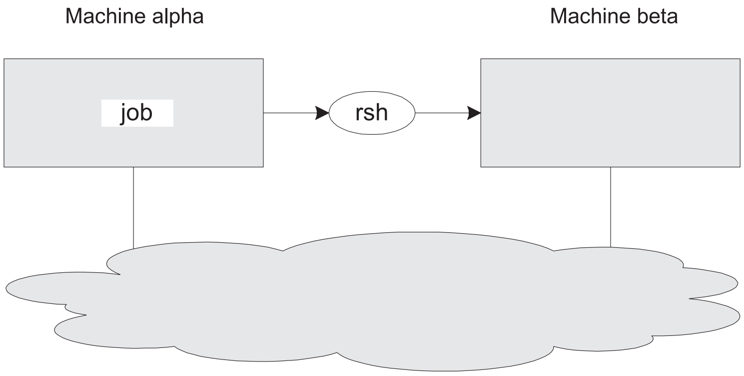

Figure 6: A parallel job running on two machines

Resource Manager to Job Interface

Once our system decides to allocate a machine to a running job, this is then carried out by the *agent processes* (see Section 3) responsible for that job. Oversimplifying for now, the default behavior of the agent process is to replace the symbolic host name with the designated one and to allow the original `rsh` command to proceed. The default mode is appropriate for Calypso and sequential jobs. As will be discussed in section 5.2, PVM and LAM jobs do not allow an unexpected machine to join a computation. In such cases, the agent process relies on a *module* to communicate with the running job, informing it of the machine that has been allocated to it, and "coaxing" the job to accept it. Modules are executable programs (or shell scripts) outside the resource manager. This architecture enables future support for yet undefined programming systems without having to recompile the resource manager.

5 Mechanisms

To fix the context and the terminology, we briefly describe a hypothetical, but representative scenario of how parallel jobs acquire resources *in the absence* of a resource manager. Take the parallel job depicted in Figure 6. The user wants to start the executable named `app` on machine `alpha`, and wants to make available to it two additional machines, `beta` and `gamma`, when needed. The user prepares a file, say `.hosts`, containing `beta` and `gamma`. At the shell prompt at the machine `alpha`, the user types: `$ app <arguments>`. The job starts executing. At some point it needs to spawn a process on another machine. It consults `.hosts`, reads the machine address `beta` and issues the command `rsh beta <args>`. (In the job's source code this could have been a higher level function, but it ultimately was translated into the standard `rsh` command.) A new process is spawned on `beta`. It communicates with a process on `alpha` and the communication stream is established (the cloud below).

If the user chooses to use our system, the agent will cause the second process to be spawned on the machine specified by the resource manager. Schematically, the agent process responsible for that job needs to accomplish the following:

1. "Realize" that the job "would like" to spawn a remote process
2. Advise the resource manager that a new machine is requested

3. Obtain from the resource manager the address of a target machine, which presumably was determined to be "most appropriate"
4. Cause the process to be spawned on the target machine
5. Enable the two processes to establish the communication stream, and
6. "Fade in to the background," so as not to impose an overhead.

We make the following assumptions about the job and the environment in which it is running:

- We are *not* given the source code of the job, only the binary executable.
- The job does not have machine addresses "hardwired" in the executable. This is a natural assumption as otherwise the job is not targeted for execution anywhere other than a specific set of machines, defined at compile time.
- We are able to inform the program of the location of the system-wide `rsh`. This is a natural assumption as this location is specified as part of building the programming system's installed environment, and in many cases, in the user's default path is used.

5.1 Default Behavior

For jobs that have been submitted to our system, we intercept their `rsh` calls and use our own program to handle them. We first examine the job's `rsh` target machine. If it is a specific existing machine we will let it execute, as this indicates that we should not intervene in this case. If the call is targeted to a *symbolic* machine, we will intervene as described below.

To use our system, the user does two things. First, assuming that the user wants to have access to two additional machines, the user prepares `.hosts` file containing two "placeholder" machine addresses. The specific syntax is not important, so let us assume that these are IP addresses that will be easily recognizable as impossible to use (say by policies of the installation). Here, we refer to them as `IP1` and `IP2`. Second, the user submits the application to an agent process by the command: `$ agent app <arguments>`.

The `agent` starts executing and it immediately spawns a child process executing `app`. When `app` wants to add (the first) machine, it consults `.hosts`, finds `IP1` and issues `rsh IP1 <args>`. This `rsh` call is intercepted by us, and realizing that IP1 is a symbolic machine name, we intervene.

We depict the execution as `rsh`*, consult Figure 7. The `rsh` process contacts the `agent` which contacts the resource manager with a request for a machine. Once a machine address is given to it, say `delta`, the `agent` notifies `rsh`* of this machine address. Then `rsh`*, using the "standard" `rsh`, spawns a `subagent` process on `delta`. The `subagent` spawns the appropriate process on its local machine, `delta`. This process contacts the original `app` process on `alpha` and the job continues executing normally.

From this point on (until resources need to be reassigned) there is no interaction with the `agent` or the various `subagent` processes. They remain dormant, and no overhead is imposed by their existence. Interaction with them can come in two ways. First by the job attempting to add a machine. Second by the resource manager if it wants to take a machine away. To take `delta` away from the job, the `subagent` first signals (standard UNIX signals) the child, and if the child does not terminate within a specified amount of time, it terminates the process.

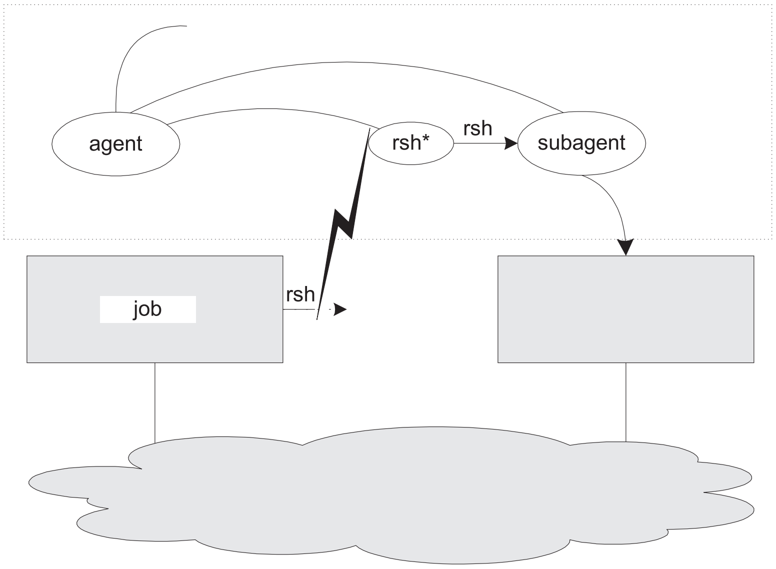

Figure 7: Adding a machine (default behavior).

The above was a simplified discussion of the capabilities of the system. For instance, it is possible to specify to the resource manage what types of machines are requested, using our specification language (if necessary using an appropriate encoding, to maintain compatibility with the syntax the programming system expects).

5.2 Using External Modules

The default behavior described above works for parallel programming environments such as Calypso, for parallelizable tasks like `parallel make`, and for executing sequential jobs remotely. However, some parallel programming environments may have additional standards that need to be followed. We describe the extensible interface in the context of PVM, due to its popularity.

Reviewing the default behavior, we note that the application process on `alpha` attempted to `rsh` a process on a machine it believed to be `IP1`, whereas in fact the process was spawned on `delta`. Generally redirecting the `rsh` goes unnoticed. However PVM jobs perform a check and will refuse to accept a process from a machine different than the one on which they attempted the spawning. To handle this type of situation the resource manager needs to interact with the application, passing it the name of the machine. Below we sketch a PVM specific scenario, see Figure 8.

The user, knowing that `app` was written for PVM, types `$ agent app <arguments> --module="pvm"`. The `agent` starts, and everything proceeds as before until `agent` is given `delta` by the resource manager. It then terminates our `rsh`, depicted as `rsh`* and executes an external script, called `pvm_grow` with the parameter `delta`. This script consists of five lines and shown in Figure 5.

This script in effect opens another PVM console, which realizes that a PVM daemon is already running on `alpha`, asks this daemon to add machine `delta` to the virtual machine, and the console closes. Therefore, the job itself will now issue another `rsh` command, but now with the machine address of `delta`. It is again executed by our `rsh` script, which now knows it is in phase 2, depicted as `rsh`**. Again `agent` is consulted by `rsh`**, `delta` is given to it, and it proceeds as before, a subagent is created, etc. As the consequence however, the machine added to a computation is the same machine the job attempted to add, hence, the

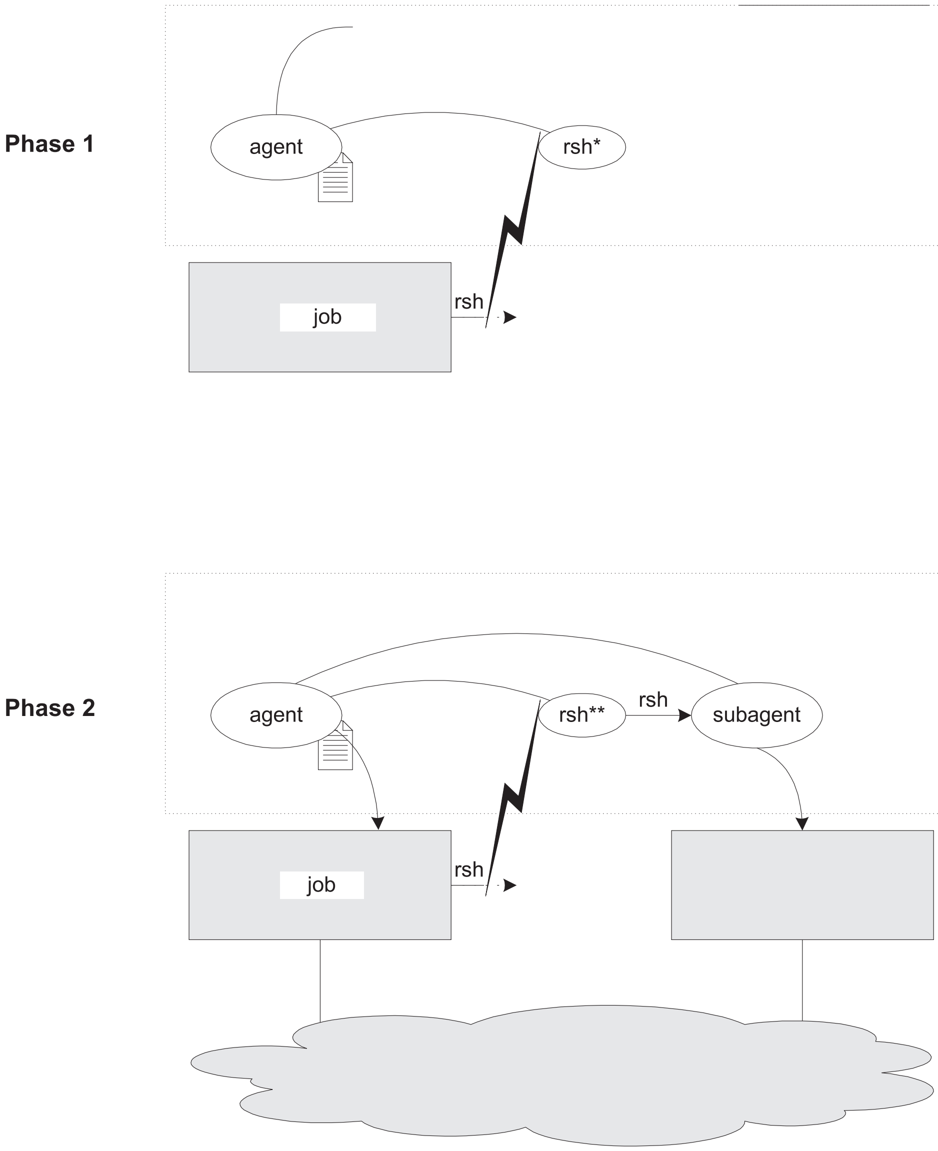

Figure 8: Adding a machine (using an external module).

machine check is passed.

A somewhat more complicated external module is used for LAM, which we do not describe here. Our "plug-in" external module approach makes the design extensible and thus able to accommodate various programming systems concurrently.

6 Experiments

Here we present experiments validating our proposed mechanisms. Experiments were conducted using up to 16 PentiumPro/200 machines running Linux RedHat 4.0, and connected by a Fast Ethernet hub. Reported times are the *median measured elapsed times*, accounting for all overheads. For the duration of this section we use `rsh` to denote the standard remote shell program, and `rsh'` to denote our implementation of remote shell program that together with management layer, agent layer, and external modules provide resource allocation services.

Micro Benchmarks

Our first set of experiments compares the performance `rsh` and `rsh'`. The results are shown in Figure 9.

Operation	Time (sec)
`rsh` *beta* `null`	0.4
`rsh'` *beta* `null`	0.6
`rsh'` ANY `null`	0.6
`rsh` *beta* `loop`	36.9
`rsh'` *beta* `loop`	37.0
`rsh'` ANY `loop`	37.1

Figure 9: Comparison of `rsh'` and `rsh` for executing sequential applications.

We use two sequential applications for this experiment: `null` is a minimal C program having an empty `main()` function, and `loop` contains a tight loop that runs in 34.4 seconds. For this experiment we used two idle machines, *alpha* and *beta*. The remote execution commands of Figure 9 were issued on *alpha*, and directed to execute on *beta*. Thus, the command "`rsh'` *beta* `null`" resulted in executing `null` on a *beta*. The ANY keyword is interpreted by our resource manager as "any available machine," thus the command "`rsh'` ANY `null`" allows our resource manager to choose a machine to execute `null`. But in this particular experiment we limited the available resources our resource manager could see, so it in fact chose *beta*. As the figure shows, the overhead associated with `rsh'` is approximately 0.2 milliseconds—hardly noticeable by users. The small overhead also indicates that replacing the system-wide `rsh` with `rsh'` is plausible, even if some users will never use the features provided by our system.

For the second set of experiments we are interested in the time required to reallocate resources. The results are shown in Figure 10.

Operation	Time (sec)
`rsh` *beta* `null`	0.4
`rsh'` ANY `null`	1.5
`rsh` *beta* `loop`	38.2
`rsh'` ANY `loop`	37.9

Figure 10: Performance of resource reallocation.

Three machines were used in this experiment: *alpha*, *beta* and *gamma*. An adaptive Calypso program was started on *gamma* and grew to use *beta* in addition. Similarily to the previous experiment, the remote execution commands of Figure 10 were issued on *alpha*, and in every case resulted in the execution of a program (`null` or `loop`) on *beta*. In the case of `rsh'`, our resource manager first terminated the Calypso process running on *beta* before satisfying the request. That is, it took a machine away from one job and gave it to another—hence reallocation. The results show that a reallocation completes in approximately 1 second. It is also interesting to note that in the case of `loop` (and other compute-intensive jobs), users experience a faster turnaround time since *beta* is cleared of external processes before executing the job.

Parallel Computations

Operation	Time to acquire 1 machine (sec)	Time to acquire 2 machines (sec)	Time to acquire 4 machines (sec)	Time to acquire 8 machines (sec)
pvm w/ `rsh`	0.5	1.1	2.3	6.7
pvm w/ `rsh'` *host*	0.6	1.4	3.2	7.1
pvm w/ `rsh'` ANY	2.1	3.6	9.1	17.6
lam w/ `rsh`	1.8	1.8	2.1	2.0
lam w/ `rsh'` *host*	2.5	2.8	2.4	2.4
lam `rsh'` ANY	4.2	8.8	16.8	33.5

Figure 11: Performance of our resource manager (using external modules) to dynamically allocate additional resources to PVM and LAM programs.

Here we present the performance of our system when managing parallel jobs. In particular, we test the effect of external modules within the agent layer. As our system managed the jobs, they implicitly used `rsh'`.

We measured the time to acquire the requested number of machines. The results are shown in Figure 11. The operation "pvm w/ `rsh'` *host*" means that the PVM program explicitly named the machines it wanted to use, and "pvm w/ `rsh'` ANY" means that the PVM program left the choice as to which machines to use to our resource manager. The results show that in the case where the machines are explicitly named, our resource manager has a negligible overhead (less than 0.1 milliseconds per machine). This reinforces the fact that replacing the system-wide `rsh` with `rsh'` is plausible, and that this will not be noticeable by users not using the additional features provided by `rsh'`. Allowing our system to choose a machine (i.e, `rsh'` ANY) incurs approximately 1.4 seconds overhead for PVM and 3.5 seconds for LAM programs. This overhead occurs once per machine, and only at startup.

n	Time (msec)
1	1.0
4	3.8
8	8.0
12	11.8

Figure 12: Performance of resource reallocation using PVM and `rsh'`.

Next we report the time to reallocate resources for parallel jobs. The setting was as follows. An adaptive Calypso program was initially running on every machine on the network. We then started a PVM program several times, and each invocation built a different size (denoted by n) virtual machine. To satisfy PVM programs request, our resource manager first had to take some machines away form the Calypso job. Figure 12 reports the measured times from the invocation until the resources were made available. The results show that the reallocation completes in approximately 1 second per machine (which is consistent with our second experiment), and that this number scales linearly to at least 12 machines.

In our final experiment we measure the utilization factor of networked resources in a dynamic environment where a jobs are dynamically introduced. The setting was as follows. An adaptive Calypso program initially utilized eight machines. Then repeatedly, we introduce a sequential program that runs for t minutes, where t is chosen uniformly from the interval [1,10], and wait 100 seconds after each run. After five hours, we measured the idleness (total amount of time when a machines was not running a job) of the machines to be less than 1% (approximately 4 milliseconds to allocate and then to deallocate resources to that job). This number can be viewed two ways. First, it indicates the efficiency of our system. Second, it shows that in the presence of adaptive programs, a resource manager can boost utilization of a network to above 99%.

7 Brief Overview of Selected Resource Management Systems

Initial resource management services were designed at the operating system level, e.g., NEST [1], V [34], Sprite [13], MOSIX [4] and Remote Unix [24]. Coshell [17] provided the same functionality, not as a part of the operating system, but as a UNIX shell.

Resource management systems such as Condor [25], Utopia/LSF [36, 32], DQS/CODINE [14], PBS [21], and LoadLever [22] were originally targeted to handle sequential batch jobs. While successful in this, their support for parallel applications is in some sense batch-like, and of restricted scope. To use these systems the user submits a job and specifies the number and the types of machines required. The resource manager first selects an appropriate set of machines based on their current status, starts up daemons, and performs other necessary tasks to prepare the parallel job for execution. In some cases the resource manager even starts and monitors the job, and informs the user when it completes. An important characteristic is the inherently static nature of the allocation of machines. Machines are allocated only at the initial start of the job's execution; there is no support for dynamic allocation, and reallocation of machines to jobs throughout the execution if higher priority jobs arrive after available machines have been allocated to lower priority jobs.

Some attempts to dynamically load balance parallel jobs relied on techniques similar to load-balancing sequential jobs: process checkpointing and migration, e.g., GLUnix [35], PRM [28], and DynamicPVM [12]. Process migration helps in dispersing a load of a heavily-loaded machine, but it does not increase the number processes in a parallel job.

Systems such as Piranha [7], MPVM [8], Condor/CARMI [33], and DRMS [26] specifically target parallel computations and are capable of dynamically allocating machines to a running job as resources become available, and deallocating machines if they are needed elsewhere. However, they are tightly integrated with either the underlying system or with the parallel programming environment they support. In fact, none of the systems that support adaptive jobs, support more than a single parallel programming model, as we elaborate next.

Piranha is a system built on top of Linda that allows machines to join an ongoing computation as they become idle and retreat when reclaimed. While this was a great step towards providing an adaptive resource management service, Piranha required modifications to Linda, and it supported only those Linda programs that had been modified to use Piranha.

Distributed Resource Management System (DRMS) is a service that runs on the IBM SP2 computer and manages MPI and HPF jobs (that are compiled to use MPI at the lower level). It is parallel application aware, in that it can grow and shrink a job's "node count." However, it is tightly integrated with the SP2 system and the MPI model (it modifies routing tables to redirect MPI messages). Its dynamic services are also limited to programs that are explicitly programmed for DRMS.

Paraphrasing, Condor/CARMI proposed that all resource management functionalities be removed from parallel programming environment code, and migrated into one or more processes which are dedicated to handling resource management requests. This proposal lead to major modifications to PVM's job management services and resulted in a well-defined interface where any system could now provide the management services. Both Condor/CARMI and MPVM utilized these modifications to support adaptive PVM programs—in fact the modifications were critical for the viability of the architecture. This required the resource management system to be integrated with PVM, i.e. link with the PVM library.

In contrast to previous work, we provide a system-independent service for dynamically evolving jobs. In addition to being able to support multiple parallel processing environments, there is no need to relink our software with each newer release of the parallel processing environment such as PVM or MPI.

8 Conclusions

There has been an increase in the popularity of parallel programs developed in systems such as PVM and MPI, which are typically executed on networks shared with other users. Since the users cannot, in practice, constantly monitor the load of every machine on a network, software systems should provide a system-wide management service. However, no existing system supports adaptive programs written with several different programming systems.

In this paper we sketched the design and the implementation of a parallel application aware resource management system. We have validated our techniques by providing dynamic resource allocation for PVM, LAM (an implementation of the MPI standard), and Calypso programs. As our approach relies on plug-in external modules for supporting diverse programming systems, it is extensible. Furthermore, our system executes at a user level, and hence, it does not compromise the security of the enterprise even if it malfunctions (due to a bug for example). The initial performance results are very encouraging. In the presence of dynamic computations all available resources were nearly utilized. It is important to note that our system does not impose an overhead to the execution of any job. It is only during the allocation of resources to jobs that our system intervenes, otherwise the system processes remain dormant and no overhead is imposed by their existence.

The focus our work so far has been a general mechanism we have developed which relies on a set of basic features common to many parallel programming systems. We view this as starting point to a more extensive research program providing end-to-end solutions which both increase the utilization of the available resources and enhance the execution of the users' programs.

Acknowledgements

We thank Toto Paxia for his helpful advice and insights.

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; by the National Science Foundation under grant number CCR-94-11590; and by the Intel Corporation.

References

[1] R. Agrawal and A. Ezzat. Location independent remote execution in NEST. *IEEE Transactions on Software Engineering*, 1987.

[2] R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proc. of SIGMETRICS*, 1995.

[3] M. Baker, G. Fox, and H. Yau. A review of commercial and research cluster management software. Northeast Parallel Architectures Center, Syracuse University, 1996.

[4] A. Barak, S. Guday, and R. Wheller. The MOSIX distributed operating system – load balancing for Unix. Lecture Notes in Computer Science, Springer-Verlag, 1993.

[5] A. Baratloo, P. Dasgupta, and Z. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proc. of IEEE International Symposium on High-Performance Distributed Computing*, 1995.

[6] N. Carriero and D. Gelernter. Linda in context. *Communication of ACM*, 1989.

[7] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook. Adaptive parallelism with Piranha. Technical Report 954, Yale University Department of Computer Science, 1993.

[8] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A migration transparent version of PVM. *Computing Systems*, 1995.

[9] D. Cheng. A survey of parallel programming languages and tools. NAS Technical Report RND-93-005, NASA Ames Research Center, 1993.

[10] E. Chung, Y. Huang, and S. Yajnik. Checkpointing in CosMic: A user-level process migration environment. In *Proc. of Pacific Rim Symp. on Fault-Tolerant Computing*, 1997.

[11] K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith and S. Tuecke. A Resource Management Architecture for Metacomputing. Available at http://www.globus.org/globus/papers.htm.

[12] L. Dikken, F. van Der Linden, J. Vesseur, and P. Sloot. DynamicPVM: Dynamic load balancing on parallel systems. In *Proc. High-Performance Computing and Networking*, 1994.

[13] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software – Practice and Experience*, 1991.

[14] D. Duke, T. Green, and J. Pasko. Research toward a heterogeneous networked computer cluster: The distributed queuing system version 3.0. Supercomputer Computations Research Institute, Florida State University, 1994.

[15] M. Eskicioglu. A comprehensive bibliography of distributed shared memory. *Operating Systems Review*, January 1996.

[16] R. Felderman, E. Schooler, and L. Kleinrock. The Benevolent Bandit laboratory: A testbed for distributed algorithms. In *IEEE Journal on Selected Areas in Communications*, 1989.

[17] G. Fowler. The shell as a server. In *Cincinnati USENIX Conf. Proceedings*, 1993.

[18] A. Geist, A. Beguelin, J.. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel virtual machine*. MIT Press, 1994.

[19] T. Green and J. Snyder. DQS, a distributed queueing system. Supercomputer Computations Research Institute, Florida State University, 1991.

[20] W. Gropp, E. Lust, and A. Skjellum. *Using MPI: Portable parallel programming with the message-passing interface.* MIT Press, 1994.

[21] R. Henderson and D. Tweten. Portable Batch System. NAS Scientific Computing Branch, NASA Ames Research Center, 1995.

[22] IBM. IBM LoadLeveler: General information. 2nd edition, 1993.

[23] J. Jones and C. Brickell. Second evaluation of job queuing/scheduling software: Phase I report. NAS Technical Report NAS-97-013, 1997.

[24] M. Litzkow. UNIX: Turning idle workstations into cycle servers. In *Proc. Summer 1987 USENIX Conference*, 1987.

[25] M. Lizkow, M. Livny, and M. Mutka. Condor: A hunter of idle workstations. In *Proc. ICDCS*, 1988.

[26] J. Moreira, V. Naik, and R. Konuru. A programming environment for dynamic resource allocation and data distribution. In *Proc. 9th Workshop on Languages and Compilers for Parallel Computing*, 1996.

[27] M. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 1991.

[28] C. Neuman and S. Rao. The Prospero resource manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice and Experience*, 1994.

[29] D. Nichols. Using idle workstations in a shared computing environment. In *Proc. of SOSP*, 1987.

[30] Ohio Supercomputer Center. MPI primer/developing with LAM. Ohio State University, 1996.

[31] J. Pruyne and M. Livny. Interfacing Condor and PVM to harness the cycles of workstation clusters. *Journal on Future Generations of Computer Systems*, 1996.

[32] Platform Computing Corporation. LSF User's and administrator's guide. Toronto, Canada, 1993.

[33] J. Pruyne and M. Livny. Parallel processing on dynamic resources with CARMI. In *Job Scheduling Strategies for Parallel Processing–IPPS'95 Workshop Proceedings*, 1995.

[34] M. Theimer and K. Lantz. Finding idle machines in a workstation-based distributed system. *IEEE Transactions on Software Engineering*, 1989.

[35] A. Vahdat, D. Ghormley, and T. Anderson. Efficient, portable, and robust extension of operating system functionality. UC Berkeley Technical Report CS-94-842, 1994.

[36] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computing systems. Computer Systems Research Institute, University of Toronto, 1992.