Extensible Resource Management For Cluster Computing

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

Abstract

Advanced general-purpose parallel systems should be able to support diverse applications with different resource requirements without compromising effectiveness and efficiency. We present a new resource management model for cluster computing that allows multiple scheduling policies to co-exist dynamically. In particular, we have built Octopus, an extensible and distributed hierarchical scheduler that implements new space-sharing, gang-scheduling and load-sharing strategies. A series of experiments performed on an IBM SP2 suggest that Octopus can effectively match application requirements to available resources, and improve the performance of a variety of parallel applications within a cluster.

1 Introduction

Advanced parallel computer systems must be capable of supporting a variety of applications with diverse resource requirements and different scheduling needs. Several hierarchical schedulers have already been discussed in the literature [5, 6], but they all focus on special application domains. Many scheduling strategies for distributed memory parallel systems exist, but they are not general enough to accommodate diverse applications. Furthermore, most batch schedulers, e.g LoadLeveler, are single paradigm schedulers and is not designed to be extensible.

The design of *Octopus* recognizes that no one scheduling paradigm is best for all application requirements. It is, therefore, generic in the sense that it allows multiple different scheduling paradigms to co-exist dynamically in a hierarchical construction, and extensible in that it allows the easy importation of new scheduling paradigms. In this hierarchical scheduler we tackle two important problems for distributed memory parallel systems: how to map incoming applications to paradigms and how to manage resources used in each paradigm.

In one possible categorization, parallel applications can be classified as *tightly-synchronized* or *loosely-synchronized*. *Tightly-synchronized* applications are applications with threads that need to interact, synchronize and communicate frequently, hence they need to be scheduled on different nodes simultaneously [15]. Three main scheduling paradigms support applications with such requirements: those which *space-share* the available nodes by partitioning them among the different parallel jobs; those which *time-share* the nodes by rotating them among a set of jobs; and those which combine space-sharing and time-sharing.

Although several space-sharing and time-sharing strategies have been proposed and evaluated extensively, neither approach performs best under all workloads. For example, there are three partitioning policies for space sharing: (1) *static*, where partitions are fixed in size, (2) *adaptive* where the number of nodes allocated to a job is decided upon its arrival and depends upon the system state [12, 14], and (3) *dynamic* where partition sizes can change dynamically, and all of them have disadvantages under certain conditions. The first two are unable to adjust scheduling decisions in response to workload changes [14, 2], while the latter incurs increased overheads [3, 13]. Similarly, in time-sharing, certain conditions can force the system into low resource utilization.

On the other hand, several strategies that combine space-sharing and time-sharing, in what is often called *gang scheduling* [15, 4], provide the benefits of both space-sharing and time-sharing. Gang Scheduling can be effective with small/interactive jobs, and can reduce the mean job response time (for a given partitioning of the nodes) [16]. These advantages are very important especially under highly variable workloads although they come with the price of increased overhead for context switching among the parallel jobs and fragmentation of node allocations. *Loosely-synchronized* applications are applications with coarse-grained independent processes, and thus do not require simultaneous resource allocation.[1]

*IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.

†Computer Science Department, Columbia University, New York, NY 10027.

[1] A scheduling paradigm for loosely-synchronized applica-

Our resource management system makes the following contributions: (1) New space-sharing, gang-scheduling and load-sharing strategies; (2) An extensible design so that a user and/or system administrator can replace different aspects of our scheduling system with new paradigms to accommodate the demands of emerging applications; (3) A general strategy that integrates space-sharing, gang-scheduling and load-sharing in a controllable manner for scheduling incoming applications to the appropriate paradigm and (4) A dynamic partitioning mechanism that manages the allocation of resources among the multiple scheduling paradigms. *Octopus* is the first resource management system design and implementation to our knowledge that includes each of these fundamental properties.

In this paper in section 2 we discuss the hierarchical design and extensibility of our scheduler and the interactions between the scheduler and applications. Section 3 describes the scheduling algorithms and their integration in *Octopus*, and section 4 presents the implementation details. Performance measurements are discussed in Section 5 and conclusions in Section 6.

2 The Structure of the Scheduler

The top level (root) of the hierarchical scheduler is called the *domain*. The domain is controlled by a *domain level scheduler* (DLS), a "global" distributed subsystem responsible for the overall resource allocation across the available nodes. The DLS maintains a list of all nodes within its control, has a global view of the available resources, and interacts with the users.

A parallel application is submitted by a *launcher*, such as a shell. The launcher specifies to the DLS the application's resource and scheduling requirements. requirements include memory, the number of nodes and application type attributes. The domain is divided into multiple dynamically-changing non-overlapping *partitions*. Each partition is controlled by a *partition level scheduler* (PLS), a distributed subsystem much like the DLS that coordinates the scheduling of all the nodes in the partition. Different PLSs may support different scheduling paradigms. A partition may itself be recursively refined into smaller non-overlapping sub-partitions.

Each PLS has the following tasks: (1) It maps an incoming application passed onto it from a higher level scheduler to a lower level scheduler. (2)It manages its own sub-partitions dynamically according to their state. This means that it can create, destroy or re-allocate resources among sub-partitions (3) It cooperates (if necessary) with its lower level schedulers to implement a scheduling paradigm.

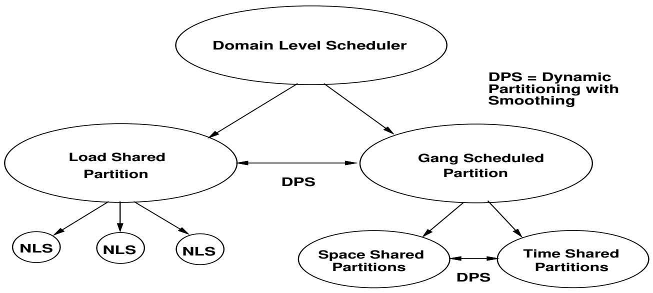

Figure 1: Applying the General Model to our Parallel Scheduling Algorithm

The bottom (leaves) of the hierarchy consists of one *node* partitions. Each one node partition is managed by a special *node level scheduler* (NLS). Each NLS is responsible for running the processes of an application on its node by allocating the node-level system resources to active processes. When the NLS is assigned a task by a PLS, the NLS inserts the task into the local dispatch queue in a location that satisfies the scheduling requirements. For example, an NLS in a gang scheduled partition is constrained to insert the task in coordination with the other NLSs.

Dynamic Partitioning with Smoothing (DPS) DPS is a new mechanism for creating and destroying partitions and for managing the allocation of the available resources among partitions. It employs a special reallocation function to determine *when* and *how* to re-allocate nodes. For example, it may use changes in partition efficiency, or partition utilization[2] to trigger resource re-allocation and move nodes from one partition to another, while satisfying a predefined performance objective. The information for the re-allocation function of the DPS is usually[3] monitored and provided by the system (e.g. CPU-utilization in the load-sharing partition). Each NLS supports a resource monitor module which monitors and reports execution times, memory usage and CPU-utilization

tions can also accommodate the considerable fraction of sequential applications that are part of any general-purpose computing environment.

[2] We approximate partition utilization by averaging on node utilizations.

[3] In some cases, the application itself can provide valuable information when it first enters the system. In a space sharing partition, for instance, where each application runs on a dedicated child partition, information such as efficiency may be supplied directly from the application.

for each node. With this information, the DPS mechanism can be implemented to support different dynamic partitioning algorithms so that the DLS and the PLSs manage their domain/partitions according to their own policies. A re-allocation decision at one level may propagate up or down the hierarchy.

Figure 1 shows our scheduling system with two different uses of DPS, one by the DLS and one by the PLS of the gang scheduling partition. In both cases, the input to the re-allocation function of the DPS is application efficiency and partition utilization; in contrast, the input to the re-allocation function of the DPS (not-shown) of the PLS managing the space shared partitions is just the application efficiency.

Scheduler Extensibility *Octopus* is an extensible system, designed using object-oriented frameworks [8, 7, 1, 9], that supports multiple scheduling paradigms and permits the easy implementation and incorporation of new paradigms. Partitions with new scheduling policies are created by specializing two independent interactions: between a parent PLS and a child PLS and between a PLS and its NLSs.[4]

One aspect of the parent PLS - child PLS interaction is related to mapping applications to children PLSs. The parent PLS invokes each child's `find_resource_match(Application*)` method to determine the appropriate partition. By customizing this method, the children PLSs direct the parent PLS with respect to the type of applications they support most effectively. After choosing the appropriate PLS, the parent PLS forwards the application to the child PLS by calling the latter's `take_application(Application*)` method.

With respect to the PLS - NLS interactions, the PLSs overload the following methods: (1) `plsSchedule()` for determining when an application should be scheduled, (2) `searchNodes()` for mapping an application to a set of nodes and (3) `launchApplication()` for sending an application to a node. For example, method `launchApplication()` is specialized to filter the amount of information given to each NLS; thus, in load-sharing it provides just the processes to be executed, whereas in gang scheduling it also sends a synchronization message to schedule a group of processes simultaneously.

Application-Scheduler Negotiations A major task for each PLS is to map an application to one of its children partitions. This mapping depends upon the characteristics of the incoming application, the current state of its partitions and the current workload of the system. Therefore a good characterization of an application would yield a better mapping. *Octopus* provides an interface to applications for specifying their properties (see Table 1) in order to obtain the appropriate types of resources, and an interface for responding to changes in resource allocation. Such properties include the type of application (reconfigurable and fault-tolerant applications are different in the sense that the first can handle loss of nodes only if they are notified in advance), whether it is tightly or loosely synchronized, a minimum and maximum number of nodes on which it can be executed, an estimate of its per-node memory requirements[5], a measure of its execution efficiency, and a classification of its execution time (e.g., in our experiments we used *very small* (less than 1 second), *small* (1 to 10 sec), *medium* (10 to 100 sec), *large* (100 to 1,000 sec), *very large* (over 1,000 to 10,000 sec) and *huge* (over 10,000 sec).

By execution efficiency we mean, in our examples, the application's speedup on the maximum number of nodes in the system.[6]

Property	Range
Type of Application	Legacy,Reconfigurable, Fault Tolerant, Sequential
Efficiency (Speedup)	Inefficient , Efficient
Execution Time	Very Small, Small, Medium , Large
Synchronization Type	Tightly synchronized
	Loosely synchronized
Number of Nodes	1- P
Memory Per Node	1 - M megabytes

Table 1: Application Properties and acceptable values of these properties

Overall Policy Initially, the DLS creates partitions for the paradigms for which the system administrator has specified minimum sizes and a single default partition with the rest of the resources. When a new application enters the system the DLS determine which of the currently instantiated partitions is more appropriate for this new application. When no suitable partition exists, the DLS creates an appropriate partition by taking nodes from existing partitions. If no partition can be created (e.g when there are already too many partitions), the DLS enqueues the application for later consideration. PLSs handle incoming applications the same way. Further resource re-allocation

[4]The DLS-PLS interaction is a special case of a parent PLS - child PLS interaction.

[5]Applications that do not fit into memory are rejected outright. Others are admitted based on policies presented later in the paper

[6]More complicated definitions could employ the continuous function of the execution efficiency or its first derivative.

among partitions occurs dynamically via DPS as the DLS/PLSs periodically re-evaluate the state of the system.

3 Integrating Policies

Octopus is designed to be extensible to accommodate different resource management paradigms. Here, we consider the integration of space-sharing, time-sharing, gang scheduling and load sharing paradigms into a single, coherent and efficient system.

Space Sharing The dynamic form of space sharing can provide the best system performance for a wide variety of application workloads, especially those with lower efficiencies and/or those with less variable service time requirements (as our results show).

Octopus uses the dynamic partitioning with smoothing approach [10] to determine the number of partitions in the system, denoted by K. According to this approach, the resource management system attempts to balance the node utilizations by reallocating the resources among the applications. Reallocations are triggered when applications arrive or exit the system but their frequency is limited by the SS_smooth parameter. Furthermore, *Octopus* can exploit service time and efficiency information provided by the applications and combine it with information provided by the monitoring facilities to considerably improve system performance.[7] The parameters of the space-sharing policy are given in Table 2. SS_{min} is the minimum size of a partition and SS_smooth the interval between reconfigurations.

Parameter	Policy	Example Value
SS_min	sys-admin	min. size of partition
SS_smooth interval	sys-admin	min. interval to re-allocate

Table 2: Space Sharing (SS) Scheduling Parameters

Time Sharing Unlike space-sharing, time-sharing provides the best system performance for highly efficient workloads, and for workloads with highly variable service demands. However, in most cases, the speedup of parallel jobs does not increase linearly with the number of processors allocated. This, in conjunction with the context-switching overheads result in poor utilization of resources in time-sharing systems.

[7]In the absence of application information (i.e. for portability reasons) this approach reduces to dynamic equi-partitioning with K equal-sized partitions approach which outperforms other space-sharing strategies.

Gang Scheduling – Time and Space Sharing Gang-scheduling combines the advantages of space-sharing and time-sharing. Our approach can be conceptually viewed as a generalization of the global matrix originally proposed by Ousterhout [15]. Each *column* represents one node and each *row* represents one timeslice. In our approach, the columns are divided into K disjoint *partitions* , or *groups*. The number of columns in the k^{th} partition represents the number of nodes S_k allocated to the parallel applications assigned to that partition, and the number of rows of the k^{th} partition represents its degree of multiprogramming D_k, $1 \leq k \leq K$. Naturally, this reduces to a space-sharing partition, when $D_k = 1$ and to time-sharing partition when K = 1. Each row in the matrix represents the mapping of a set of applications to the set of partitions. The processes of an application assigned to a specific partition are executed at approximately the same time[8]. The overall scheduling matrix is shown in Figure 2.

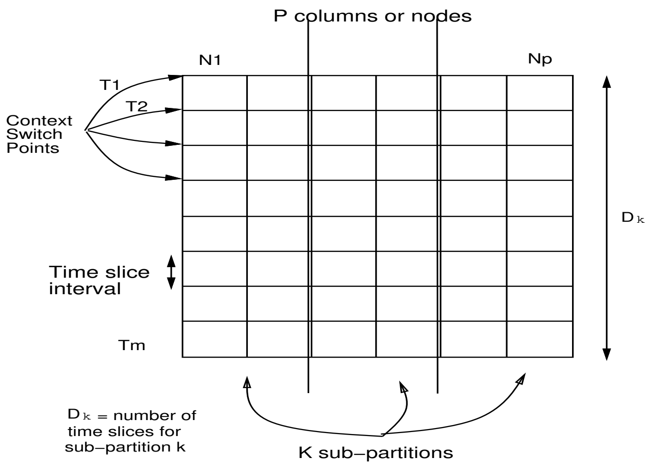

Figure 2: The gang scheduling matrix

In our implementation of the gang-scheduling paradigm, the K sub-partitions are divided into two separate groups, one that manages per-application space shared sub-partitions ($D_k == 1$) and another that manages a set of time-shared partitions ($D_k > 1$). The parent PLS managing the gang-scheduling partition uses the DPS mechanism to re-allocate the nodes between these two children PLSs. The system administrator specifies the minimum partition size as well as a threshold T for the load differential $\Delta \equiv \mid U_{SS} - U_{TS} \mid$

[8]Network Time Protocol (NTP) is used to keep the clocks synchronized.

between the utilization U_{SS} of the space-sharing paradigm and the utilization U_{TS} of the time-sharing paradigm. Whenever Δ becomes greater than the value of T, DPS attempts to move nodes between the two partitions to even out the differences (subject to the partition constraints). As with the space sharing policy, a special *PLS_smooth* parameter prevents such changes from happing too often.

The smoothing interval between repartition epochs is larger than that for space-sharing, since repartitioning in this case implies a reconstruction of (portions of) the gang-scheduling matrix. This plus a number of other optimizations [11] can reduce substantially the undesirable cost of repartitioning. During each smoothing interval (in which K is fixed), the system maintains a high-level of performance by adjusting the values of D_k in response to changing load conditions (subject to memory constraints and partition utilization). The various parameters of the gang scheduling partition are shown in Table 3.

Parameter	Policy	Description
D_K	sys-monitored	Multi-programming level
K	sys-monitored	Number of partitions
T_i	sys-admin	Time slice duration
SS_smooth	sys-admin	min. interval to invoke DPS
PLS_smooth	sys-admin	min. interval to invoke DPS
T_{RM}	sys-admin	time for resource info update
K_{min}	sys-admin	min. size of partition

Table 3: Summary of Gang Scheduling Parameters.

Load-Sharing Paradigm *Octopus* adopts a new practical scheduling strategy for the load-sharing paradigm that is based on previous theoretical scheduling results [17]. It outperforms alternative approaches for a class of loosely-synchronized applications.

Upon the arrival of a job J the scheduling policy first determines the best set of nodes on which to schedule the tasks of the job. Let W_n^K denote the amount of time required to complete the collection of J's tasks assigned to node n given that the set of nodes allocated to J are the K least-loaded nodes. The response time for J under this allocation is then given by $M^K \equiv \max\{W_1^K, W_2^K, \ldots, W_K^K\}$, i.e., the maximum time to complete the tasks of this job on all K nodes. The optimal set of nodes is obtained by determining the value of K that minimizes this quantity M^K, which we denote by K^*. The algorithm schedules the tasks of J in a manner that tends to equalize the work assigned to the K^* least-loaded nodes. Naturally, sequential applications are placed on the least lightly loaded node.

Note that the first phase of the algorithm consists of determining the number of nodes that minimizes the expected time to execute the job under the load balancing method used in the second phase. In our implementation, the monitoring facilities provide estimates of the mean (R^{LS}) and variance (V^{LS}) of the workload at each of the nodes. The load-sharing policy uses these estimates to approximate the optimal algorithm and efficiently determine the allocation that tends to minimize the job's response time given the current system state. L^{LS} is a system administrator controlled parameter which specifies the possible range of values for K and at most it is equal to the number of nodes in the partition. These policy parameters are summarized in Table 4.

Parameter	Control Policy	Definition
L^{LS}	sys-admin	max number of nodes for a job
R_j^{LS}	sys-monitored	mean execution time at node j
V_j^{LS}	sys-monitored	execution time variance at node j

Table 4: Load Sharing Parameters

Integrating Gang and Load-sharing Our resource management system integrates the above scheduling paradigms by dividing the nodes into disjoint sets, one for each paradigm supported. For example, in our implementation this consists of the gang-scheduling partition for the tightly-synchronized applications and the load-sharing partition for the loosely-synchronized applications. Upon submission of each job, the user specifies the type of job being submitted and the resource management system routes the job to the appropriate paradigm. The DPS mechanism of the DLS manages the resources of these two partitions in a fashion similar to the DPS of the gang-scheduling partition described earlier.

Table 5 lists the various parameters that can be adjusted to control the behavior of the DLS. The system administrator has full control over the minimum partition size (S_{min}) and the maximum number of jobs (J_{max}) that can be accommodated and may also configure the load differential T and the smoothing interval *DLS_smoothing* (which is coarser than the interval used within the gang-schduling paradigm).

4 Prototype Implementation

Hardware The IBM SP2 is a distributed-memory multicomputer that is connected by a high-speed switch. We use TCP/IP to communicate over the switch and facilitate software portability. The nodes

Parameter	Policy Type	Definition
U_{GS}	sys-monitored	utilization of GS partition
U_{LS}	sys-monitored	utilization of LS partition
Δ	$\mid U_{GS} - U_{LS} \mid$	load differential
T	sys-admin	load differential threshold
DLS_smooth	sys-admin	DPS invocation interval
J_{max}	sys-admin	max number of jobs
S_{min}	sys-admin	min size of a partition

Table 5: The scheduling parameters for the DLS

run 63 MHz RS1 machines. We use an SP2 with 16 nodes, and 32 megabytes of memory.

System Software We have prototyped *Octopus* on AIX 4.1. Currently, the DLS and the PLSs are integrated in one process. The NLS runs as a separate process. The Launcher and the applications run as separate processes. Typically the DLS/PLS and the NLSs run on different nodes.

The Applications We used the following set of applications in our study. *AtEarth* simulates the flight of neutrinos from the sun toward earth. *Adaptive quadrature* is an algorithm for numerical integration. *Matrix Multiplication* performs matrix multiplication in parallel. *Fast Fourier Transform* is a parallel version of fast fourier transform. *Parallel Make* is a parallel version of the DQS make program. *Conjugent Gradient* performs conjugate gradient algorithm in parallel form. *Sim* is a synthetic application which consists of small parallel applications with irregular finishing times for its constituent processes.

Workloads We studied a large number of workloads with different service times by varying the mixture of the above applications and by changing the sizes of their datasets. The results presented later in this paper are a representative sample of the trends observed in our numerous experiments. They were obtained with three different workloads. W1, is a mixture of the *Fast Fourier Transform*, *Matrix Multiplication* and *Conjugent Gradient* applications. It is an efficient and variable workload with an average speedup of 6.2 when executed on 8 nodes (comparing to the execution time on a single dedicated node) and service time coefficient of variation of 4.5. W2 is a less efficient as well as less variable workload with a speedup of 4 when executed on 8 nodes and a coefficient of variation of 1.3. It consists of a mixture of the *AtEarth*, *Adaptive quadrature* and *Conjugent Gradient* applications. Both W1 and W2 respresented characteristics of tightly-synchronous applications. W3 consists of a mixture of loosely-synchronous applications (*Sim*, *Parallel Make* and sequential applications). WN, finally, is an equal mixture of the above three workloads.

5 Performance Measurements

Space Sharing To quantitatively evaluate the benefits of our dynamic partitioning with smoothing policy (see Section 3), we compared the instance used within the space-sharing PLS against two other previously proposed strategies for dynamic space sharing, the greedy dynamic partitioning policy (the smoothing interval was set to 0), and the basic folding policy without rotation [13]. In these experiments, we measured the mean response times of the jobs of workload W2, which performs best under space-sharing due to its inefficiencies (see Section 4), at different utilization levels and we reported the normalized response times in Figure 3.

The main reason greedy dynamic partitioning gives better results than folding (for the workloads considered) is that the former achieves better node allocations. Dynamic partitioning with smoothing outperforms the other strategies in all cases and especially at medium to high utilizations. At these utilization levels, the system is still below full capacity (system queue is empty) and every job arrival and departure is likely to cause delay to both system and applications. By exploiting the concept of smoothing, the system gives time to applications to exit without being disturbed too often. At the same time, it batches the new applications that arrive and performs all reconfigurations in one step.

Gang Scheduling We have performed several experiments associated with the gang scheduling partition, including measuring overheads (context switching, clock drifting between the node clocks, matrix downloading) and parameter tuning. Detailed results can be found in [11]. Here, we examined how the different values of K and D_k (degree of multiprogramming for partition k) can affect performance with respect to system load. For this, we measured the mean response times for jobs with high efficiency and highly variable services times (workload $W1$) as a function of K and D_k for different CPU-utilizations on an 8-node system and we presented the normalized response times in Figure 4.

The overall behavior depends upon factors such as queuing time (when the degree of multi-programming

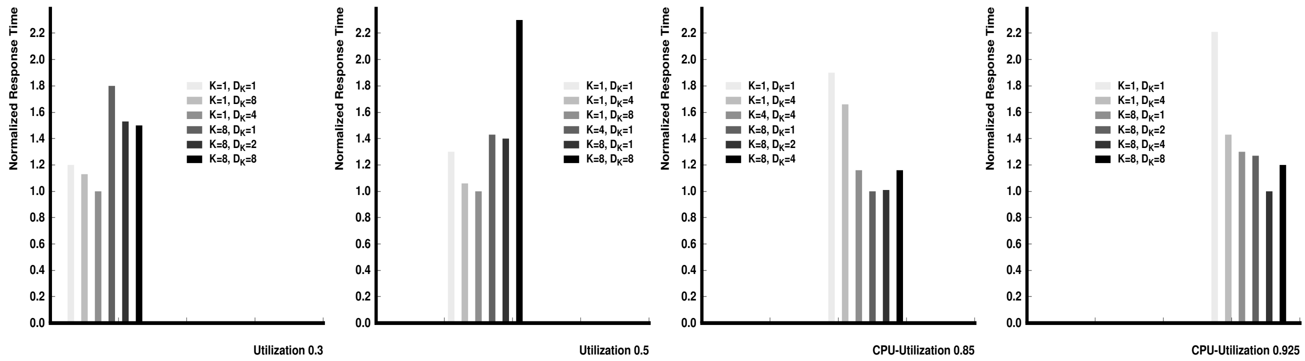

Figure 4: Best values of K and D_k for 0.3, 0.5, 0.85 and 0.925 CPU-utilizations. The results are for 8 nodes and workload W1.

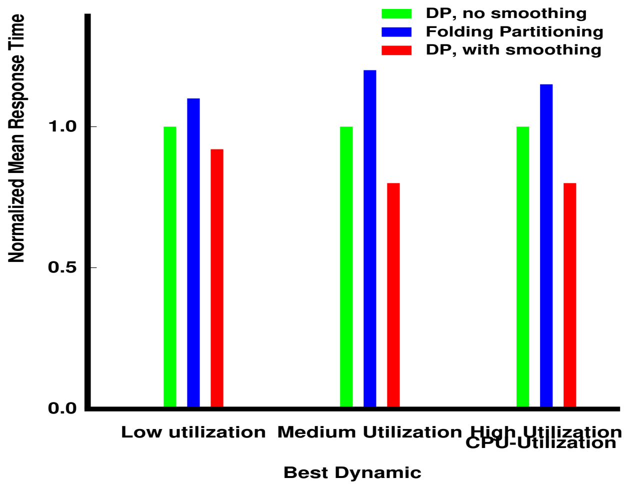

Figure 3: Normalized Mean Response Times for Workload W2, 8 nodes and SS_smooth of 30 sec.

reaches the maximum possible value D_k), resource fragmentation (when there are less jobs than partitions), memory demands and context switch overheads. The best values of K and D_k for this setting are summarized in table 6.

In general, efficient applications can effectively use all the given processors. Thus, when CPU-utilization is low, large values of K can result in fragmentation and idle resources. Thus small values of K are better. By increasing the degree of multi-programming (large D_k), the system can reduce the waiting times and also accommodate any bursts of arrivals. At the other end, at high CPU-utilizations, it is more important to maintain efficient node allocations (larger values of K). In this case, parallel applications run on fewer nodes with higher memory demands per node.

CPU-Utilization	K	D_k
0.0 < U< 0.4	1	4
0.4 < U< 0.7	1	8
0.70 < U< 0.9	8	1
0.9< U< 1.0	8	4

Table 6: The best values of K and D_k for various utilizations and workload W1

Maintaining a low D_k can reduce both the thrashing due to memory misses and context switch overheads. The task of the monitoring facilities is to watch for any changes in the workload and for the gang scheduling PLS to adjust K and D_k accordingly.

Comparing scheduling paradigms Pursuing further the relations among the characteristics of the workloads and the scheduling policies of each of the paradigms, we compared the relative performance for workloads $W1$ and $W2$ against the Space and Time Sharing partitions. Moreover, we measured the mean response time for workloads $W2$ and $W3$ against the Load and Gang Scheduled partitions. Tables 7 and 8 show these results. As expected, the Time Sharing partition is good for highly variable and efficient workloads, due in part to its preferential treatment of short jobs, while Space Sharing is good for less efficient workloads. Similarly, loosely-synchronized, irregular and sequential applications are better suited for the Load Sharing than Gang Scheduling partition due in part to the "holes" the irregural applications leave in the gang-scheduling matrix, while workloads such as $W2$ perform better on a partition that supports synchronized scheduling. In summary, different scheduling policies are geared towards different workloads and no one partition is best for all workloads.

Workload	Space Shared	Time Shared
W1	1.55	1.0
W2	1.0	1.42

Table 7: Comparing Time sharing to Space Sharing for an 8 node system

Workload	Load Shared	Gang Scheduled
W3	1.0	1.28
W2	1.17	1.0

Table 8: The relative performance of Load Sharing vs Gang Scheduling for two different workloads.

Combining approaches, correctly mapping and dynamically managing partitions and resources, provides better performance results that any single approach. Table 9 shows the superiority of such an approach on a particular workload. WN is a workload that consists of an equal percentage of the three workloads (W1, W2, W3). For these experiments we used 16 nodes, *DLS_smooth* was set to twice the *PLS_smooth* parameter and T was 0.2 (i.e., re-allocate nodes when load differential between two partitions is 20 percent of each other).

Workload	Combined	Gang Scheduled	Load Shared
WN	1.0	1.37	1.54

Table 9: Combined approach versus Gang Scheduled versus Load Shared

6 Conclusions

In this paper we have presented the design and prototype implementation of *Octopus*, a hierarchical and extensible resource management system that allows the concurrent execution of multiple scheduling paradigms concurrently. In the implementation of the hierarchical scheduler we have adopted a mechanism called dynamic partitioning with smoothing, to manage the different paradigms simultaneously, and adjust the number of resources allocated to each paradigm to match the needs of the executing workloads. To demonstrate the effectiveness of the scheduler, we have presented a new parallel scheduling algorithm that combines three known scheduling paradigms (load sharing, time sharing and space sharing) and we have measured the performance of the system under workloads with different characteristics.

References

[1] R. Campbell, N. Islam, D. Raila, and P. Madany. Designing and Implementing Choices:an Object-Oriented System in C++. *CACM*, Sept. 1993.

[2] S.-H. Chiang, R. K. Mansharamani, and M. K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proc. ACM SIGMETRICS Conf. Meas. Mod. Comp. Sys.*, pp 33–44, May 1994.

[3] K. Dussa, B. Carlson, L. Dowdy, and K-H. Park. Dynamic partitioning in transputer environments. In *Proc. ACM SIGMETRICS Conf. Meas. Mod. Comp. Sys.*, pp 203–213, 1990.

[4] D. G. Feitelson and L. Rudolph. Distributed hierarchical control for parallel processing. *Computer*, pp 65–77, May 1990.

[5] B. Ford and S. Susarla. CPU Inheritance Scheduling. *OSDI 96*, Oct. 1996.

[6] P. Goyal, X. Gao, and H. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. *OSDI 96*, Oct. 1996.

[7] N. Islam. *Distributed Objects: Methodologies for Customizing System Software.* IEEE Computer 1996.

[8] N. Islam. Customizing System Software Using OO Frameworks. In *IEEE Computer*, Feb. 1997.

[9] N. Islam and M. Devarakonda. An Essential Design Pattern for Fault-Tolerant Distributed State Sharing. In *CACM*, Oct. 1996.

[10] N. Islam, A. L. 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.), pp 244–270. Springer-Verlag, 1996. Lecture Notes in Comp. Sci. Vol. 1162.

[11] N. Islam, A. L. 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.

[12] S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proc. ACM SIGMETRICS Conf. Meas. Mod. Comp. Sys.*, pp 226–236, May 1990.

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

[14] V. K. Naik, S. K. Setia, and M. S. Squillante. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Proc. Supercomputing*, pp 824–833, Nov. 1993.

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

[16] 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.), pp 219–238. Springer-Verlag, 1995. Lecture Notes Comp. Sci. Vol. 949.

[17] 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.