Extensible Resource Scheduling for Parallel Scientific Applications

Nayeem Islam* Andreas Prodromidis† Mark S. Squillante* Ajei S. Gopal*
Liana L. Fong*

Abstract

The resource requirements and processing characteristics of parallel scientific applications are quite diverse. In this paper, we present a new resource management approach for scheduling such parallel applications that combines multiple scheduling paradigms with a fault-tolerance paradigm into a coherent system. Results from a prototype implementation demonstrate that our system provides significant performance improvements over existing methods in a controllable and extensible manner.

1 Introduction

The class of parallel scientific applications is extremely large and diverse. Such applications, together with different instances of each application, exhibit a wide range of speedup and efficiency characteristics. The communication and synchronization traits among the constituent tasks of these parallel applications are equally diverse, with one extreme consisting of *fine-grained* tasks that frequently communicate and/or synchronize, and the other extreme consisting of *coarse-grained* tasks that are independent. Moreover, current and expected workloads for large-scale scientific computing environments consist of various mixtures of parallel applications with very different resource requirements.

Parallel processing systems make it possible to execute this diverse class of scientific applications in significantly less time and to solve larger and more complex instances of these applications. The manner in which the system resources are allocated to the tasks of such parallel applications is fundamental to realizing these potential performance benefits, as well as to minimizing mean job response time and maximizing system throughput. Given the wide variety of parallel scientific applications each with different resource requirements, there is no single scheduling paradigm that can achieve these goals for all of the application requirements. As a specific example, the communication and synchronization characteristics of fine-grained applications typically require the simultaneous allocation of their tasks on computing nodes, whereas the independent tasks of coarse-grained applications are scheduled more effectively under a very different approach.

A number of scheduling strategies have been proposed for these parallel computing environments, each differing in the way nodes are shared among the jobs submitted for execution. One important scheduling paradigm, which is tailored primarily for fine-grained applications, is the class of *space-sharing* policies where the nodes are partitioned among different parallel jobs. This class includes static partitioning (i.e., the nodes are divided into a fixed number of disjoint sets), adaptive partitioning (i.e., the number of nodes allocated

*IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.
†Computer Science Department, Columbia University, New York, NY 10027.

to a job is determined when jobs arrive and depart based on the current system state), and dynamic partitioning (where the number of nodes allocated to a job can also be modified during its execution). The advantages of static partitioning are its low system overhead and its simplicity from both the system and application viewpoints, but this approach can lead to relatively low system throughputs and resource utilizations under nonuniform workloads. Adaptive partitioning tends to outperform such static policies by adjusting allocated partition sizes to the current load, although its performance benefits can be limited due to its inability to modify scheduling decisions in response to subsequent workload changes. Dynamic partitioning alleviates these problems by adjusting the size of partitions allocated to applications throughout their execution, at the expense of increased overhead and the additional support needed for this form of application reconfiguration.

Another important scheduling paradigm primarily for fine-grained applications is the class of *gang scheduling* policies that combines space-sharing together with time-sharing (i.e., the nodes are rotated among a set of jobs in time). Although a pure time-sharing policy is not suitable for most parallel scientific computing environments due to factors such as efficient resource utilization, several commercial systems have employed the combination of time-sharing and space-sharing in various forms. In addition to the benefits of space-sharing, the time-sharing aspect of gang scheduling can provide very efficient response times for short jobs and it can reduce the overall mean job response time (for a given partitioning of the nodes), both of which are particularly important under workloads with highly variable service requirements. These performance benefits are potentially obtained at the expense of increased overhead for context switching among different parallel jobs and fragmentation of the node allocations. Moreover, gang scheduling has proven to be difficult and/or expensive to implement due to its multi-node, simultaneous context switches.

An important scheduling paradigm tailored primarily for coarse-grained applications is the class of resource allocation schemes in which the tasks of arriving jobs are assigned to particular nodes and each node independently and concurrently executes the tasks assigned to it. Many different approaches have been employed to allocate resources under this paradigm, each with varying and limited degrees of success. However, an optimal scheduling strategy for this parallel environment has been recently established, which has been shown to provide significant performance improvements over other methods for coarse-grained applications [8, 3]. We refer to this paradigm as *load sharing*.

Our objective in this paper is to present the design of a new resource management approach that combines the space-sharing, gang and load-sharing scheduling paradigms into a coherent and efficient system, and that integrates these scheduling paradigms with a fault-tolerance paradigm. Our system effectively supports the diverse resource requirements of different parallel scientific applications in a controllable manner to achieve the best and desired levels of system performance. We present an overview of our resource management system in §2. A few experimental results from our prototype implementation are provided in §3, and our concluding remarks are presented in §4.

2 Resource Scheduling System Design

Our resource scheduling system, which is based on a design and implementation study that explored scheduling and fault-tolerance services for large-scale distributed and parallel computing environments [1], has many diverse aspects and makes several important contributions. The system design is based on a hierarchical scheduling architecture that supports and integrates the dynamic co-existence of multiple resource management paradigms into a coherent and efficient system. A fundamental aspect of our approach is a new strategy for dynamically adjusting the partitioning of nodes at all levels of the system hierarchy in a controllable manner while reducing the overheads associated with dynamic partitioning in large-scale parallel environments. Our system also provides a new gang scheduling approach based on the use of these dynamic partitioning methods, and on a novel scheme for multi-node simultaneous context switching that exploits logical clocks; this scheme provides efficient, controllable time-sharing in a fully distributed manner while eliminating the potential bottleneck and overheads of centralized approaches. Another fundamental aspect of our approach is a new load-sharing scheduling strategy that efficiently yields the optimal performance for coarse-grained parallel applications. Our design integrates fault-tolerance together with these scheduling paradigms so that the system is fault tolerant and it provides such services to applications. The system is also designed to be extensible so that it can accommodate a variety of additional scheduling and fault-tolerance paradigms.

In the interest of space, we focus here on certain elements of our resource management system design. Specifically, we first present a brief overview of the system architecture, and then we describe certain aspects of our dynamic partitioning and gang scheduling strategies. We refer the interested reader to [3] for more details on these elements of our resource management system, as well as a description of the other system components.

2.1 System Architecture Overview

Our resource scheduling system architecture is hierarchical. At the top level of the hierarchy, the system consists of a set of non-overlapping *domains* $\mathbf{D} \equiv \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_{\mathcal{N}^{\mathbf{D}}}\}$ into which the administrator divides the system resources to be managed independently, $\mathbf{U} \equiv \bigcup_{d=1}^{\mathcal{N}^{\mathbf{D}}} \mathcal{D}_d$. Each domain $\mathcal{D}_d$ is composed of $P_d^{\mathbf{D}}$ computing nodes that are divided into $\mathcal{N}_d^{\mathbf{P}}$ disjoint *partitions* $\mathbf{P}_d \equiv \{\mathcal{P}_1^d, \mathcal{P}_2^d, \ldots, \mathcal{P}_{\mathcal{N}_d^{\mathbf{P}}}^d\}$, $\bigcup_{p=1}^{\mathcal{N}_d^{\mathbf{P}}} \mathcal{P}_p^d = \mathcal{D}_d$, $1 \leq d \leq \mathcal{N}^{\mathbf{D}}$,[‡] where the node is the unit of resource allocation. Let P denote the total number of computing nodes in the system, and thus $\sum_{d=1}^{\mathcal{N}^{\mathbf{D}}} P_d^{\mathbf{D}} = P$. Each partition $\mathcal{P}_p^d$ consists of P_p^d nodes, $1 \leq p \leq \mathcal{N}_d^{\mathbf{P}}$,[‡] $\sum_{p=1}^{\mathcal{N}_d^{\mathbf{P}}} P_p^d = P_d^{\mathbf{D}}$, which can be further divided into $\mathcal{N}_{d,p}^{\mathbf{K}}$ disjoint *subpartitions* $\mathbf{K}_p^d \equiv \{\mathcal{K}_1^{d,p}, \mathcal{K}_2^{d,p}, \ldots, \mathcal{K}_{\mathcal{N}_{d,p}^{\mathbf{K}}}^{d,p}\}$, $\bigcup_{k=1}^{\mathcal{N}_{d,p}^{\mathbf{K}}} \mathcal{K}_k^{d,p} = \mathcal{P}_p^d$, depending upon the policies associated with the partition. These subpartitions can continue to be recursively refined into smaller, non-overlapping partitions. The bottom level of the hierarchy consists of P single-node partitions. Although not considered in this paper, inter-domain scheduling can be easily incorporated in our hierarchical system architecture.

The system administrator sets the number of domains, the size of each domain and the collection of scheduling and fault-tolerance paradigms employed in each domain, all of which remain fixed until the administrator explicitly modifies these settings. For each partition $\mathcal{P}_p^d$ the system administrator specifies the minimum, m_p^d, and maximum, M_p^d, size of the partition. The administrator also sets several additional parameters and thresholds that control the decisions and behavior of our resource management system, some of which are described below.

Associated with each domain is a *domain-level scheduler* (DLS) that is responsible for managing and allocating the set of resources in the domain. Each DLS maintains the list of computing nodes within its control, the scheduling paradigm supported by each of the

[‡]Throughout this paper it will be assumed that $1 \leq d \leq \mathcal{N}^{\mathbf{D}}$, $1 \leq p \leq \mathcal{N}_d^{\mathbf{P}}$ and $1 \leq k \leq \mathcal{N}_{d,p}^{\mathbf{K}}$.

partitions under its control, and the fault-tolerance characteristics for the entire domain. The DLS for $\mathcal{D}_d$ defines the manner in which arriving jobs are assigned to the partitions $\mathcal{P}_p^d$. For example, the DLS may decide that a particular scheduling paradigm is best for a job based upon its various characteristics (specified by the user when the job was submitted) and the current state of the domain. If no suitable partition is available when a job arrives (e.g., there are a sufficient number of partitions, each of which already have enough work assigned to them), the DLS enqueues the job for later consideration.

The DLS for $\mathcal{D}_d$ also defines and dynamically adjusts the size of its partitions $\mathcal{P}_p^d$ under the constraint that $m_p^d \le P_p^d \le M_p^d$. An instance of our new dynamic partitioning mechanism (see Section 2.2) is employed together with a special reallocation function to determine *when* and *how* nodes are moved among the partitions of a domain to achieve the level of performance specified by the system administrator. This DLS function can use a number of different weights and measures (which are monitored by the system) to trigger and perform the repartitioning process. As a specific example, in our current prototype implementation [3] the system administrator specifies a threshold $T_{i,j}^d$ for the load differential $\Delta_{i,j}^d \equiv \|U_{\mathcal{P}_i^d} - U_{\mathcal{P}_j^d}\|$ between each pair of partitions $\mathcal{P}_i^d$ and $\mathcal{P}_j^d$ (or the same threshold can be used for all partitions, i.e., $T_{i,j}^d = T^d$), $1 \le i, j \le \mathcal{N}_d^{\mathbf{P}}$. The DLS attempts to maintain each partition size P_p^d within its specified range (i.e., $m_p^d \le P_p^d \le M_p^d$) so that the pairwise differences in partition utilizations do not exceed their specified thresholds (i.e., $\Delta_{i,j}^d \le T_{i,j}^d$). Whenever $\Delta_{i,j}^d > T_{i,j}^d$, the DLS moves resources from underutilized partitions to overutilized partitions, subject to the partition constraints, using our dynamic partitioning mechanism. The repartitioning of resources among the partitions occurs dynamically as the DLS periodically re-evaluates the state of the system.

The set of resources in each partition of a domain are managed by a *partition-level scheduler* (PLS) in a manner similar to the way that the DLS coordinates the scheduling of the resources within its control, but in accordance with the particular scheduling paradigm employed in the partition. The PLS for $\mathcal{P}_p^d$ defines the mapping of all of the jobs it receives from its parent DLS to its subpartitions $\mathcal{K}_k^{d,p}$ (possibly creating a subpartition), $1 \le k \le \mathcal{N}_{d,p}^{\mathbf{K}}$.‡ This mapping depends upon a number of important factors, including the characteristics of the parallel application, the current state of the subpartitions $\mathcal{K}_k^{d,p}$, and the current workload of the partition $\mathcal{P}_p^d$. The PLS for $\mathcal{P}_p^d$ also defines and dynamically adjusts the number of its subpartitions $\mathcal{K}_k^{d,p}$ and the size of these subpartitions, both according to their states. It further provides any coordination among these subpartitions required by the scheduling paradigm associated with the partition. Any repartitioning decisions percolate down the hierarchy at pre-set intervals specified by the systems administrator. Each subpartition may itself be recursively refined into smaller, non-overlapping subpartitions.

A *node-level scheduler* (NLS) manages the single-node subpartitions at the leaves of the scheduling architecture hierarchy. Each NLS is responsible for running the processes of an application on its node by allocating the node-level system resources to the active processes assigned to it by the parent PLS. The NLS implements the scheduling policies and system attributes set by its PLS and DLS. When the NLS is assigned a task by its parent PLS, the NLS inserts the task into the local dispatcher queue in a location that satisfies the PLS scheduling requirements; e.g., an NLS in a gang scheduling partition inserts the task in coordination with the other NLSs in the partition. To assist in the collection of resource usage information employed throughout our resource management system, each NLS also monitors a number of important measures (including execution times, memory usage and effective utilization for each node) and exchanges this information with its parent PLS as part of their regular communication.

Another important component of our resource scheduling system design is an application level manager (ALM) associated with each job, which is the point of contact between the system and the application. For example, the PLS and an application's ALM communicate and work together to handle any node allocation changes resulting from either a node fault or a reconfiguration (where the application gains or loses nodes due to changes in the subpartition size).

2.2 Dynamic Partitioning

Each DLS dynamically adjusts the partitioning of resources within its control, as described above. The repartitioning of resources can also occur at other levels of the scheduling architecture hierarchy. In particular, a PLS that employs a dynamic partitioning paradigm can provide the best system performance for certain application workloads (see §3), since the dynamic policy can maintain very efficient node utilizations by adjusting node allocation according to workload changes. However, for other application workloads, the overheads associated with greedy dynamic partitioning in distributed environments can generally eliminate its potential performance benefits [7, 2]. In this section we describe a new approach for dynamically adjusting the partitioning of resources at any level of our resource scheduling system hierarchy in a controllable manner while reducing the overheads associated with dynamic partitioning in large-scale parallel environments.

One dynamic partitioning strategy for decreasing these performance penalties consists of using the *folding* approach found in [4], which reduces the number of reconfigurations performed under dynamic partitioning, at the expense of a less optimal allocation of the nodes among the competing jobs. We propose another approach, called *dynamic partitioning with smoothing* (DPS), that consists of using a more optimal allocation of the nodes whenever a repartition is performed, while enforcing a minimum interval of time between changes in node allocations, which can be set by the system administrator and can also be dynamically adjusted. This approach reduces the number of reconfigurations, decreases the total costs of repartitioning by performing multiple reconfigurations together, and prevents a form of reconfiguration thrashing [7]. It also tends to reduce the adverse effects of highly variable job arrivals, and tends to result in better system performance by batching multiple allocation decisions.

Our resource management system uses this DPS approach to determine the size of each partition $\mathcal{P}_p^d$ comprising every domain $\mathcal{D}_d$, as well as the size of each subpartition $\mathcal{K}_k^{d,p}$ comprising every partition $\mathcal{P}_p^d$ that dynamically adjusts the division of its resources; this DPS strategy can also be used to dynamically adjust resource allocations at any other level of the system hierarchy that requires such repartitioning of resources. In each case, system performance can be improved considerably by exploiting service time and efficiency information for each application workload. However, in the absence of such workload information, the nodes are divided into equal-sized partitions, which tends to outperform other partitioning strategies when job service time requirements are not used in scheduling decisions. In the presence of application workload information, our DPS method attempts to divide the nodes into partitions with equal node utilization characteristics based on the corresponding information provided by the resource monitoring of our system. This is achieved by maintaining an ordered list of the partitions based on a general (extensible) function of their utilization characteristics. When the number of partitions increases, new partitions are populated with nodes taken from those at one end of the list, whereas if the number of partitions decreases, nodes are given to those at the other end of the list.

2.3 Gang Scheduling

Gang-scheduling is most effective when its time-sharing and space-sharing components are integrated to yield the advantages of both while eliminating their disadvantages. Our resource management system integrates the time-sharing and space-sharing paradigms in a very natural and efficient manner, which can be conceptually viewed as a generalization of the global matrix originally proposed by Ousterhout [6]. The number of *columns* in the matrix equals the number of nodes reserved for the gang scheduling partition, and these columns are divided into K^{GS} disjoint subpartitions $\mathcal{K}_k^{GS}$, $1 \leq k \leq K^{GS}$. The number of columns in $\mathcal{K}_k^{GS}$ represents the number of nodes P_k^{GS} allocated to the parallel applications assigned to $\mathcal{K}_k^{GS}$ by the gang scheduling PLS, whereas the number of rows in $\mathcal{K}_k^{GS}$ represents the degree of time-slicing D_k^{GS} for $\mathcal{K}_k^{GS}$. Each row in the global matrix represents the mapping of a set of applications to a set of subpartitions, where each application is mapped to only one subpartition. The time-slice, or *quantum*, length for each row of a subpartition $\mathcal{K}_k^{GS}$ is denoted by $T_{k,i}^{GS}$, $1 \leq i \leq D_k^{GS}$, $1 \leq k \leq K^{GS}$. The processes of an application assigned to a specific subpartition are executed at approximately the same time on all nodes of the subpartition via the strategy described below. Our approach also supports the allocation of multiple quanta to smaller job classes for each quantum allocated to larger job classes (i.e., smaller jobs can be placed in multiple rows of a subpartition of the matrix) so that small jobs receive very efficient response times and the overall mean response times are reduced, both in a controllable manner [9].

This organization for the gang-scheduling matrix is due in part to our distributed gang-scheduling design, which is in stark contrast to previous methods that employ centralized, tightly-coupled and/or hardware-based control mechanisms. Our approach consists of a general, software-based mechanism that is used to provide coordinated context-switching across the system nodes. Specifically, each node independently time-slices among the jobs allocated to it according to its local *logical* "time", and synchronized clocks are used to maintain consistent time across the nodes [5]. Since each node switches independently based on its local clock, the actual overhead of a simultaneous multi-node context switch in our system is no more expensive than a local context switch. A number of different methods have been implemented for synchronizing clocks, any one of which can be used to obtain an efficient and portable mechanism for keeping the local clocks on each node synchronized. Moreover, by having a fixed and independent degree of time-slicing for each subpartition, node fragmentation is significantly reduced and each node appropriately responds to job completions in an independent manner. This structure also reduces considerably the complexity to maintain the gang-scheduling matrix as jobs enter and especially when jobs leave the system.

The DPS mechanism of Section 2.2 is used to maintain a proper value for K^{GS} where the smoothing interval between repartition epochs will be larger than that for a pure dynamic partitioning PLS, since repartitioning in this case implies a reconstruction of (portions of) the gang-scheduling matrix. The cost of such repartitioning can be further reduced by performing all of the reconfigurations for each subpartition $\mathcal{K}_k^{GS}$ in parallel, and the draining of columns due to departures can be used to reduce the number of reconfigurations required for each subpartition. Additional context-switching overheads (e.g., memory and cache effects) are kept relatively low by using sufficiently coarse quantum lengths. During each smoothing interval (in which K^{GS} is fixed), the system maintains a high-level of performance by increasing the values of D_k^{GS} in response to increasing load conditions subject to memory constraints and the current utilization of the system. When a job in $\mathcal{K}_k^{\text{GS}}$ departs, the PLS assigns a waiting job to this subpartition and the value of D_k remains the same. If there are no waiting jobs, however, the value of D_k is decremented independently in the PLS and the corresponding NLSs, and thus the values of D_k^{GS} naturally decrease with reductions in system load during each smoothing interval. These mechanisms are used to maintain a proper D_k^{GS} value for each subpartition, which is further adjusted according to the application workload efficiency, the variability of workload service demands and the system load.

3 Experimental Results

We have developed a prototype implementation of the above resource management system design that runs on the IBM SP2 and a cluster of workstations. Several parallel scientific applications, and diverse workloads based on mixtures of these applications, were used to study a large number of scheduling performance issues on this experimental platform. In this section we briefly present a small subset of our experimental results. We refer the interested reader to [3] for the details on our prototype implementation and application workloads, as well as for a detailed description of the complete set of experimental results.

A fundamental premise of our system design is that no single scheduling paradigm provides the best performance for all parallel application workloads. To demonstrate this, we conducted numerous experiments with a wide variety of scientific application workloads, including a highly efficient and variable workload (W1), a workload with relatively low efficiency and variability (W2), and a workload composed of coarse-grained applications (W3). Our results show that a pure time-sharing strategy (i.e., $K^{\text{GS}} = 1$) provides the best performance for W1, a pure space-sharing scheme (i.e., $D_k = 1$) provides the best performance for W2, and our load-sharing policy provides the best performance for W3.

We next consider the performance characteristics of different dynamic partitioning strategies under W2. In Fig. 1 we plot the response time of greedy dynamic partitioning (i.e., DPS with a smoothing interval of 0), folding and DPS with a 30 second smoothing interval, all normalized by the greedy policy response time. We note that DPS consistently outperforms the other space-sharing schemes across the range of utilizations considered. These results can be explained by the observations made in §2.2.

Our next set of results considers the performance characteristics of different versions of gang scheduling under W1. In Fig. 1 we also plot the response time under gang scheduling for different values of K^{GS} and $D_k = D_K$, $1 \leq k \leq K^{\text{GS}}$. We observe that $K^{\text{GS}} = 1$ is optimal at light loads, whereas $K^{\text{GS}} = P$ provides the best performance at high utilizations. In both cases, we further observe that the D_k values should increase as the load rises within the region where a specific value of K^{GS} is optimal. Note that larger values of D_k tend to reduce job queueing times at the expense of longer execution times, and that the best D_k value for a particular (steady-state) load balances this delicate tradeoff between queueing delays and execution times.

Our resource management system exploits the DPS strategy to efficiently maintain the best value of K^{GS}, as discussed in §2.3. Within each range of utilizations for a fixed value of K^{GS}, a small value of D_k is used at the low end of the range and this value is increased as the load rises within the range (and vice versa), as described in §2.3. The system monitoring facilities are used to watch for changes in the characteristics of the workload, and the system adjusts these key gang scheduling parameters accordingly.

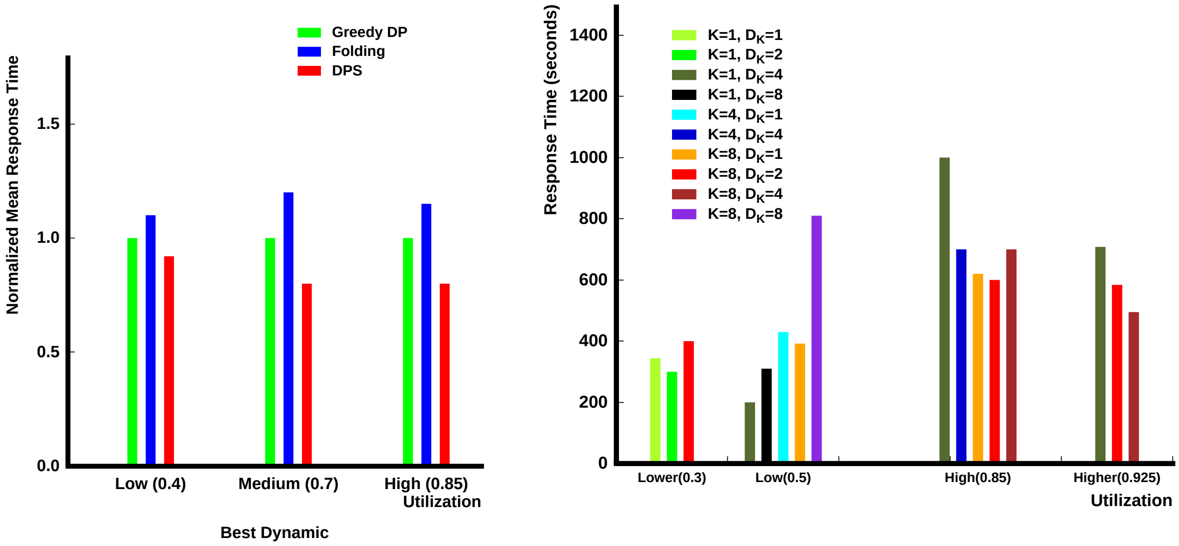

FIG. 1. *Results for Dynamic Partitioning and Gang Scheduling Paradigms with* $P = 8$.

4 Conclusions

We presented the design of a new resource management system that effectively supports the diverse resource requirements of different parallel scientific applications in a controllable manner. Our approach includes a hierarchical scheduling architecture that integrates the dynamic co-existence of multiple paradigms into a coherent system, as well as novel strategies for gang scheduling and for dynamically adjusting resource partitioning at all levels of the system hierarchy. A prototype implementation was used to demonstrate that our approach provides significant performance benefits over existing methods.

References

[1] S. A. Fakhouri, L. L. Fong, A. S. Gopal, N. Islam, J. A. Pershing, and M. S. Squillante, *Milliways scheduling and membership design*, Tech. Rep., IBM Research Division, Nov. 1994.

[2] N. Islam, A. Prodromidis, and M. S. Squillante, *Dynamic partitioning in different distributed-memory environments*, in Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1996. Lecture Notes in Comp. Sci. Vol. 1162.

[3] N. Islam, A. Prodromidis, M. S. Squillante, A. S. Gopal, and L. L. Fong, *Extensible resource management for cluster computing*, Tech. Rep. RC 20526, IBM Research Division, May 1996.

[4] C. McCann and J. Zahorjan, *Processor allocation policies for message-passing parallel computers*, in Proc. ACM SIGMETRICS Conf. Meas. Mod. Comp. Sys., May 1994, pp. 19–32.

[5] D. L. Mills, *Improved algorithms for synchronizing computer network clocks*, IEEE Trans. Networks, (1995), pp. 245–254.

[6] J. K. Ousterhout, *Scheduling techniques for concurrent systems*, in Proc. Third Int. Conf. Dist. Comp. Sys., Oct. 1982, pp. 22–30.

[7] M. S. Squillante, *On the benefits and limitations of dynamic partitioning in parallel computer systems*, in Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), Springer-Verlag, 1995, pp. 219–238. Lecture Notes in Comp. Sci. Vol. 949.

[8] M. S. Squillante and K. P. Tsoukatos, *Optimal scheduling of coarse-grained parallel applications*, in Proc. Eighth SIAM Conf. Par. Processing Sci. Comp., Mar. 1997.

[9] M. S. Squillante, F. Wang, and M. Papaefthymiou, *Stochastic analysis of gang scheduling in parallel and distributed systems*, Perf. Eval., 27&28 (1996), pp. 273–296.