A Resource Management System for Adaptive Parallel Applications in Cluster Environments

Sheikh K. Ghafoor[1], Tomasz A. Haupt[1], Ioana Banicescu[2,3], Ricolindo L. Carino[2] , and Nisreen Ammari[1]

[1]Center for Advanced Vehicular Systems, [2]Center for Computational Sciences
[3]Department of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762-5404, USA
{ghafoor, haupt, ioana, rlc, ammari}@erc.msstate.edu

Abstract

Adaptive parallel applications that can change resources during execution, promise better system utilization and increased application performance. Furthermore, they open the opportunity for developing a new class of parallel applications driven by unpredictable data and events, capable of amassing huge resources on demand. This paper discusses some of the requirements for a resource management system to support such applications including communication and negotiation of resources. To schedule adaptive applications, interaction between the applications and the resource management system is necessary. While managing adaptive applications is a multidimensional complex research problem, this paper focuses only on support that a RMS requires to accommodate adaptive applications. An early prototype implementation shows that scheduling of adaptive applications is possible in a cluster environment and that the overhead of management of applications is low compared to the long running time of typical parallel applications. The prototype implementation supports a variety of adaptive parallel applications in addition to rigid parallel applications.

1. Introduction

A classification of parallel applications, proposed by Feitelson and Rudolph [1], divides applications into four classes. **Rigid** applications use a fixed number of processors specified by the user at the time of application submission. **Moldable** applications are assigned processors by the system scheduler within the range specified by the user. However, once an application is started, the number of processors it uses cannot be changed. In contrast, **evolving** applications can change the number of processors during execution. The change in resource requirements in evolving applications is triggered by the applications themselves, due to the nature of the employed algorithms and/or unpredictable data. In the case of **malleable** applications, the number of processors assigned to an application may change during the application execution as a result of a change in resource availability. The change is triggered by events external to the applications, and controlled by the resource management system. Throughout this paper the malleable and evolving applications of Feitelson's classification are defined as adaptive applications.

A timely response to an emergency is an example of a situation where a routine, low resource-consuming application suddenly requests additional resources after detecting the signature of a potential disaster. After solving the urgent problem, such an evolving application would release the excess resources and return to its routine processing. In the future, the evolving applications, driven by unpredictable data and events, will be able to perform so-far unfeasibly large, time-sensitive computational tasks by taking advantage of the vast computational resources available in the grid environment, or those created on demand through spontaneously formed peer-to-peer networks. A malleable application, on the other hand, can utilize otherwise idle resources and thus enable higher system utilization and lower turnaround time. Conversely, a malleable application can shrink at a scheduler's request to relinquish resources. The change in resource availability may result from hardware failures or the necessity of allocating additional resources to higher priority applications.

Unfortunately, current Resource Management Systems (RMS) require an application to specify its resource requirements (such as the number of processors) at the time of submission. Consequently, the resources allocated to an application cannot be changed once the application starts execution, as it is in the case of rigid and moldable applications. This limitation makes it impossible to efficiently execute adaptive applications. Absence of infrastructure support is one of the major obstacles for application developers to write adaptive applications. On the other hand, since there is an absence of a large number of adaptive applications, researchers have not been sufficiently motivated to research and develop infrastructure support. The development of the RMS is an attempt to break this cycle.

Management of adaptive applications in a distributed environment is a complex and multi-faceted problem. If workloads include adaptive applications, interactions between applications and the RMS are necessary to request and negotiate change in resources. This interaction has two aspects: a protocol for communication and resource negotiation, and management of negotiations by the RMS to handle multiple requests from many applications. Unlike rigid applications a scheduling algorithm for adaptive applications has to consider requests from running applications along with the queued applications. In addition, support for resource reallocation (allocate additional resources to or claim released resources from) for running applications is required. This paper addresses only some aspects of the problem, in particular it focuses on the architecture of a RMS for a cluster environment, which is capable of handing adaptive parallel applications along with rigid and moldable applications.

The rest of the paper is organized as follows: a discussion about adaptive applications is presented in section 2. Requirements of RMS and Related works are presented in sections 3 and 4. The proposed architecture is described in section 5. The prototype implementation and results are discussed in section 6 followed by summary and future works in section 7.

2. Adaptive Applications

Large scientific and engineering applications are typically iterative and are composed of inherently serial sections, some data-parallel sections, and possibly, task-parallel sections where some of the tasks are also data-parallel. Intuitively, such an application would require one processor in the serial sections and multiple processors in the parallel sections. The application may also exhibit other interesting behavior. A data-parallel part may require a minimum number of processors to store the data, but can efficiently utilize only a certain maximum number of processors for the computations. Some of the computations may generate additional work during the earlier iterations, while work may be reduced in the later iterations. The computations may be non-uniform, potentially requiring load balancing among the allocated processors. In the case of the load imbalance, the usual approach is to migrate work from a heavily loaded processor to a processor with lesser load. Several dynamic loop-scheduling techniques have been developed for balancing workloads among a fixed number of processors executing a data parallel scientific application. These techniques are derived from theoretical advances in research on scheduling parallel loop iterations with variable running times on a fixed set of processors [2][3][4][5][6], and have resulted in the development and implementation of more advanced techniques based on probabilistic analysis [5][7][8][9][10][11][12]. Many of these techniques have been successfully utilized in the parallelization of Monte-Carlo simulations [5], N-body simulations [7][9] radar applications [8], computational fluid dynamics [11][12], and simulation of wave packet dynamics using the quantum trajectory method [13]. On distributed systems, these techniques are implemented using a master-worker strategy. The master schedules the initial workloads, monitors worker loads, and initiates data migration from heavily loaded workers to lightly loaded or idle workers when a load imbalance is detected.

Many existing applications, especially those that exhibit unpredictable load imbalance, may be turned into adaptive applications. Data-driven applications where the relationship between the amount of allocated resources and the time to completion is not known ahead of time are very good candidates for conversion into evolving ones. This category of applications includes those that process unpredictable amounts of input data, applications that are modeled by tree computations, such as branch-and-bound, search, and divide-

and-conquer algorithms or data-parallel applications that exhibit load imbalance. In an evolving framework, when a load imbalance is detected, instead of balancing the load among the allocated processors, an application would request additional processors from the RMS, and distribute the excess load from the current processors to any newly allocated processors. Even applications that do not change resource requirements during execution and do not exhibit load imbalance are worth making adaptive. Running as malleable applications they would avoid being preempted when a higher priority job arrives, and they would complete earlier by utilizing otherwise idle resources.

A general framework for adaptive applications may be derived from the structure of applications that utilize loop scheduling for dynamic load balancing. In loop scheduling, a master process is responsible for making load balancing decisions. In both malleable and evolving applications, the master process can act as a coordinator process. The coordinator process carries out the communication and negotiation with the RMS on behalf of the application. Whenever an evolving application needs additional resources, the coordinator process communicates and negotiates with the RMS, and once the resources are allocated, a load-balancing phase is initiated. In the case of malleable applications the RMS communicates with the coordinator process of the applications, requesting it to release resources or allocate it additional resources. This communication and negotiation can happen while the application blocks, or it can take place in a separate thread while the computation is on-going. In general, parallel applications have natural "breakpoints" in the execution logic where resources can be added or released. As a result, resources are not consumed or released immediately after completion of negotiation, but at the next breakpoint of the application.

3. Requirements of RMS for Adaptive Applications

In order to manage adaptive applications, interactions between applications and the RMS is required. The communication scenarios that may occur between adaptive applications and a RMS may widely vary; two examples are briefly described below:

1. An evolving application, which requires that the number of processors be a power of 2 and is executing on 8 processors. In mid execution the application needs additional resources and asks the scheduler for 24 additional processors. The scheduler may have only 15 processors available; and instead of rejecting the request it may offer the application 15 processors. Since the application can use only 8 additional processors out of 15 offered, it may ask the scheduler to allocate 8 additional processors.

2. In an environment where applications pay for resources, when some idle resources are available the scheduler may offer the resources to a malleable application for a price. The application may be willing to accept the additional resources at a lower price, and makes a counter offer. Depending on the policy the scheduler may accept or reject the offer or even make another counter offer.

From the above scenarios it is evident that a simple accept/reject type of communication is not enough. A complex multi-round negotiation between applications and the RMS is required to support a wide variety of parallel adaptive applications. To manage adaptive applications as well as rigid and moldable applications, a traditional RMS has to address the following additional requirements:

Negotiation mechanism: A negotiation module, which can manage communication and resource negotiation with running adaptive applications, and can communicate with adaptive applications using a standard protocol.

Negotiation protocol: A standard negotiation protocol, which can handle a wide variety of negotiation scenarios.

Modified node controller: Unlike traditional RMS, a modified node controller capable of handling resource reallocation (granting additional, or claiming previously allocated resources to and from running applications).

Scheduling algorithm: Scheduling algorithms for adaptive applications are more complex than rigid applications [1]. The scheduler has to respond to the demands of both running and queued applications in a timely manner. For example, a scheduling algorithm for rigid applications needs only to consider how to distribute the resources among applications, while a scheduler for adaptive applications has to also consider how many resources to assign or reassign to these applications. These added degrees of freedom increase the computational complexity of the scheduler, which in turn affect the time taken to produce a schedule. Promising research has been conducted that considers moldable applications [14]. Scheduling adaptive applications is a more complex problem that requires a deeper discussion, and is beyond the scope of this paper.

Management of additional scheduling events: A RMS for adaptive applications has to handle a new scheduling event (request for adaptation from evolving applications) in additional to traditional scheduling events (arrival, termination of applications and expiry of scheduling timer). The requests from evolving applications will add additional cost to the scheduler and the frequency of the request will depend on the number and nature of running evolving applications. The RMS has to manage these requests efficiently.

Structure of adaptive applications: A RMS for adaptive applications puts some constraints on applications. Both evolving and malleable applications are required to communicate with the RMS using the protocol that the RMS is following. Therefore, they need to know the contact information (such as host and port no.) of the negotiation module. In addition, a malleable application has to inform the RMS of its contact information, to enable the RMS to communicate with it, when necessary.

4. Related Work

The goal of a Resource Management System (RMS) is to provide support for efficient utilization of computational resources and resolving conflicts between interests of end users. Typically, this goal is achieved by organizing the workload composed of sequential and/or parallel applications, into queues and creating a schedule. The schedule determines the order in which the applications are executed on computing nodes. The schedule is generated according to policies specified by the resource stakeholders. The RMS functionality also includes actual resource allocation (submitting the applications) or de-allocation (terminating the applications), and reporting the applications' status (pending, running or completed) to the system administrator and to the end users. The actual resource allocation is done by node controllers associated with each computing node of the system managed by the RMS.

Much work has been conducted on RMS both in academia and industry [15][16][17][18][19][20][21] addressing many vital aspects of efficient resource management. They represent the current state-of-the-art for problems that can be solved using static and open-loop scheduling approaches. They deal with both simple workloads (independent sequential and parallel applications) and dependent workloads (workflows). Some implementations support moldable applications [21]. Other implementations address resource co-allocation through advance resource reservation [20][21].

The need for RMS support for adaptive applications on distributed memory systems has been recognized and addressed by some researchers. In the simplest form, the adaptation can be achieved by checkpointing a running application and restarting it with a different resource allocation [22]. A more advanced support is offered by the Dynamic Resource Management System (DRMS) system [23], which can handle malleable applications. However, DRMS does not support negotiations between the applications and the scheduler. Instead, the possible application resource configurations are set at the submission time (as in the case of the moldable applications). Then, depending on the resources availability, the scheduler may change the initial resource allocation at runtime by selecting one of the predefined configurations. The adaptation, including the data redistribution is facilitated by dedicated runtime support. The AMPI-based

RMS by Kale *et al*. [24] as well as work based on Adaptive Multiblock Parti (AMP) library [25] ignores the actual resource reallocation mechanisms. Both support adaptive applications within a fixed number of processors shared between different applications and the adaptation is therefore reduced to balancing the processors' load. The load balancing is accomplished by the management, including migration, of computational threads (AMPI) or data redistribution (AMP). Finally, the work by Jha *et al*. [26] tackles the adaptive resource allocation problem for a pool of dependent applications (subtasks) cooperating in real time towards a common goal. This solution is application domain-specific, as the scheduler is responsible for converting the rate of data processing of the subtasks into an estimate of the completion time, leaving no room for the resource negotiations.

None of the above mentioned efforts explicitly target the communication and negotiations between the applications and the RMS or dynamic resource reallocation, which are imperative to support adaptive applications.

5. Architecture of Proposed Resource Management System

Figure 1 shows a proposed architecture of the RMS for adaptive applications. Similar to other RMS designs, the heart of the proposed system is a server, which is responsible for gathering information about the available resources, accepting applications from the users, organizing those applications in queues, and initiating a schedule cycle. Once a schedule is contrived, the server contacts the individual node controllers, which place applications into execution. There is one node controller per computational node. Each controller acts as an agent of the server and starts and controls applications on the nodes. In particular, the node controller allocates additional resources to or claims released resources from adaptive applications running on the system.

The schedules are generated by a scheduling module, which, unlike systems supporting only rigid or moldable applications, comprises two components: a scheduler and a negotiation manager. The scheduler implements a scheduling algorithm to satisfy the system policies taking into account the current status of the system, that is, information about the state of the resources and queued applications. An adaptive scheduler incorporates information about the running applications as well, including engaging in resource negotiations. The generation of a schedule is triggered by scheduling events. The negotiation manager carries out negotiations with running adaptive applications on behalf of the scheduler. The explicit support for resource negotiations between the scheduler and the running applications makes the proposed architecture capable of handling adaptive applications. For negotiation of resources between adaptive applications and the RMS, a negotiation protocol has been developed and implemented.

5.1 Resource Negotiation Protocol

Figure 2 shows the finite state machine representation of the negotiation protocol. The initiator (either applications or the RMS) specifies resource requirements and associated terms and conditions. The other party examines the resource request and responds. The response can be accept, reject, or counter offer with modified requirements. The negotiation results in either accepted or rejected status. In the case of accept, the agreed resources are allocated. All the information regarding a negotiation (resources requested, terms and condition etc.) is encapsulated in an object, which we call the Negotiation Template (NT). The negotiation takes places by exchanging this template. It is composed of three sections. The first section contains general information related to the two parties. The second section contains the status of the negotiation, and the third section contains a list of resource objects that are being negotiated. Each resource object has information about the resource being negotiated, the quantity of the resource being requested, and the status of each resource request. A NT can have more than one resource request inside it. Each resource request has its own terms for the negotiation and its own status. The overall status of the negotiation depends on the combined status of all the requested resources. A detailed description of the resource negotiation protocol is described in [27][28]. A set of APIs that can be used by adaptive applications and the RMS for resource negotiation has been developed. The negotiation manager uses these APIs to communicate and negotiate with adaptive applications.

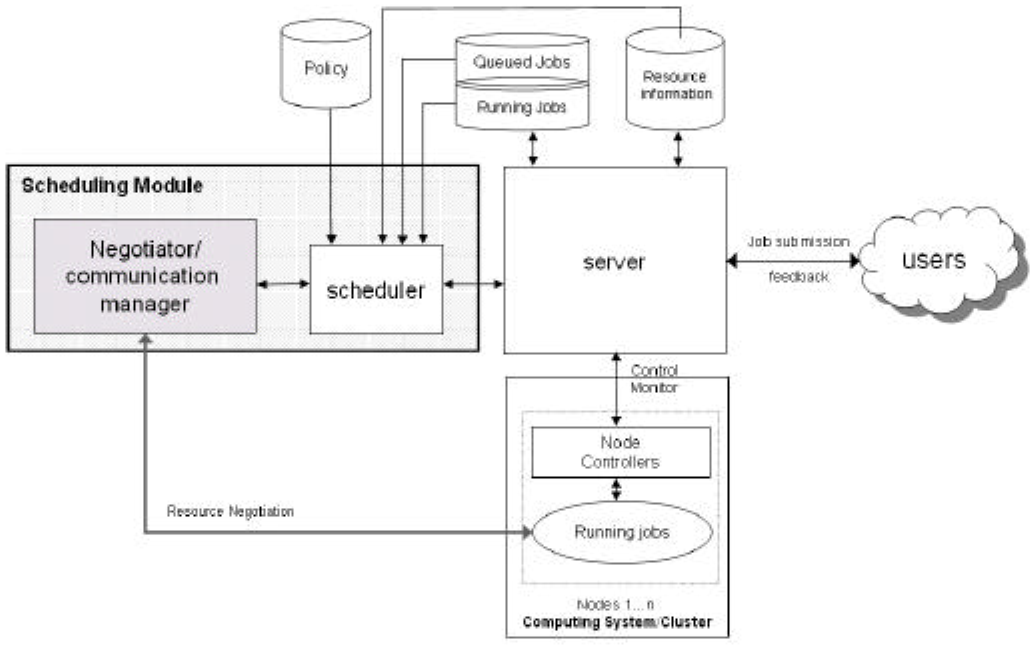

Fig. 1. Architecture of a RMS for Adaptive Applications. The feature that distinguishes this design from the current practice is the explicit support for resource negotiations between the scheduler and allocated applications.

6. Implementation and Results

To validate the proposed architecture a prototype RMS has been implemented. A set of parallel adaptive applications (both malleable and evolving) to test the RMS has also been developed. It is possible to modify an existing open source RMS such as PBS[18] to enable it to negotiate resources and reallocate resources to an adaptive application. However, that requires a detailed and clear understanding of huge volume of codes written by others, which is time consuming. Therefore, an alternative approach of developing a prototype RMS has been adopted, whose main purpose is to serve as a test bed for the proposed architecture. The prototype implementation was tested on an eight-processor Pentium 4 cluster running Red Hat Linux that has Network File System (NFS) support. The RMS was implemented in C and test applications were developed in C using LAM MPI. The resource negotiation APIs were implemented as a library, which was used to implement the negotiation manager and adaptive applications.

The test RMS consists of a scheduling module comprised of a First Come First Serve (FCFS) scheduler and a negotiation manager, a server, and a node controller. The scheduling module and the server are implemented as a single process, and the node controller is implemented as a separate process. The server process runs on the head node of the cluster, and each node of the cluster runs a node controller. The server acts as coordinator, which manages the application queue, talks to clients, communicates with the node controller, maintains resource information, initiates scheduling cycle, etc. The negotiation manager listens on a predefined port for connections from evolving applications. Similarly malleable applications have a separate negotiation thread, which listens on a port; at the beginning of the execution a malleable application sends its host and port number to the RMS. The RMS maintains this information in the application queue and uses it to communicate with the applications.

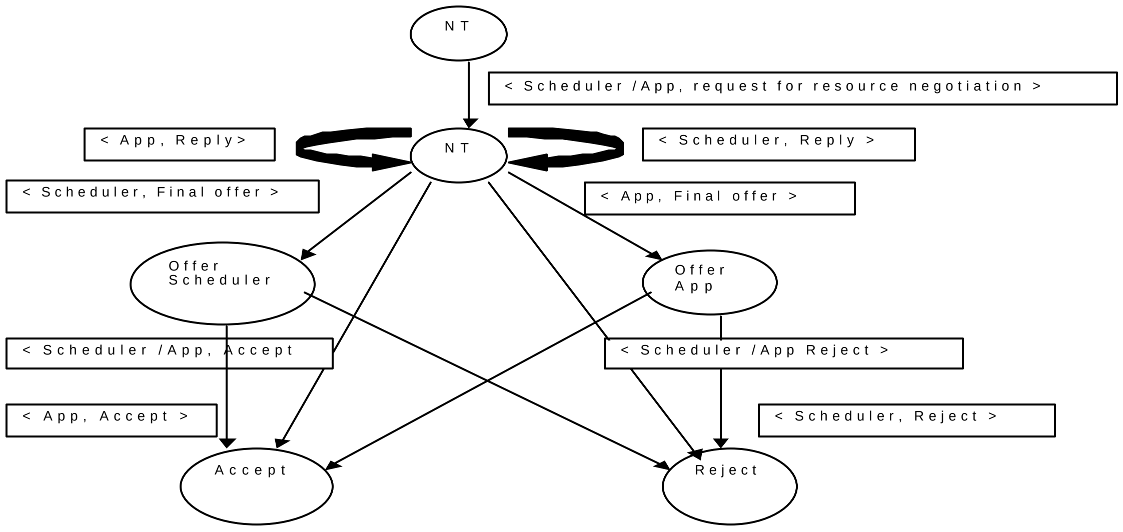

Fig. 2. Finite State Representation of Resource Negotiation Protocol

The methodology or programming model for developing adaptive applications is orthogonal to the execution of RMS or the scheduling algorithms, as long as they communicate with the RMS using a standard protocol. The test application set consists of a rigid, a malleable and an evolving version of a parallel N-body simulation code [9]. The structure for the coordinator process of the test evolving application is as follows:

1. Start the evolving application
2. Run application specific computation
3. When additional resources are required stop computation
4. Connect to the negotiation manager using protocol API
5. Request additional resources and carry out negotiation
6. If negotiation is successful adapt and redistribute data if necessary, resume computation else, continue computations
7. Repeat step 2 – 6 as necessary
8. End Application

In the case of a malleable application the RMS contacts the application and requests it to either expand or shrink. In malleable application the coordinator process starts a separate negotiation thread, which waits for request from RMS for adaptation. When there is a request for adaptation this thread negotiate resources with the RMS. If negotiation is successful it informs the coordinator process. The structure for the coordinator process of the test malleable application is as follows:

1. Start the malleable application
2. Start negotiation thread.
3. Check for adaptation request
4. If adaptation is requested, then adapt and redistribute data if necessary.
5. Run application specific computation
6. Go to step 3 until all the computations are complete.
7. Stop negotiation thread
8. End Application

Several experiments were conducted on the test bed; the first experiment involved the evolving application. The application was written in such way that after one third of the computation the application asks for additional resources and enters into negotiation with the scheduler. If additional processors are allocated it expands and utilizes the additional processors. If the request is rejected, it continues to execute with the current number of processors. The application started on 2 processors and the rest of the processors were idle. The application asked for 2 additional processors, the scheduler allocated the additional processors, and the application expanded to 4 processors. The experiment was repeated, but this time the application asked for 14 additional processors, the scheduler offered 6 processors and the application accepted the offer and expanded to 8 processors. The next experiment involved a malleable application. It was written in such a way, that if it is offered additional processors it enters into negotiation with the scheduler. It accepts the additional processors as long as the total number of processors is a power of 2. The application started on four processors and a short running rigid application started immediately on four other processors. When the rigid application completed its execution, the scheduler offered the 4 idle processors to the malleable application, which accepted the offer and expanded to 8 processors. These tests validate experimentally that the RMS is capable of scheduling adaptive applications and it handles different scenarios of negotiation.

The gain of running an adaptive application as opposed to running it in rigid mode depends on several factors: total running time, scalability, cost of data redistribution, scheduling overhead, as well as negotiation and communication cost. To measure the overhead of the communication, a dummy application was executed on the test bed. The dummy application created an NT and sent it back and forth to the RMS. The number of negotiation rounds has been parameterized. The total time for negotiation (opening connection + send and receiving NT + closing connection) was measured for different numbers of negotiation rounds. The overhead is shown in Table 1. This overhead is independent of scheduling overhead or application runtime; it depends on the number of negotiation rounds and is very low, especially compared to a typical long running parallel application.

Table 1. Communication overhead

No. of Negotiation Rounds	1	*2*	*3*	*4*	*5*
Time in Milliseconds	0.03	0.04	0.12	0.20	0.28

7. Summary and Future Work

We believe adaptive applications are the next generation of parallel applications and promise to improve the performance of parallel systems. When a system load is high, instead of halting an adaptive application, its resources can be reduced to allow the execution of higher priority applications. Similarly, adaptive applications can be expanded to utilize idle resources, thus increasing the performance of the system as well as the applications themselves. Also, present work on adaptive applications open up the possibility of the development of a new class of parallel applications driven by unpredictable data and events, which is very difficult in current distributed environments. Current RMSs are unable to handle adaptive parallel applications efficiently. This paper discussed the requirements of a RMS capable of managing such applications. Based on these requirements an architecture has been proposed and a proof of concept system has been developed. Experiments with prototype implementation show that the RMS is capable of handing adaptive applications and that the overhead of communication and resource negotiation is very low.

The development of the RMS is part of a larger effort to develop a complete infrastructure required to schedule adaptive applications to optimize some objective function in a cluster environment. The prototype implementation was done to prove the concept. Future work includes a full blown robust implementation of the RMS. Experiments with large workloads on a larger cluster are necessary to discover and resolve any other issues in the RMS design. Lack of availability of parallel adaptive applications is one of the major problems of testing the RMS. The current implementation handles requests from evolving applications on a first come first serve basis. However with larger workloads there may be multiple simultaneous requests, or the frequency of requests may be high and unpredictable. The issue of handling a large volume of requests efficiently requires further experiments and research. Absence of a well-defined programming model and lack of infrastructure support is the major obstacle for application developers to write adaptive applications. On the other hand, since there is an absence of a large number of adaptive applications, researchers have not been sufficiently motivated to develop infrastructure support yet. The development of the RMS is an attempt to break this cycle and move towards building complete adaptive systems that address all aspects of developing and executing adaptive applications in cluster environments.

* This research is funded by NSF grant EIA 0103594

References

1. D. G. Feitelson and L. Rudolph, "Towards convergence in job scheduling for parallel super computers," in Job Scheduling Strategies for Parallel Processing, Vol. 1162, Lecture Notes in Computer Science D. G. Feitelson and L. Rudolph Eds. Springer-Verilag, 1996, pp 1-26.
2. C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. IEEE Transactions on Software Engineering, 11(10):1001-1016, Oct. 1985.
3. C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. IEEE Transactions on Computers, 36(12):1425-1439, Dec. 1987.
4. T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel computers. IEEE Transactions on Parallel Distributed Systems, 4(1):87-98, Jan. 1993.
5. S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A method for scheduling parallel loops. Communications of the ACM, 35(8):90-101, Aug. 1992.
6. E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 5(4):379-400, Apr. 1994.
7. I. Banicescu and S. F. Hummel. Balancing processor loads and exploiting data locality in n-body simulations. In Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM). ACM Press, 1995.
8. S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures, pages 318-328, 1996.
9. I. Banicescu and R. Lu. Experiences with fractiling in n-body simulations. In Proceedings of High Performance Computing 98 Symposium, pages 121-126, 1998.
10. I. Banicescu and Z. Liu. Adaptive factoring: A dynamic scheduling method tuned to the rate of weight changes. In Proceedings of the High Performance Computing Symposium (HPC 2000), pages 122-129, 2000.
11. I. Banicescu and V. Velusamy. Load balancing highly irregular computations with the adaptive factoring. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002) - Heterogeneous Computing Workshop. IEEE Computer Society Press, 2002.
12. I. Banicescu, V. Velusamy, and J. Devaprasad. On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. Cluster Computing: The Journal of Networks, Software Tools and Applications, 6(3):215-226, 2003.
13. R. L. Carino, I. Banicescu, R. K. Vadapalli, C. A. Weatherford, J. Zhu, "Message-passing parallel adaptive quantum trajectory method. High performance Scientific and Engineering Computing: Hardware/Software Support," L. T. Yang and Y. Pan (Editors). Kluwer Academic Publishers, 2004, pp. 127-139
14. P.-F. Dutot, G. Mounie and D. Trystram, "Scheduling Parallel Tasks – Approximation Algorithms", Handbook of Scheduling: Algorithms, Models, and Performance Analysis," Edited by Joseph Y-T. Leung Published by CRC Press, Boca Raton, FL, USA, 2004
15. The Condor Project Homepage, http://www.cs.wisc.edu/condor/
16. M.L. Massie, B.N. Chun, D.E. Culler, "The Ganglia distributed monitoring system: Design, implementation and experience" Parallel Computing, vol. 30, Issue 7, July 2004.
17. MPI Software Technology, Inc., ClusterController™ User Guide Version 1.0.1, 2001.
18. Portable Batch System. http://www.openpbs.org

19. Platform LSF Family of Products. http://www.platform.com/products/LSFfimily.
20. Maui Cluster Scheduler. http://www.clusterresources.com/products/maui
21. Moab Cluster Suit. http://www.cluterresources.com/products/moabclustersuit.shtml
22. S.S. Vadhiyar and J. Dongarra, "SRS: A framework for developing malleable and migratable parallel applications for distributed systems," Parallel Processing Letters, Vol. 13, No. 2 (2003) 291-312
23. J. E. Moreira and V. K. Naik, "Dynamic Resource Management on Distributed Systems Using Reconfigurable Applications," IBM Journal of Research and Development, Vol. 41, No. 3, May 1997, pp 303 – 330.
24. L. V. Kale, S. Kumar, and J. DeSouza, "A Malleable-Job System for Timeshared Parallel Machines," 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002), May 21-24, 2002, Berlin, Germany.
25. G. Edjlali, G. Agrawal, A. Sussman and J. Saltz, "Data Parallel Programming in an Adaptive Environment", in Proceedings of the Ninth International Parallel Processing Symposium, April 1995, pages 827-832. IEEE Computer Society Press.
26. R. Jha, M. Muhammad, S. Yalamanchili, K. Schwan, D. Ivan Rosu, and C. de Castro, "Adaptive resource allocation for embedded parallel applications," in Proceedings of the 3rd International Conference on High Performance Computing", Trivandrum India, December 1996.
27. S. Ghafoor, T. Haupt, S. Gosula, "A Communication Protocol for Adaptive Parallel Applications in Cluster Environments," accepted for publication in Proceedings of High Performance Computing Symposium (HPC 2005), to be held in San Diego in April 3-7, 2005.
28. Vortal Research Group, "User Guide for Resource Negotiation Protocol for Adaptive Parallel Applications", Version 1.0, http://www.erc.msstate.edu/~ghafoor/vortal/RNP_UserGuide.pdf.