Mechanisms for Just-in-Time Allocation of Resources to Adaptive Parallel Programs

Arash Baratloo Ayal Itzkovitz Zvi M. Kedem Yuanyuan Zhao
{baratloo,ayali,kedem,yuanyuan}@cs.nyu.edu
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University

Abstract

Adaptive parallel computations—computations that can adapt to changes in resource availability and requirement—can effectively use networked machines because they dynamically expand as machines become available and dynamically acquire machines as needed. While most parallel programming systems provide the means to develop adaptive programs, they do not provide any functional interface to external resource management systems. Thus, no existing resource management system has the capability to manage resources on commodity system software, arbitrating the demands of multiple adaptive computations written using diverse programming environments.

This paper presents a set of novel mechanisms that facilitate dynamic allocation of resources to adaptive parallel computations. The mechanisms are built on low-level features common to many programming systems, and unique in their ability to transparently manage multiple adaptive parallel programs that were not developed to have their resources managed by external systems. We also describe the design and the implementation of the initial prototype of ResourceBroker, a resource management system built to validate these mechanisms.

1. Introduction

Adaptive programs are those that can adapt to external changes in resource availability and internal changes in resource requirements. Most master-slave PVM [10] programs, self-scheduling MPI [11] programs, bag-of-tasks Linda [4] programs, and all Calypso [1] programs are adaptive. For PVM, MPI, and Linda, programs must be written so that they are able to tolerate machine removals; whereas for Calypso, this service is provided by the runtime layer. Adaptive computations can efficiently utilize networked machines for two reasons. First, they can expand to execute on machines that, otherwise, would be left unused. Second, they can acquire machines as their requirements grow, rather than reserving the largest pool of machines required at any single instance during their execution.

Even in computing environments of adaptive jobs with excessive resources, significant load-unbalancing could still occur, resulting in slow turnaround and non-responsiveness. This indicates that intelligent allocation of resources is necessary, especially one that could fullfil the potential imposed by the existence of adaptive programs. We call the entity performing this functionality a *resource manager*. Ideally, the service provided by a resource manager should be available to any compiled executable without requiring explicit programming. For efficient utilization of resources, machines should be allocated to computations only when requested and not pre-allocated and reserved. We refer to this ability as *just-in-time allocation of resources*. Furthermore, a resource manager should be able to manage programs developed using different programming systems. However, as discussed later, none of the existing systems achieves this, which severely limits their applicability.

Parallel programming systems like PVM, LAM (MPI implementation), Calypso, and Linda have a built-in resource manager with limited functionality. It is common for this type of resource manager to (1) schedule parallel tasks among the (already) allocated machines, and to (2) assist in adding explicitly-named machines to a computation. This type of resource manager is refered to as an *intra-job resource manager*, since its primary responsibility is to manage resources within one job. To differentiate from the resource manager built into parallel programming systems, *inter-job resource manager* is used to refer to the system that manages *all the machines among all the executing jobs*. The concept of separating these two types of managers was first discussed in the work in Benevolent Bandit Laboratory [8].

Dynamic allocation and reallocation of machines to jobs requires communication between inter-job manager and multiple intra-job managers. The lack of a common interface introduces a challenge to inter-job resource manager developers: how can their software system communicate with a variety of parallel systems, especially with systems that do not provide an interface? This is the reason existing resource managers for adaptive programs restrict themselves to supporting only a single programming system.

ResourceBroker is the first resource manager to use low-level features common to popular parallel programming systems for its communication, hence, it is able to manage unmodified PVM, MPI, Calypso, and PLinda programs.

The rest of this paper is organized as follows. Sections 2 and 3 present the goals and the architecture of ResourceBroker respectively. Section 4 describes the interactions between the three parties - the user, job, and resource manager. Section 5 describes the mechanism used to dynamically allocate and reallocate resources among running jobs. Experimental results are presented in Section 6 and related work is given in Section 7. We conclude in Section 8.

2. Design Goals

The goal of ResourceBroker is to provide a comprehensive resource management service that is able to dynamically allocate machines among multiple competing computations written in different parallel programming systems. The following is the key objectives met in this system:

1. Parallel programming systems are treated as commodity, so that the executables built for environments with no global resource management service are not expected to be modified to take advantage of the service;

2. Operating systems are treated as commodity, and the only privilege required is user level, thus ResourceBroker doesn't comprise the security of the network;

3. The use of the resource manager is optional, and it is sometimes referred to as a "service" to emphasize its unobtrusive availability;

4. The service can dynamically reallocate resources among adaptive jobs as resource availability changes;

5. Mechanism and policy are separated, making the latter an easily plugin module.

Compared with mechanism, policy is relatively dynamic. Different user requires different policy, and same policy evolves over time, it's desirable to make the integration of new policy into the system as easily as possible. In our initial implementation, the following has been used. User jobs are divided into two classes: adaptive, and others. Similarly, machines are divided into two classes: *private* and *public*. Private machines belong to individuals and the owner has absolute priority over its use, where as public machines are available to all users and typically reside in a public laboratory. The policy implemented by ResourceBroker is to allocate private machines only to adaptive jobs. Hence, adaptive jobs running on a privately owned machine can be deallocated once the owner of the machine returns. In other cases, ResourceBroker tries to evenly partition machines among jobs.

3. Architecture

ResourceBroker consists of two weakly coupled layers: the *resource-management layer* and the *agent layer*. Figure 1 depicts the software layers during the execution of two parallel jobs.

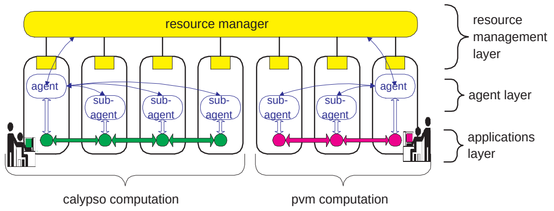


Figure 1. Components of ResourceBroker.

The *resource management layer* consists of a single network-wide resource manager process and a single daemon process on each machine. It is possible to run the resource manager process with non-privilege user access rights, as opposed to administrator access rights. The resource manager process spawns the daemon processes at startup and restarts them if they fail. Daemons are responsible for monitoring resources such as the CPU status, the users who are logged on, the number of running jobs, and the keyboard- and the mouse-status on the machine. This information is periodically reported to the resource manager process. The resource manager process is responsible for deciding which job can use which machine.

The *agent layer* consists of dynamically changing sets of processes. A user who wants to use ResourceBroker's services first starts an *agent* process to submit the job for execution. As the job extends to remote machines, *subagent* process are automatically started to monitor the new processes. The combination of agent and subagent processes forms the agent layer. The agent layer provides the means for the resource manager to monitor and actively intervene in the execution of adaptive jobs. In a broader sense, it acts as a broker between the resource management layer and the running jobs, and is able to coerce programs to achieve the allocation policy determined by the resource management layer.

Many existing resource managers [13, 17, 15, 12] employ a single-level architecture, where the monitoring daemon processes also carry out the responsibilities of agent and subagent processes. The purpose behind a two-level architecture is to allow ResourceBroker to run with user-level privileges only. Although resource-management layer processes run with user privilege, it is able to manage other user's jobs. We describe these mechanisms later in Section 5.

4. User, Job, and ResourceBroker Interactions

In the presence of a resource manager, there are three entities in every job invocation: (1) the resource manager, (2) the user, and (3) the job submitted for execution. Figure 2 depicts them and their interactions. The interaction between a user and the resource manager and the interaction between a running job and the resource manager are of particular interest and are briefly discussed next, more details are available at [2].

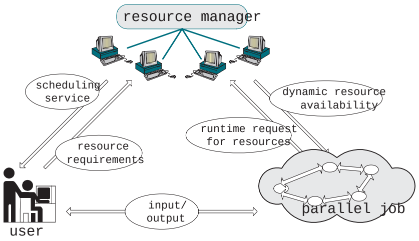


Figure 2. The three entities in job execution.

4.1. Users' Interaction with ResourceBroker

Users communicate with ResourceBroker to query machine availability, to learn the status of queued jobs, to submit a job for execution and specify its resource requirements. ResourceBroker adopted the Resource Specification Language of Globus [9], and extended it to support adaptive programs. Specifically, `adaptive`, `start_script`, and `module` parameters were added to describe adaptive jobs. As an example, the specification `+(count>=4)(arch="i86Linux")(module="pvm")` is a request to execute a PVM program on at least four Intelx86 Linux machines. The option (`module="pvm"`) specifies that ResourceBroker should use external modules to carry out some of its responsibilities. We next discuss external modules.

4.2. Interaction of Jobs and ResourceBroker

Interaction between resource manager and jobs is complicated by the fact that they are typically developed independently. To address this, ResourceBroker relies on a set of common features to build an interface between intra- and inter-job resource managers.

Most parallel programming systems' intra-job resource managers are capable of relinquishing machines, but are unable to locate additional underutilized machines. As the requirements of adaptive jobs change over time, the resource manager must be alerted to these changes. Thus, at a minimum, adaptive jobs must inform the resource manager of their desire to add additional machines.

On Unix, the `rsh` command and `rexec()` system call are the common underlying mechanisms to start program executions on remote machines, although the actual method used by the programmers is likely to be higher-level, e.g., Calypso and PVM programs grow by calling `calypso_spawnWorker()` and `pvm_addhosts()` respectively, but ultimately they results in a `rsh` command. This is also true if the computation pool is grown using the Calypso graphical user interface or PVM console.

The `rsh` command requires an explicit machine name argument, as in "`rsh` *`host`* `<command>`." ResourceBroker intercepts `rsh` commands issued by jobs running under its control. Intercepted `rsh` commands with *symbolic* host names, e.g. *anyHost*, are interpreted as intra-job resource managers' requests for assistance; `rsh` with real host names are allowed to proceed. Symbolic host names are also used as a request specification, e.g., *anyLinux* indicates any machine running Linux. As we show in the next section, it is easy to make existing programs issue `rsh` commands with symbolic host names.

```
#! /bin/bash
echo add $1 > $HOME/.pvmrc
echo quit >> $HOME/.pvmrc
pvm > /dev/null
rm $HOME/.pvmrc
```

Figure 3. Module to grow PVM.

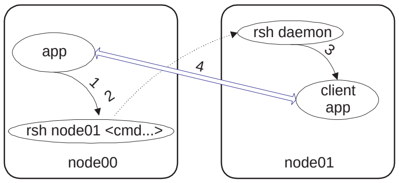


Figure 4. Representative scenario of a parallel job acquiring a machine.

Once ResourceBroker decides to allocate a machine to a running job, the action is carried out by the agent process responsible for that job. Parallel programming systems that allow anonymous machines to join a computation are handle slightly differently from the systems that do not. The default behavior of the agent process is to replace the symbolic host-name with a real name and then to allow the `rsh` command to proceed—in a sense, the agent process redirects the `rsh` command to a machine unknown to the job. The default behavior is appropriate for Calypso, PLinda and sequential jobs. For systems like PVM and LAM which do not allow an unexpected machine to join a computation, the agent process relies on external *modules* to communicate the real host name to the job and to coerce the job to accept it. Modules are executable programs, or shell scripts, that are external to ResourceBroker. This architecture allows future support for as yet undefined programming systems without having to recompile the resource manager.

When a user submits a job along with a module option, as in `module="xxx"`, ResourceBroker assumes the existence of three external programs named `xxx_grow`, `xxx_shrink`, and `xxx_halt`, to assist in growing, shrinking, and halting the job respectively. According to this scheme, PVM modules would be `pvm_grow`, `pvm_shrink`, and `pvm_halt`. Figure 3 depicts the source code of `pvm_grow` : it writes a sequence of commands to `$/.pvmrc` and then invokes a PVM console to execute them. Notice how this is a simple script that simulate users' actions, this is also true for LAM modules.

5. Mechanisms

To provide the context used in illustrating the mechanisms behind ResourceBroker, a representative scenario of how parallel jobs acquire resources *in the absence* of resource managers is briefly described.

A user running a program named `app` on machine `node00` wants the computation to grow to `node01` when needed, s/he prepares a hostfile, named `.hosts`, containing `node01` and starts `app` on `node00`. At some point `app` decides to spawn a process on another machine. Figure 4 depicts the steps involved in this process. The `app` process consults `.hosts` for a machine name, reads `node01`, and issues the command "`rsh` *`node01`* `<command>`"(step 1). In the job's source code this could have been a higher-level function, but ultimately it is translated to the standard `rsh` command. The `rsh` command contacts the rsh daemon (`rshd`) on `node01` (step 2), which spawns a process on `node01` on behalf of `app` (step 3). The new process establishes a communication with `app` (step 4). This completes the addition of `node01` to the computation.

When using ResourceBroker, selection of the second machine can be delayed until the job is ready to use the machine. It is then the responsibility of the agent process to coerce the job to use a machine that is selected at runtime. Schematically, the agent process needs to (1) realize that the job "wants" to spawn a remote process; (2) notify the resource manager that a new machine has been requested; (3) obtain the name of the target machine that is "most appropriate"; (4) spawn, or cause to spawn, the second process on the target machine; (5) enable the two processes to establish a communication stream; (6) fade in to the background, so as not to impose an overhead.

5.1. Required Conditions

ResourceBroker is capable of dynamically selecting resources for unmodified executables as long as the following requirements are met:

1. The program doesn't use hard-coded machine names.

2. The program does not have the absolute path of the `rsh` command hard-coded. This is the case for most programming systems, e.g. PVM and LAM (MPI).

3. For programming systems that do not allow anonymous machines joining a computation, there is a command line interface for users to grow the pool of machines used in a computation, moreover, it could tolerate failed attempts to add additional machines. This is the case for PVM and LAM.

5.2. Default Behavior

The default behavior for ResourceBroker is as follows. First, the user prepares the `.hosts` file containing *anyLinux*, a symbolic machine name. Second, on *host00* the user types: `$agent app <arguments>`. This starts an agent process, which immediately spawns a child process to execute `app`. Figure 5 depicts the steps involved for `app` to spawn a process on another machine when executing under ResourceBroker's control. When `app` decides to grow, it consults `.hosts`, finds *anyLinux*, and issues the command "`rsh` *`anyLinux`* `<arguments>`". See step 1 of Figure 5, where our implementation of `rsh` is depicted as `rsh`′. Realizing that *anyLinux* is a symbolic machine name, ResourceBroker intervenes. The `rsh`′ process contacts the agent (step 2), which contacts the resource manager process with a request for a machine (step 3). Once a target machine name is given to the agent, say `node01`, it notifies `rsh`′ of this machine name (step 4). Then `rsh`′ uses the standard `rsh` to spawn a subagent process on `node01` (steps 5-7). The subagent contacts the agent process for a program to execute (step 8), and spawns the appropriate process (step 9) on `node01`. The newly created process contacts the original `app` (step 10) and the job continues executing normally. Notice how the agent layer redirects the `rsh` to use a target machine selected at runtime.

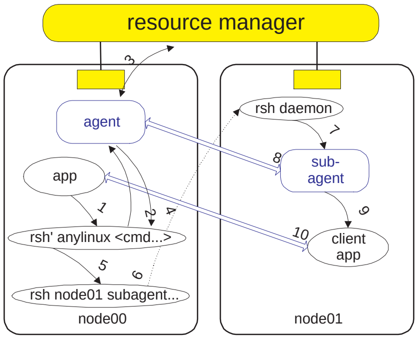


Figure 5. Adding a dynamically allocated machine (default behavior).

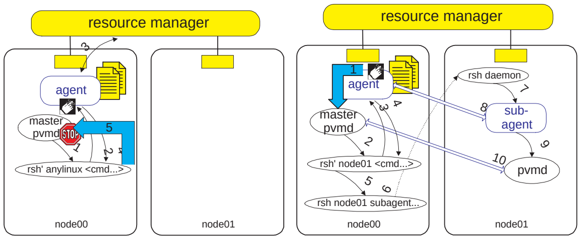
From this point, until resources need to be reallocated, there is no interaction between `app` and ResourceBroker. The various agent-layer processes remain dormant, and no overhead is imposed by their existence. Future interactions can begin in two ways: first, by the job attempting to add another machine; second, by the resource manager deciding to reallocate machines. To take away `node01` from `app`, the subagent sends a standard Unix signal to the child process, and if the child does not terminate within a specified amount of time, the subagent terminates the child process.

The default behavior described above is used for Calypso and PLinda programs, for parallelizable tasks such as `make`, and for executing sequential jobs remotely.

5.3. External Modules

In reviewing the default behavior, note that `app` running on `node00` attempted to spawn a process on a machine it believed to be *anyLinux*, whereas the process was spawned on `node01`. Generally, redirecting the `rsh` goes unnoticed. However, PVM and LAM programs will refuse to accept processes from machines other than those they attempted to spawn.

To handle this type of situation, ResourceBroker relies on external modules to carry out its responsibilities. A similar mechanism is used for both PVM and LAM programs:



(a) Phase I (b) Phase II

Figure 6. Adding a dynamically allocated machine (using external modules).

the “plug-in” external module approach makes the design extensible and thus able to accommodate various programming systems concurrently. A PVM-specific scenario, as illustrated below, is used as a representative usage of external modules.

Knowing that `app` is a PVM program, the user types: `$agent pvm --(module="pvm")`. This submits the PVM console program, `pvm`, to an agent process and instructs ResourceBroker to use PVM-specific modules. The `agent` process immediately spawns `pvm`, which in turn starts the master PVM daemon. The user can create a PVM virtual machine and start PVM programs as usual.

PVM’s virtual machine can grow in two ways: at the PVM console, the user can type `pvm> add` *`anyLinux`*, or the program can call the `pvm_addhosts()` library function with *anyLinux* as the host name argument. In both cases, this results in the master PVM daemon issuing “`rsh` *`anyLinux`* `<arguments>`”. ResourceBroker intervenes when it detects an `rsh` with a symbolic host name. The allocation of resources using external modules happens in two phases. Figure 6 depicts the steps involved in this process.

Once “`rsh` *`anyLinux`* `<arguments>`” is issued by PVM daemon(step 1 of Figure 6(a)), `rsh`′ contacts the agent (step 2), which asks the resource manager process for a machine (steps 2 and 3). The resource manager process knows that the job is a PVM task and propagates this information to `rsh`′ (steps 3 and 4). The `rsh`′ then terminates with an error status code (step 5). The first phase is completed with the result that (1) the master PVM daemon sees a failed attempt to grow the virtual machine, but more importantly, (2) ResourceBroker recognizes the request for an additional resource.

Figure 6(b) depicts the steps in the second phase of this process. Following ResourceBroker’s recognition of PVM’s request for an additional machine, phase two begins by the agent executing the external module `pvm_grow` with argument *host01* (step 1). This script consists of five lines and is shown in Figure 3. The script opens another PVM console, asks the master PVM daemon to add machine *host01* to the virtual machine, and closes the console. This results in the PVM daemon issuing another `rsh` command with `node01` as the machine name (step 2). The second phase proceeds like the default case and results in starting a PVM daemon process on *host01* (steps 3-10).

Operation	Time (s)
`rsh` *n01* `null`	0.4
`rsh`′ *n01* `null`	0.6
`rsh`′ anyLinux `null`	0.6
`rsh` *n01* `loop`	36.9
`rsh`′ *n01* `loop`	37.0
`rsh`′ anyLinux `loop`	37.1

Table 1. Performance of `rsh`′

Three important features are worth emphasizing. First, the PVM daemon was coerced into accepting *host01*. Second, PVM’s second attempt to add a host proceeds as usual; allocating “a suitable machine at the time of request” became an “invisible” service. Finally, as machines become available, ResourceBroker is able to asynchronously initiate the second phase to increase the size of PVM’s virtual machine.

6. Experiments

This section presents experimental results that validate the proposed mechanisms. Experiments were conducted using up to 16 200MHz PentiumPro machines running Linux RedHat 4.0 connected by Fast Ethernet. Reported times are median measured elapsed times taking into account all overheads. In this section, we use `rsh` to denote the standard Unix remote shell program, and `rsh`′ to denote ResourceBroker’s version.

6.1. Micro Benchmarks

Table 1 shows the performance of `rsh` ′ compared with `rsh`. Two sequential applications - `null` (a C program with empty `main()` function), `loop` (a C program with a tight loop running in 34.4 seconds), and two idle machines - *n00*, *n01* were used in the experiment. The commands in Table 1 were issued on *n00* and directed to execute on *n01*. Thus, the command “`rsh`′ *`n01`* `loop`” results in executing `loop` on *n01*. The *anyLinux* keyword is interpreted as “any available Linux machine.” Thus, the command “`rsh`′ *`anyLinux`* `loop`” allows the resource manager to choose a machine to execute `loop`. In this particular experiment, the available set of machines was limited to *n01*, so in fact *n01* was always chosen. As the results indicate, the overhead associated with `rsh`′ is approximately 0.2 seconds, which is hardly noticeable by users. The small overhead also indicates that replacing the system-wide `rsh` with `rsh`′ is feasible, even if some users do not use the features provided by ResourceBroker.

The second set of experiments measured the required times to reallocate machines. The results are shown in Table 2. Three machines were used in this experiment: *n00*,

Operation	Time (s)
rsh *n01* null	0.4
rsh′ anyLinux null	1.5
rsh *n01* loop	38.2
rsh′ anyLinux loop	37.9

Table 2. Performance of reallocation.

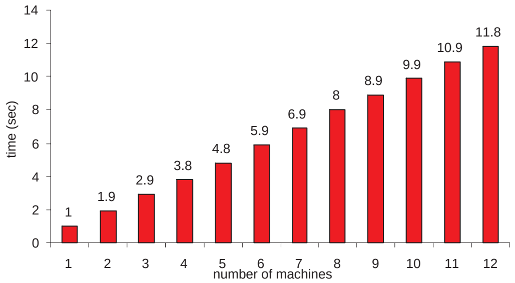


Figure 7. Performance of resource reallocation using PVM and rsh′.

n01, and *n02*. An adaptive Calypso program ran on *n01* and *n02*. Similar to the previous experiment, the commands of Table 2 were issued on *n00*, and in every case resulted in the allocation of *n01*. In the case of rsh′, ResourceBroker terminated the Calypso process running on *n01* before satisfying the request. 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 *n01* is cleared of external processes before executing the job.

6.2. Parallel Computations

We now present the performance of ResourceBroker managing parallel jobs, in particular, the effects of external modules within the agent layer are measured.

Table 3 shows the performance of rsh′ when used by parallel programs. The operation "pvm w/ rsh′ *host*" means the PVM program explicitly named the machines it wanted to use, and "pvm w/ rsh′ *anyLinux*" means the PVM program left the choice of the machine to ResourceBroker. The results illustrate that when the machines are explicitly named, ResourceBroker introduces less than 0.1 milliseconds of overhead per machine. Allowing ResourceBroker to choose a machine (i.e, rsh′ *anyLinux*) 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. This result reinforces that replacing the system-wide rsh with rsh′ is feasible, and that this will go unnoticed by users who do not use the additional features provided by rsh′.

Next experiment measures the time to reallocate resources for parallel jobs. An adaptive Calypso job ran on every machine. A PVM virtual machine was created several times, and each time a different size (denoted by n) virtual machine was built. To satisfy the PVM requests, machines had to be taken away from the Calypso job first. Figure 7 reports the elapsed 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 the second set of experiments), and that this number scales linearly to at least 12 machines.

The final experiment measures the utilization factor of a dynamic environment. The setting was as follows. An adaptive Calypso job ran initially on eight machines. Every 100 seconds, a script started a sequential program that ran for *t* minutes, where *t* was chosen uniformly from the interval [1,10]. After five hours, the total detected idleness (the total amount of time that the machines were idle) was less than 1%. This number can be viewed in two ways. First, it indicates the efficiency of the reallocation mechanisms. Second, it shows that in the presence of adaptive programs, a resource manager can boost utilization of a network to above 99%.

7. Related Work

Early systems for resource management were designed to disperse jobs among available machines on a network, in which the system selects the machine to execute the user-submitted program, and redirects the terminal IO. However, the focus of them is to support sequential computations and they do not make any special provision for parallel programs.

Other systems like Condor [13] and Utopia [17], were developed for managing heterogenous resources of networks of workstations. They are typically Queue Management Systems and were originally intended to be used with batch sequential jobs. With the increased popularity of parallel programming systems such as PVM and MPI, they extended to support parallel interactive jobs. Globus [9] and Legion [7] are large-scale resource managers designed to unite machines from multiple administrative domains. In these systems, resource requirements are only submitted together with the job, and resources are allocated only once and before the jobs starts executing.

More recent systems specifically target adaptive parallel computations and are capable of dynamic resource allocation. Cilk-NOW [3] and Piranha [5] are systems combined of a parallel programming language as well as a resource manager. Piranha extends Linda to allow adaptive jobs, however, it required modifications to the Linda system, and only supported Linda programs that had been modified to use it; both Condor/CARMI [16] and MPVM [6] required major modification of PVM system to support adaptive PVM programs and they only work for PVM programs. Distributed Resource Management System (DRMS [14]) is tightly integrated with the SP2 system and MPI implementation (it modifies routing tables to redirect MPI messages), its dynamic services are also limited to programs that are explicitly programmed for DRMS.

Unique from the previously mentioned resource managers, our system is the first to support adaptive programs written for different parallel programming systems.

Operation	1 machine(s)	2 machines(s)	4 machines(s)	8 machines(s)
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`′ anyLinux	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`′ anyLinux	4.2	8.8	16.8	33.5

Table 3. Performance of ResourceBroker to dynamically add a resource to PVM and LAM programs.

8. Conclusions

The efficiency of adaptive computations in utilizing networked machines relies on the resource manager's ability to communicate resource availability to computations, and computations' ability to communicate their resource requirements to the resource manager. The lack of standard interface for this communication limits existing resource managers unable to support different parallel programming systems in a uniform way.

In this paper we presented a set of mechanisms to effectively build this communication interface, even for programs not developed to work with resource managers. Built on mechanisms of intercepting and interpreting low-level actions common to many parallel programming systems and external plug-in modules, ResourceBroker is the first to support multiple systems such as PVM, MPI (LAM), Calypso and PLinda without any requirement to modify them, it also facilitates future support for as yet undefined programming systems. Furthermore, ResourceBroker executes at user level, and hence does not compromise the security of the networked machines even if it malfunctions.

Acknowledgements

This research was sponsored by DARPA and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320, and by NSF under grant number CCR-94-11590. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

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

[2] A. Baratloo, A. Itzkovitz, Z. Kedem, and Y. Zhao. Just-in-time transparent resource management in distributed systems. Technical Report TR1998-762, Department of Computer Science, New York University, 1998.

[3] R. Blumofe and P. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the USENIX Annual Technical Conference*, 1997.

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

[5] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook. Adaptive parallelism with piranha. Technical report, Yale University Department of Computer Science, 1993.

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

[7] S. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. Resource management in legion. *International Journal on Future Generation Computer Systems(to appear)*, 1999.

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

[9] I. Foster and C. Kesselman. The globus project: A status report. In *Proceedings IPPS/SPDP '98 Heterogeneous Computing Workshop*, 1998.

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

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

[12] R. Henderson and D. Tweten. Portable batch system. *NAS Scientific Computing Branch, NASA Ames Research Center*, 1995.

[13] M. Lizkow, M. Livny, and M. Mutka. Condor: A hunter of idle workstations. In *Proceedings International Conference on Distributed Computing Systems*, 1988.

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

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

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

[17] 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.