
138
053
THS

THESIS
1
2004

This is to certify that the
thesis entitled

Parallel Network RAM: Effectively Utilizing Global Cluster Memory For Large Data-Intensive Programs

presented by

Jonathan James Oleszkiewicz

has been accepted towards fulfillment
of the requirements for the

degree of M.S. in Computer Science and Engineering

Major Professor's Signature

5/3/2004

Date

MSU is an Affirmative Action/Equal Opportunity Institution

LIBRARY
Michigan State
University

PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.
MAY BE RECALLED with earlier due date if requested.

DATE DUE	DATE DUE	DATE DUE

6/01 c:/CIRC/DateDue.p65-p.15

PARALLEL NETWORK RAM
EFFECTIVELY UTILIZING GLOBAL CLUSTER MEMORY
FOR LARGE DATA-INTENSIVE PROGRAMS

By

Jonathan James Oleszkiewicz

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science

2004

ABSTRACT

PARALLEL NETWORK RAM
EFFECTIVELY UTILIZING GLOBAL CLUSTER MEMORY
FOR LARGE DATA-INTENSIVE PROGRAMS

By

Jonathan James Oleszkiewicz

Large scientific parallel applications demand large amounts of memory space. Current parallel computing platforms schedule jobs without fully knowing their memory requirements. This leads to uneven memory allocation in which some nodes are overloaded. This, in turn, leads to disk paging, which is extremely expensive in the context of scientific parallel computing. To solve this problem, we propose a new peer-to-peer solution called "Parallel Network RAM". This approach avoids the use of disk, better utilizes available RAM resources, and will allow larger problems to be solved while reducing the computational, communication and synchronization overhead typically involved in parallel applications.

To Patty, my constant source of inspiration and the love of my life.

Acknowledgments

This research was made possible by grants from the Michigan Space Grant Consortium and the National Science Foundation.

Table of Contents

LIST OF TABLES vii

LIST OF FIGURES viii

1 Introduction 1
- 1.1 Problem Statement . . . 1
- 1.2 Terminology . . . 5
- 1.3 Background and Related Work . . . 6
  - 1.3.1 Parallel Scheduling Algorithms . . . 6
  - 1.3.2 Previous Solutions to the Memory Problem . . . 8
  - 1.3.3 Network RAM . . . 9

2 Parallel Network RAM 10
- 2.1 Introduction . . . 10
- 2.2 Generic Description . . . 11
  - 2.2.1 Clients . . . 11
  - 2.2.2 Managers . . . 12
  - 2.2.3 Servers . . . 15
- 2.3 Designs . . . 15
  - 2.3.1 Client . . . 16
  - 2.3.2 Centralized . . . 17
  - 2.3.3 Local Managers . . . 17
  - 2.3.4 Backbone . . . 19

3 Methodology 20
- 3.1 Simulator . . . 20
  - 3.1.1 Workload Model . . . 20
  - 3.1.2 System Models . . . 21
  - 3.1.3 Job Behavior Models . . . 22
  - 3.1.4 Scheduler Models . . . 26
  - 3.1.5 Metrics . . . 26
- 3.2 Experimental Setup . . . 27
  - 3.2.1 Experiment Table Detail . . . 28
  - 3.2.2 Dimensions of the Experiment Table . . . 29

4 Performance Evaluation 31
- 4.1 64 Node Cluster . . . 31
  - 4.1.1 Base Experiments . . . 31
  - 4.1.2 RAM Variations . . . 35
  - 4.1.3 Network Performance Variations . . . 37
  - 4.1.4 Network Topology Variations . . . 40
  - 4.1.5 Space Sharing Scheduler Experiments . . . 42
- 4.2 128 Node Cluster . . . 42
  - 4.2.1 Base Experiments . . . 46

4.2.2 RAM Variations 49
4.2.3 Network Performance Variations 49
4.2.4 Network Topology Variations 51
4.2.5 Space Sharing Scheduler Experiments 54

5 Analysis **56**
5.1 Discussion 56
5.1.1 Memory Load 56
5.1.2 Network 57
5.1.3 Scheduling 58
5.1.4 PNR Designs 59
5.2 Conclusion 60
5.3 Future Work 61

LIST OF REFERENCES **63**

List of Tables

3.1 Table of Experiments. 28

5.1 Recommended PNR Design Usage. 60

List of Figures

2.1 Diagram of Parallel Network RAM. Application 2 is assigned to PEs P3, P4 and P5 but utilizes available memory space of P2, P6 and P7. 10
2.2 The architecture of a servent. Each servent can act in three roles. . . 12
2.3 Client-only design. 16
2.4 Centralized design. 16
2.5 Local manager design. 18
2.6 Backbone design. 18

4.1 Base experiment - 64 PE - several workloads - response time. 32
4.2 Base experiment - 64 PE - 4,000 jobs - response time. 32
4.3 Base experiment - 64 PE - 4,000 jobs - optimization ratio. 32
4.4 Base experiment - 64 PE - 5,000 jobs - response time. 33
4.5 Base experiment - 64 PE - 5,000 jobs - optimization ratio. 33
4.6 RAM experiments - 64 PE - 4,000 jobs - response time. 36
4.7 RAM experiments - 64 PE - 4,000 jobs - optimization ratio. 36
4.8 RAM experiments - 64 PE - 5,000 jobs - response time. 36
4.9 RAM experiments - 64 PE - 5,000 jobs - optimization ratio. 38
4.10 Network experiments - 64 PE - 4,000 jobs - response time. 38
4.11 Network experiments - 64 PE - 4,000 jobs - optimization ratio. 38
4.12 Network experiments - 64 PE - 5,000 jobs - response time. 39
4.13 Network experiments - 64 PE - 5,000 jobs - optimization ratio. 39
4.14 Topology experiments - 64 PE - 4,000 jobs - response time. 41
4.15 Topology experiments - 64 PE - 4,000 jobs - optimization rate. 41
4.16 Topology experiments - 64 PE - 5,000 jobs - response time. 41
4.17 Topology experiments - 64 PE - 5,000 jobs - optimization ratio. . . . 43
4.18 Space sharing experiments - 64 PE - 4,000 jobs - response time. . . . 43
4.19 Space sharing experiments - 64 PE - 4,000 jobs - optimization ratio. . 43
4.20 Space sharing experiments - 64 PE - 5,000 jobs - response time. . . . 44
4.21 Space sharing experiments - 64 PE - 5,000 jobs - optimization ratio. . 44
4.22 Base experiment - 128 PE - several workloads - response time. 44
4.23 Base experiment - 128 PE - 4,000 jobs - response time. 45
4.24 Base experiment - 128 PE - 4,000 jobs - optimization ratio. 45
4.25 Base experiment - 128 PE - 5,000 jobs - response time. 45
4.26 Base experiment - 128 PE - 5,000 jobs - optimization ratio. 47
4.27 RAM experiments - 128 PE - 4,000 jobs - response time. 47
4.28 RAM experiments - 128 PE - 4,000 jobs - optimization ratio. 47
4.29 RAM experiments - 128 PE - 5,000 jobs - response time. 48
4.30 RAM experiments - 128 PE - 5,000 jobs - optimization rate. 48
4.31 Network experiments - 128 PE - 4,000 jobs - response time. 50
4.32 Network experiments - 128 PE - 4,000 jobs - optimization ratio. . . . 50
4.33 Network experiments - 128 PE - 5,000 jobs - response time. 50
4.34 Network experiments - 128 PE - 5,000 jobs - optimization ratio. . . . 52
4.35 Topology experiments - 128 PE - 4,000 jobs - response time. 52

4.36 Topology experiments - 128 PE - 4,000 jobs - optimization ratio. . . . 52
4.37 Topology experiments - 128 PE - 5,000 jobs - response time. 53
4.38 Topology experiments - 128 PE - 5,000 jobs - optimization ratio. . . . 53
4.39 Space sharing experiments - 128 PE - 4,000 jobs - response time. . . . 54
4.40 Space sharing experiments - 128 PE - 4,000 jobs - optimization ratio. 54
4.41 Space sharing experiments - 128 PE - 5,000 jobs - response time. . . . 55
4.42 Space sharing experiments - 128 PE - 5,000 jobs - optimization ratio. 55

Chapter 1

Introduction

1.1 Problem Statement

Many scientific computing applications demand a large amount of processing power, memory space and I/O accesses. Cluster systems with networked server nodes are becoming more popular for executing high-performance scientific computing applications for both economic and technical reasons [16]. One standard approach to reducing the runtime of such applications is to parallelize them into multiple parallel processes so many cluster nodes can run parts of the application simultaneously. An advantage of this approach is the CPU and memory resources of the application are evenly distributed and used. However, this advantage may not serve the best performance interests of parallel processes because a balanced workload distribution among parallel processes may result in unbalanced resource utilization in a cluster. Uniform use of resources at the parallel process level does not necessarily mean the system itself is evenly utilized.

We can attempt to adjust memory load by adjusting the number of processes each parallel process has, but there are trade-offs between memory usage and efficiency. In order to ensure each node has enough memory space to accommodate processes, we

could partition parallel processes into a large number of processes. This results in less work and more required synchronization for each process. As a consequence, the CPU on each node may be underutilized. Parallel speedup is very hard to improve as the number of processes increases, due to increasing communication and synchronization overhead. This problem is sometimes referred to as the performance exponential degradation issue.

To ensure good CPU utilization we must limit the number of processes in a parallel process. But, as the problem size increases, nodes may run out of available memory and be forced to use the local disk as a swapping site [3, 4]. Performance will suffer from frequent page faults since hard disks are orders of magnitude slower than RAM. Research has shown that disk paging results in unsatisfactory performance on parallel platforms and should be avoided [3, 4, 26].

We can increase the number of cluster nodes or the amount of RAM at each node, but doing so may be impossible or very expensive given the cluster setup. In addition, it is unlikely that the additional nodes will offer any real benefit over the long-term, since it is likely the users of the cluster will simply increase their usage of the system to match the new resources available. This will lead us back to the original problem.

The key problem is memory usage, and this problem has two parts: memory fragmentation and paging overhead. The aggregate memory capacity of the system may be enough to satisfy memory demands, but cluster memory is distributed into small chunks. Usage of these chunks may be uneven and inefficient. On the most heavily loaded nodes, disk paging is invoked which incurs a high cost.

Network RAM [1, 18] has been proposed for use by sequential jobs in clusters to even memory load and reduce paging overhead. This technique allows applications to allocate more memory than is available on the local machine while avoiding paging to disk by allocating idle memory of other machines over a fast interconnecting network.

This remote RAM is treated as a new layer in the memory hierarchy between RAM and disk. Resulting page accesses are slower than RAM, but faster than disk [1, 9, 18, 23, 39].

Existing network RAM techniques should not be directly applied to parallel jobs for performance gains. One issue is that processes from the same parallel job synchronize regularly. If each node hosting these processes seeks network RAM independently, they may be granted an uneven amount of network RAM. With this uneven allocation, the processes executing on these nodes will run at different speeds. However, the parallel job as a whole will only run at the speed of the slowest process, due to synchronization. The nodes with extra network RAM waste it, since their hosted processes will spend most of their time waiting for other processes. Therefore, coordination is required to grant overloaded nodes equal portions of memory to allow hosted processes to run at equal speeds.

Another issue is network congestion. If parallel processes individually seek out network RAM with no coordination among themselves, a potentially large amount of unnecessary network traffic will result. This may induce congestion on the cluster. Parallel applications require high performance networks to run efficiently. Congestion could seriously impact the performance of jobs on the system.

We propose a new peer-to-peer solution, called Parallel Network RAM (PNR), so overloaded cluster nodes can utilize idle remote memory. In this scheme, each node may request memory resources from remote nodes and provide memory resources for others. Requests are indirect: each node contacts a manager (super-peer) node and requests that it allocate network RAM on its behalf. Managers coordinate the allocation of network RAM of several nodes and ensure that memory resources are distributed evenly to the nodes hosting parallel processes belonging to the same parallel job. PNR will allow more jobs to execute concurrently without resorting to disk paging. This will lead to decreased average response times and higher system

throughput.

This thesis makes several contributions.

- We first identify the unbalanced resource utilization problem in a cluster with a mixed workload of jobs with different resource requirements. Existing techniques cannot maximize the performance gain of parallel jobs in such an environment in terms of both parallel speedup and execution time.

- We propose a novel and effective solution to this problem called Parallel Network RAM (PNR). PNR makes it possible for parallel jobs in a cluster to utilize memory resources from available remote nodes. The CPU cycles will be provided by a small subset of nodes while the global memory space of the cluster is open to the memory demands of any parallel job. Since the speed gap between accessing local memory and remote memory is shrinking, and the speed gap between accessing local disk and remote memory continues to enlarge, the proposed scheme is expected to be beneficial for large scale scientific computing applications now and in the future.

- We build a simulator that models a cluster and several proposed PNR algorithms. Conducting trace-driven simulations, we compare the performance of six different PNR designs to each other and to a disk paging-only solution. We then identify which proposed algorithms are superior and under what conditions.

The rest of this thesis is organized as follows. Chapter 1 describes the problem presented in cluster systems and reviews related work on this issue. Chapter 2 describes Parallel Network RAM, the algorithms used to implement PNR, different proposed PNR designs and their strengths and weaknesses. Chapter 3 describes the simulator we have created to test our PNR algorithms and it describes the specific

experiments set up for these tests. Chapter 4 describes the results of the experiments introduced in chapter 3. Chapter 5 discusses the results gathered and draws conclusions about the various designs.

1.2 Terminology

The systems described in this thesis include computing clusters and supercomputers. We describe individual computers in the clusters and CPUs in the supercomputers with the terms "PE" (Processing Element) and "node". The terms are equivalent and will be used interchangeably.

The terms "parallel process" and "job" are also equivalent and are used to describe programs that have multiple threads of execution that run on separate nodes. "Thread" and "process" denote the individual threads of execution of parallel processes.

We define four different Parallel Network RAM designs. We compare these designs to a system that uses only disk paging (no network RAM). For brevity, we will use acronyms to identify these designs. The following list defines each acronym.

DP - Disk Paging - applies to systems that do not use Parallel Network RAM.

PNR - Parallel Network RAM - a generic label for all Parallel Network RAM designs.

CEN - Centralized PNR - only one manager exists.

CLI - Clients only PNR - a pure peer-to-peer design.

MAN - Local Managers PNR - a design that uses randomly selected local managers.

BBX - Backbone PNR - a design where a certain number of servents (represented by X) act as "backbone" managers.

1.3 Background and Related Work

The majority of work in this area has focused on parallel job scheduling. In this section, we describe various parallel job schedulers and previous solutions to the problem of overloaded memory on cluster systems.

1.3.1 Parallel Scheduling Algorithms

The primary duty of the scheduler on a cluster system is to ensure high system throughput and low overall response times of submitted jobs. The most simplistic scheduling model is the dedicated machine model. In this model, only one job runs on the system at a time. Each job is scheduled using a priority queue which may be sorted in a variety of ways, such as first-come first-serve (FCFS), estimated shortest job first, best fit, or worst fit. Each job runs until completion and may not be preempted. This model lacks time sharing and space sharing and since most parallel jobs do not use all of the available nodes in the cluster, this scheme can waste a large amount of resources.

An approach that makes better use of available resources is the space sharing model. Space sharing allows more than one job to be scheduled on the cluster at one time. Each node is devoted to one process, and each job runs until completion without preemption. The space sharing model is vulnerable to large, long-running jobs monopolizing the system and a bad scheduling decision is difficult to correct. The dedicated machine model shares this problem [16].

Gang scheduling combines both space sharing and time sharing to avoid the problems associated with large jobs. Each job is alloted a time slot and the job may execute in its time slot. Nodes within each time slot are space shared. When a time quantum has expired, all jobs in the current slot are preempted and replaced with jobs in the next slot. The preemption process is called a "Parallel Context Switch" (PCS)

and involves a certain amount of overhead. There is at most one process running on each node at any given time, although some schemes relax this constraint [36]. The time slot mechanism ensures no large job may monopolize the system for a long period of time.

The maximum number of time slots allowed is known as the Multi-Programming Level (MPL). Setting the MPL to a low number is a convenient way to reduce PCS overhead and reduce overall memory load on nodes. A system with an MPL value of one is a simple space sharing system with no time sharing.

The length (in time) of time slots varies. It is intuitive to make each time slice the same length. Fixed-length time slots range from 1 second [4] to 120 seconds [29]. One disadvantage of uniform-length time slices is that excessive idleness can be caused by short jobs [10]. Some research has shown that variable length time quantums may be desirable to increase efficiency [14, 34].

There are a variety of processor allocation strategies and variations of gang scheduling. These include first fit, best fit, left-right by size, left-right by slots, load-based allocation strategies, "buddy" systems such as distributed hierarchical control, and migration-based algorithms [10, 41].

One problem with gang scheduling is that the usage of nodes within time slots may become fragmented as jobs terminate and new jobs take their place. Two techniques can reduce node fragmentation: alternative scheduling and slot unification. Alternative scheduling discovers time slots that have idle nodes and finds jobs that run on these same nodes in other time slots. These jobs are allowed to run in both their original time slot and the partially idle slot. This effectively doubles the amount of time the selected jobs are allowed to run, increasing system utilization and reducing average response time.

Slot unification attempts to unify time slots which use disjoint nodes. Typically, such situations occur after job termination [10]. This technique reduces the total

number of time slots used by the scheduler and increases the speed of the system as perceived by the parallel processes.

Another available technique is backfilling. If users provide estimated job run times, the scheduler can identify idle areas in the schedule which can be filled with smaller, lower priority jobs. The smaller jobs must not interfere with the scheduled running time of jobs ahead in the queue. If a backfilled job runs for too long, it is killed. Backfilling has been shown to increase utilization and reduce response time. Several variations of backfilling have been studied [24, 28, 32, 33, 35, 40].

The MAUI scheduler, a particularly popular and mature scheduler for cluster systems is a space-sharing, backfilling scheduler [20]. Research work has been done to implement gang schedulers on both supercomputing platforms [7] and clusters of workstations [19].

1.3.2 Previous Solutions to the Memory Problem

Previous studies agree that unmodified disk paging results in severely reduced performance on parallel systems [4, 30]. Generally speaking, previous solutions to this problem either attempt to avoid disk paging entirely or attempt to reduce its effect.

Various ways to avoid paging by altering the system scheduler have been suggested. If no memory information about incoming jobs is known, then the simplest solution is to keep MPL to a minimum [24]. Another solution attempts to guess memory usage information based on information about the job provided by the user and information contained in the program executable. This guess is used in scheduling decisions [3]. Another solution uses speedup information known about jobs ahead of time to make scheduling decisions [26, 27]. Given this information, the scheduler can choose to give more processors to efficient jobs to increase utilization or to memory-intensive jobs to increase throughput.

User-provided runtime and memory information can be used in scheduling decisions as well. In fact, the use of user estimates of runtimes is the basis of backfilling. However, it is well known that such information is unreliable, as users will tend to estimate numbers that will ensure they get a good position in the scheduler queue. Some backfilling schemes attempt to take advantage of systemic inaccurate runtime estimations [28].

One method that is aimed at reducing the disk paging penalty is called block paging. In this scheme, the system groups sets of pages together and acts upon these groups as units. Groups are defined by the system based on memory reference behavior of jobs [34].

1.3.3 Network RAM

Network RAM is a technique that reduces paging overhead. Much work has been done for sequential job scheduling with memory considerations. Regarding network RAM implementations, the Global Memory System (GMS) [9] and the Remote Memory Pager [23] attempt to reduce page fault overhead by using remote paging techniques. DoDo [1] is designed to improve system throughput by harvesting idle memory space in a distributed system. In DoDo, processes running on the local system have the highest priority for using CPUs and memory on their workstations. This divides the global memory system into different local regions.

A memory ushering algorithm is used in MOSIX for memory load sharing [2]. This solution is a job-migration-based load sharing approach. Recently, several load sharing alternatives have been developed. These techniques consider both CPU and memory resources with known and unknown memory demands [37, 38]. The objective of the designs is to reduce the number of page faults caused by unbalanced memory allocations of distributed jobs so that overall performance can be significantly improved.

Chapter 2

Parallel Network RAM

2.1 Introduction

We propose a novel and effective technique called Parallel Network RAM (PNR) to better utilize both CPU and memory and minimize communication and synchronization overhead. We demonstrate the basic idea of PNR in Figure 2.1. In this figure, application 2 runs on nodes P3, P4 and P5 but utilizes available memory space in nodes P2, P6 and P7 where non-memory intensive applications 1 and 3 reside. With PNR, instead of three nodes being overloaded while four are underloaded, all seven nodes are fully utilized.

Our objective is to fundamentally improve the efficiency of large-scale scientific

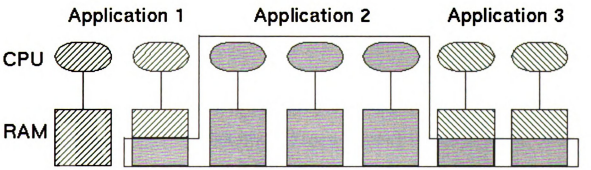

Figure 2.1: Diagram of Parallel Network RAM. Application 2 is assigned to PEs P3, P4 and P5 but utilizes available memory space of P2, P6 and P7.

computing. Under our proposed solution, CPU and memory resource allocations are considered separately. CPU utilization requirements can be taken into account during the scheduling phase, and memory requirements can be handled as needed during execution. In this way, CPU usage can be optimized to maximize utilization and minimize communication and synchronization overhead while memory usage can include both the local memory space from the assigned CPUs and the remote memory space in other nodes (as needed). With PNR, speedup can be scaled as the available remote memory increases, and performance can also be scaled as the problem size increases.

Because CPU usage and memory usage are considered separately, PNR does not coordinate with or receive information from the assumed centralized scheduler of the system. This allows PNR to be implemented on a variety of platforms without changes to their existing scheduling policies.

2.2 Generic Description

In brief, all nodes in the system host PNR servents (see figure 2.2). All servents act as PNR clients and servers. Some servents may act as managers. Managers act as proxies for clients to communicate with servers. The purpose of managers is to coordinate client requests.

2.2.1 Clients

A PNR client attempts to allocate and deallocate network RAM on the behalf of its hosting node. The node uses allocated network RAM as additional virtual memory just as it would with disk space. When a process starts execution on a node, it allocates the amount of memory it will use during its execution. If the node's client determines that this allocation will lead to disk usage, the client contacts a manager

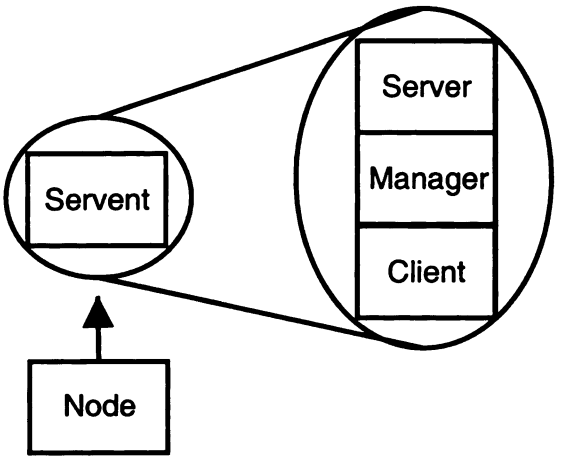

Figure 2.2: The architecture of a servent. Each servent can act in three roles.

and requests network RAM. Once network RAM is allocated, the client is informed by the manager what machines are serving the network RAM, and how much was allocated. The client may then start sending pages to the servers for storage and later retrieval.

Similarly, when a process stops execution it will deallocate its memory. If the client detects that network RAM previously allocated is no longer needed, then the client signals a manager that it may be deallocated. Clients deallocate chunks of memory first from servers believed to have the highest memory load.

2.2.2 Managers

PNR managers are the centerpiece of most of the proposed PNR algorithms. They do the majority of allocation and deallocation work and act as proxies between clients and servers.

Network RAM Request

Managers listen for network RAM requests from clients (including the client residing on the same node as the manager). Depending on the PNR strategy, the manager may act immediately on this request or wait for other requests to come in

before acting. Most strategies require that all threads within a parallel job must contact the same manager before the manager is allowed to act on any one of the threads' requests. For this to work, it is assumed that the manager can discover the total number of threads belonging to the parallel job. As each new request comes in, information on the aggregate memory request is stored in a request table.

Server Request and Response

When all requests are received from relevant clients, the manager attempts to select a server from an availability list. Multiple selection schemes are possible. Currently, a randomized worst-fit scheme is used. That is, the server perceived to have the most RAM available for remote allocation (i.e. the server with the most idle RAM) is always selected. If servers tie for the highest amount of RAM available, one server is randomly chosen.

The server contacted may grant, partially grant, or deny the request. It is important to note that a server that was listed as having RAM available for allocation may not have the same amount available by the time the request message reaches it.

The server's response is received by the PNR manager. If too little memory was allocated, the PNR manager will attempt to contact another server. The process is repeated until enough network RAM is granted by multiple servers or until no servers are left to query.

Network RAM Request Response

When either enough network RAM is allocated or when all possible servers are queried, the manager will calculate the total amount of network RAM granted. Based on this, the manager will divide network RAM evenly among the requesting clients. If a client requests less network RAM than it would get in its fair share, exactly as much as it requests is granted and the remainder is reserved for other clients. If a

client requests more network RAM that it would get in its fair share, it gets its share and only receives more if other clients do not need all of their shares. Each client is informed of which servers are hosting the granted network RAM.

The even distribution of network RAM is done to ensure that each process in a parallel job runs at roughly the same speed. If network RAM were granted haphazardly, each process would run at different relative speeds. The job as a whole would run at the speed of the slowest process because of synchronization and the fast processes will be held back.

Network RAM Deallocation Notification

It is assumed that deallocation notifications are only sent at job termination time. The manager will listen for deallocation notifications from clients. The deallocation amounts indicated in the messages may have nothing to do with how much was previously allocated to those clients. In some cases, no network RAM may be deallocated because it is still needed by the client. In most strategies, the manager will wait for each client associated with a parallel job to send a deallocation notification.

Server Notification and Response

When all deallocation notifications are received by the manager, the manager determines how much network RAM needs to be deallocated and on which servers. Each server is then notified that it may deallocate the specified amount of RAM. The server will respond to this message with an acknowledgment.

Broadcasting Availability Information

During each message-passing interaction, clients and servers piggyback current memory load information onto messages sent to managers. The manager receives this information in addition to the relevant payload. It uses this up-to-date memory load

information to update its own network RAM availability table. It uses this table for server selection.

Some PNR strategies may prefer to share this information by broadcasting it to all servents. Other strategies may only broadcast to selected servents. Currently, broadcasts are only executed after network RAM has been allocated or deallocated.

It should be noted that memory load information may already be out-of-date before it is even broadcast. There is no way around this problem, since it is possible for memory loads to change as messages are being passed on the network. We believe this broadcasting solution provides an economical way to keep availability information reasonably up-to-date.

2.2.3 Servers

Servers receive requests from managers for network RAM. If the server has more unallocated RAM than a certain threshold, it will grant the network RAM request and allocate memory to the manager up to that threshold. Currently, the unallocated memory threshold is set to a low value - a tenth of a megabyte. This value can be adjusted and may be useful for future work.

After the memory is allocated, servers receive requests to read and write the allocated network RAM directly from clients. Servers grant all valid deallocation attempts.

2.3 Designs

We propose four different PNR designs. Each design has a slightly different architecture and has different amounts of communication overhead associated with it. This section describes these designs.

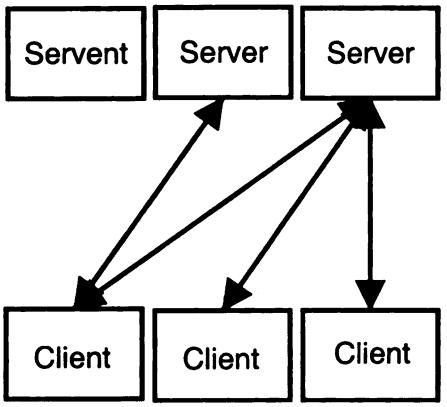

Figure 2.3: Client-only design.

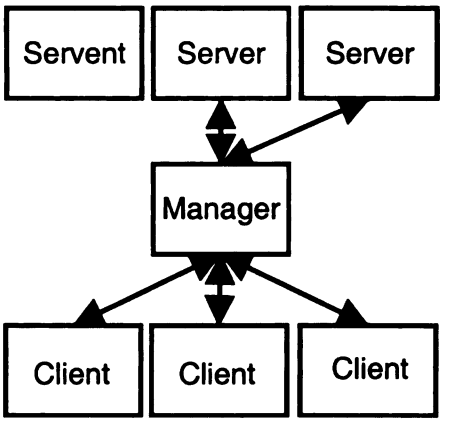

Figure 2.4: Centralized design.

2.3.1 Client

In this strategy (CLI), each client uses its servent's manager to send allocation requests (see figure 2.3). The local manager does not wait for messages from other clients - it acts immediately upon the client's request. The manager attempts to allocate as much network RAM as possible for its client. When the client receives network RAM, it begins execution immediately. It does not wait for the other threads in the parallel job. The local manager does not share memory load information with other servents.

This strategy allows clients to allocate network RAM quickly and eliminates all coordination overhead. It is scalable, since each client is responsible only for itself. It is also simple to implement, since there is no need for a manager at all.

However, this solution has major drawbacks. First, memory load information is not shared. Each client must discover memory load information for itself and this may lead to a large amount of ineffective network RAM allocation requests.

Second, and more serious, the clients do not coordinate memory allocation with each other. Some clients may receive large amounts of network RAM if they get their messages to the servers first while other clients get very little network RAM. This will not improve the performance of parallel jobs as a whole, since each job will only

execute at the speed of its slowest thread.

In fact, this scheme may worsen overall performance, since much of the network RAM allocated is wasted. Thus, no benefit is gained from network RAM and higher memory loads on the servers may induce disk paging, the very problem we are trying to avoid!

2.3.2 Centralized

In this strategy (CEN), only one manager exists, and it receives all client requests (see figure 2.4). All servents know the identity of this manager. All clients will contact this manager and it will coordinate network RAM allocation. The server uses memory load information sent in by clients and servers to make allocation decisions. Since only one active manager exists, the centralized manager does not broadcast any memory load information it receives.

This scheme has the advantage of one agent having all of the information and making all of the decisions. No time or network bandwidth is wasted in sending coordination messages to other managers. The disadvantage of this strategy is that it is not scalable. As the system size grows, the network connections leading to the node hosting the manager will become a bottleneck and limit the performance of the system. In the real world, the computational and memory overhead of hosting this central manager would also become major factors. However, in our simulator, these are not taken into account.

2.3.3 Local Managers

In this strategy (MAN), each client contacts a "local" manager (see figure 2.5). Specifically, whenever a job starts or stops, one of the servents running on a node associated with that job will volunteer to act as the manager in addition to acting as a client. Each servent involved must agree on which servent will act as the manager.

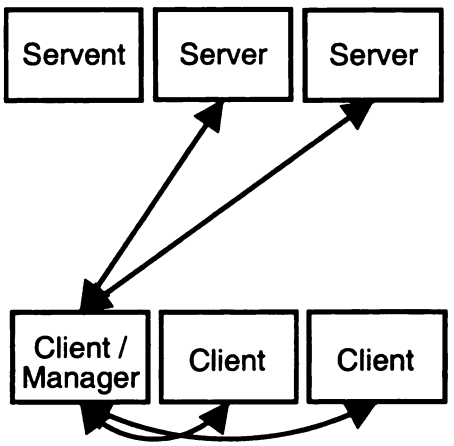

Figure 2.5: Local manager design.

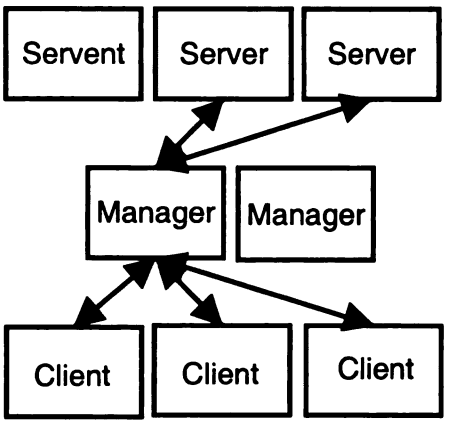

Figure 2.6: Backbone design.

All clients from the involved servents will contact this manager. The manager will take their requests, allocate network RAM (if possible), and divide the received network RAM evenly among the requesting clients. At the end of each allocation and deallocation, the manager will broadcast memory load information to each servent in the system. This is necessary since each servent on the system can potentially act as a manager.

Since no single node is loaded with all requests, this solution is scalable. It makes good use of allocated resources via the coordination of client requests. Broadcasting the memory load information should keep the tables of the servents relatively up-to-date.

However, the major drawback of this approach is the broadcasting step. Sending a message to each servent on the system can introduce congestion into the system, especially if no real broadcasting facility exists and broadcasting must be implemented via multiple point-to-point messages. The larger the system, the bigger this issue will become.

Also, since there is no central authority on memory allocation information, servents may be more likely to act on outdated information. Like the centralized strategy, waiting time is incurred in the coordination step by waiting for all clients to send in

their requests.

2.3.4 Backbone

This strategy is a hybrid of the centralized and local manager strategies. Here, a variable-sized subset of servents will act as managers (see figure 2.6). This subset of servents will be well known and all clients will contact these servents for their network RAM requests. Clients will randomly select a manager for service. As in the MAN design, all clients associated with a job must agree on who to contact.

The backbone of managers will coordinate among themselves by broadcasting memory load information only to managers. If the backbone of managers consists of only one member, then this strategy is equivalent to the centralized strategy. If the backbone of managers consists of all servents, then this strategy is closely equivalent to the local managers strategy.

This scheme can potentially be a "best of both worlds" solution, compared to the centralized and local managers solutions. It is more scalable than the centralized solution, since load is shared among many servents, and it uses fewer messages for coordination than the local managers solution, since broadcast messages only need to be sent to a subset of nodes.

The backbone strategy also has the advantage of being customizable to the cluster setup. If the network is small, then a small backbone (perhaps a backbone of one) may be all that is required. As the network gets larger, then the number of servents in the backbone can be increased appropriately to manage scalability.

Chapter 3

Methodology

3.1 Simulator

To test our proposed PNR designs, we created a simulator that models a parallel platform, parallel jobs, and our PNR algorithms [25]. This simulator is written in C++ and compiles under UNIX systems. We used a modified version of Sim++ as our main simulation library. Sim++ is the C++ version of Simpack, a simulation library created by Paul Fishwick [6, 17]. Sim++ is open-source, and we modified the implementation to suit our needs and maximize performance.

This chapter describes the models we implemented in our simulator and the experiments we ran on the simulator.

3.1.1 Workload Model

Many workload traces and synthetic workload generators exist for use by simulators. However, no standard memory usage benchmarks currently exist [5]. We decided to use a large trace with memory allocation information that has been assembled and discussed by Feitelson [11].

The trace has been gathered from the CM-5 parallel platform at the Los Alamos

National Lab. It contains information about 201,387 jobs run through the majority of 1996. We will use a subset of jobs from this well-studied trace. Since the workload profile at a given site tends to be fairly stable over time [22], using a subset of jobs should be indicative of the general load on the system. However, using a workload trace may be biased toward that collected site's policies and not representative of all workloads [5, 15, 22]. For example, in our trace no process attempts to allocate more memory than is physically available on a node and all jobs allocate PEs in powers of two [13]. Obviously, not every cluster installation will have a workload with these characteristics.

3.1.2 System Models

To complement the CM-5 workload trace, we model a system architecture similar to the CM-5. Each node in the simulated system runs at 33 Mhz and has 32 MB of local memory. The original CM-5 did not support paging [13] and we add this capability to our simulated system. Each node has a disk of infinite capacity that runs at 7200 RPM with a seek time of 9 ms and a transfer rate of 50 MBps. The interconnecting network is assumed to be a simple Ethernet 100 Mbps star topology. Each link has a latency of 50 nanoseconds and the central switch has a processing delay of 80 microseconds. The original CM-5 had 32 processors dedicated solely to system tasks, so it is assumed that the operating system of the CM-5 imposes no CPU or memory load on the nodes.

Network Models

The simulator provides simulated network links. A link connects either one computer to another computer (or set of computers) or a computer to a switch. Only one message may be transmitted on a link at a time. A link has a fixed latency time and a transmission time based on the bandwidth and the size of the message

transmitted. In our experiments, we set the latency of each link to be that of the speed of light through 10 meters of copper wire (50 nanoseconds). No collisions occur on the simulated links. Incoming messages are queued if the link is currently in use.

A simulated network switch stores communications and forwards them to the appropriate link. Switches may only forward one message at a time and all other messages are queued. Switches are assumed to have a single infinite length queue sorted in FIFO order. Switches have a fixed length processing delay that can be set as a parameter. The base value of this delay is set to 80 microseconds.

Point-to-point messages traverse links and switches only. Routes including nodes as intermediate points are not used. Some PNR methods use broadcasts to send information to servents. In our simulator, broadcasts are simulated using N point-to-point messages from the broadcasting node to each other node.

3.1.3 Job Behavior Models

Each job is composed of multiple processes. It is assumed that each process allocates a static amount of memory at start time. Each process accesses memory at its node independently. Previous studies have shown that parallel scientific applications generate memory references every three to five CPU cycles and have a cache hit ratio that ranges from approximately 50% to 65% [8, 31]. We assume that our processes access memory every four CPU cycles and have a cache hit ratio of 50%.

Synchronization Model

All processes in a job synchronize with each other at regular intervals. The synchronization pattern is a simple master/worker pattern. One process is chosen as the "master" process and all other threads in the job are "workers". After a certain amount of CPU time, each worker sends a message to the master. After the master receives all messages, the workers are allowed to proceed with execution. In

our experiments, we set the time interval between synchronizations to be one CPU second (once every 32 million cycles). This is not a heavy synchronization load. Future experiments should use more complex and heavier loads.

Page Fault Detection Algorithm

Originally, we used a memory access model similar to that presented in [3] to produce a low-locality memory access pattern. However, its usage became a bottleneck in our simulator. We replaced its use with a new model based on an exponential function.

Each thread has an average memory access rate. The page fault detection algorithm determines how much time will elapse between the current time and the time when the next page fault will occur. For example, if no pages of the currently running thread are loaded into memory, then the first memory access will be a page fault and the time until the next page fault will be the same as the average memory access rate. If all pages of the thread are loaded into memory, then the amount of time until the next page fault will be infinite.

If some, but not all, pages are loaded into memory, there should be some time between page faults that is greater than the time between memory accesses, since some references will be to pages already loaded. In fact, this should happen often due to locality of memory references. We construct a model to mimic this behavior.

To start, we define the time until the next page fault to be T. We then define the average memory access rate to be R. If we assume no pages are ever loaded into memory, then the next page fault will occur in exactly this amount of time.

$T = R$

The operating system will load a page into memory when a page fault occurs. It is very likely that subsequent memory accesses will reference the same page because memory accesses, in general, follow locality. If the program accessed memory linearly,

then the memory references will traverse the entire loaded page before encountering a new, unloaded page. We assume this is the case and multiply the memory access rate by the page size P. In the case of our simulated system, P has the value 4096.

$$T = R \times P$$

No thread will ever access memory perfectly linearly and it is possible that the thread will return to pages already loaded into memory. The more data that is loaded into memory, the more likely it is that this situation will occur. We define the data brought in from page faults as D. D starts at a zero value and increases by the size of a page at each page fault event. There must be a limit on the amount of memory paged in by a single thread, and that limit is L. L is defined as the minimum of the total amount of memory overallocated on the node and the thread data size. If, for instance, the amount of memory overallocated on the current PE is 32 MB, and the current thread size is 16 MB, L is 16.

$$L = min(NodeOverallocation, ThreadSize)$$

We can calculate the amount of overallocated data O of the thread not currently in RAM by subtracting D from L.

$$O = L - D$$

From this amount, we can calculate the percentage of overallocated data not currently in RAM. We call this value U.

$$U = O/L$$

We can then apply U to the calculation of T. Dividing our previous equation by U will cause the average time until the next page fault to go up as more data is loaded into RAM. Note that when U is 0%, T will be defined as infinite.

$$T = (R \times P)/U$$

Finally, the resulting T is given to an exponential distribution as an average. The distribution will randomize the time between accesses while still following a general trend.

$T = expntl((R \times P)/U)$

One problem with this model is that it does not handle processes that are larger than physical RAM correctly. Specifically, this model will have the processes load a only finite amount of data into RAM. This is not correct for very large processes where, in order to page parts of a program in, other parts will have to be paged out. In this case, it is possible that the process will thrash between localities forever.

We avoid this problem using the following method. When all of the process' data (as determined by the model) is loaded into memory (U = 0%), the simulator will reset the amount of memory unloaded using a triangle distribution. This triangle distribution is given the parameters $A = P$, $B = P$, and $C = (RAMSize - ThreadSize)$. This makes the amount unloaded likely to be small - generating a low level of page faults for future execution. This model mimics locality changes in the program and will produce the desired behavior of continued page fault activity for very large processes.

Network RAM Access Model

The paging model is used as the foundation of the network RAM access model. When a page fault occurs, the simulator checks what percentage of virtual memory of the node is stored as network RAM and what percentage is stored on disk. A random number is chosen to decide if network RAM or disk has just been referenced.

If the disk is referenced, then the activity is no different than normal paging. If network RAM is used, a round-trip message occurs between the host and the server node storing its pages. It is assumed that the page is read from remote RAM instantaneously and the only time penalty is that caused by communication overhead.

3.1.4 Scheduler Models

It is assumed there is one centralized scheduler for the system. In our simulations we experiment with two schedulers: a space sharing scheduler and a gang scheduler. For both schedulers, we use FCFS as our queuing discipline since it has been shown to be a simple yet efficient ordering which guarantees fairness [24]. Most simple node packing schemes lead to identical performance, so we use best-fit packing to follow the example of [10]. It is assumed that the schedulers have no knowledge of the memory requirements of jobs. Neither scheduler takes these requirements into account when scheduling decisions are made.

Each time slice in the gang scheduler runs for a 60 second quantum as suggested by [29]. The time required to perform a PCS is fixed at 4 ms [4]. The maximum number of time slices (MPL) is set to two. This number is conservative and limits paging activity [24]. Alternative scheduling and slot unification are provided.

3.1.5 Metrics

There are no universally valid and accepted metrics. In fact, different metrics can give contradictory results [12, 15, 21]. However, most installations use a few "de facto" standard metrics such as response time and utilization [5]. In this section we list and justify the metrics we use in our experiments.

The first metric is average response time, or total wallclock time from submit to finish. This metric is used very often and directly reflects the goal we are attempting to achieve - improved parallel process performance. One disadvantage of this metric is that it can overemphasize large jobs. In parallel workloads, small jobs account for the majority of jobs. [10, 12, 15]

To directly compare DP to the various PNR designs, we create another metric: "optimization ratio". This metric is based on average response time and represents the improvement of a PNR design over DP.

$$OptimizationRatio(PNRDesign) = \frac{R_{DP} - R_{PNR}}{R_{DP}} \times 100\%$$

We also calculate the average and standard deviation of node memory allocation and disk allocation by sampling the memory allocation information of each node every 50,000 simulated time seconds.

Many other possibly useful metrics, such as utilization, throughput, slowdown, and bounded slowdown are not reported here. We have briefly examined these metrics and trends in response times appear to match well with trends in these metrics. However, additional comparisons with these metrics may yield interesting data and could be the focus of future work.

3.2 Experimental Setup

To determine the effectiveness of PNR in comparison to DP, we define several experiments. The basic set of experiments is defined in table 3.1. These experiments reflect our main points of interest:

- Performance under varying memory loads.
- Performance under varying network speeds.
- Performance under different network topologies.
- Performance under different scheduling strategies.

The basic experiment table becomes multidimensional as we apply it to seven different paging methods, workloads of 4,000 and 5,000 jobs, and to 64 and 128 node systems. The total number of experiments performed for this thesis based on table 3.1 is 560. Some additional experiments are performed.

RAM	Network	Topology	Space Sharing
150%	10	Bus	50%
135%	100	Star	75%
125%	1,000	Connected	100%
115%	10,000		
100%	100,000		
85%			
75%			
65%			
50%			

Table 3.1: Table of Experiments.

3.2.1 Experiment Table Detail

RAM

To test each scheme under varying memory loads we vary the amount of RAM available at each node while holding memory demands of jobs constant. The parameter given signifies the relative amount of RAM present at each node in the experiment. For instance, for the 150% experiment, we adjust the default 32 MB of RAM to 48 MB (or 150% of the original value).

Network

To test each scheme under varying network performance, we alter link bandwidth and switch processing delay. The base value of the network performance is 100 Mbps with a 80 microsecond switch processing delay. The parameter value 1000 signifies performance 10 times better than the base value - with link bandwidth at 1000 Mbps and switch processing delay at 8 microseconds. 10,000 and 100,000 values follow the same pattern. Parameter value 10 has link bandwidth of 10 Mbps but does not alter the switch processing delay.

Topology

To test each scheme under different topologies, we define three topologies:

- Star - N links and 1 switch
- Bus - 1 link
- Connected - $N(N-1)$ links - a fully connected system with links for both directions.

The star is the base topology. Note that N is the total number of nodes in the system.

Scheduler

To test each scheme under different scheduling strategies, we define both a gang scheduler and a space sharing scheduler. The gang scheduler is used as the base scheduler. The space sharing scheduler is simply the gang scheduler with an MPL of one. When we use the space sharing scheduler, we run the scheduler under varying RAM workloads. Following the same pattern as the gang scheduling experiments, we vary the RAM ratio from 50% to 100%. We do not simulate anything beyond 100% RAM because of the CM-5 scheduling policy which prevented any jobs larger than the RAM available from running.

3.2.2 Dimensions of the Experiment Table

Each experiment defined in table 3.1 is run on 4,000 and 5,000 job workloads. We discovered in early experiments that jobs start to become backlogged under the gang scheduler with the 5,000 job workload. By comparing the 4,000 and 5,000 job workloads we can understand how each solution performs under different types of system loads.

Each experiment is run on a 64 node cluster and a 128 node cluster. Running experiments on different-sized systems will indicate how well the various designs scale. Note that each workload was originally collected on a 1,024 node system and if the system is scaled down, the workload is scaled with it. For example, a job that asked for 512 processors under the original workload will ask for 32 processors in the 64 node system, and 64 processors in the 128 node system. Congestion on the larger networks will increase since more communication is required for synchronization and broadcasts.

We run seven different paging methods against each experiment. The base is disk paging (DP). We test six PNR methods: the centralized method (CEN), the client-only method (CLI), the local managers method (MAN), and three variants of the backbone method (BB). The three backbone methods define the number of PNR managers to be $\frac{1}{4}$th, $\frac{1}{8}$th, and $\frac{1}{16}$th the total number of nodes in the system. For instance, on the 64 node system, the number of managers is defined as 16 (BB16), 8 (BB8), and 4 (BB4), respectively.

Chapter 4

Performance Evaluation

4.1 64 Node Cluster

This section describes the results of the experiments introduced in table 3.1 run on a simulated 64 node cluster. Note that we define the "base" set of experiments as experiments where all of the default parameters described in chapter 3 are set.

4.1.1 Base Experiments

In figure 4.1 we see the results for the base set of experiments for five different workloads. Response time for each design follows an exponential curve as workloads get larger. For the 5,000 job workload, jobs are starting to get backlogged. This increases average response time for all paging designs. This trend continues for the 6,000 job workload (not shown). We focus on the 4,000 and 5,000 job workloads to observe the difference between backlogged and non-backlogged systems.

Figure 4.2 shows the 4,000 job workload where all PNR designs are close in performance. Each design has an optimization ratio of 10-12% over disk paging (figure 4.3). Interestingly, CLI has the best optimization ratio at 12.08%. This goes against the expectation that CLI will always lead to poor performance due to lack of

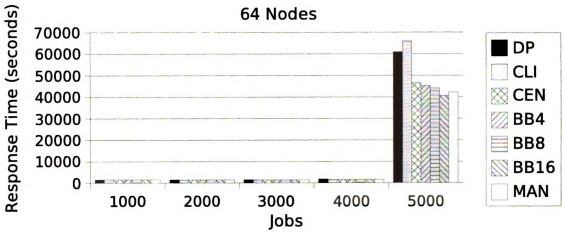

Figure 4.1: Base experiment - 64 PE - several workloads - response time.

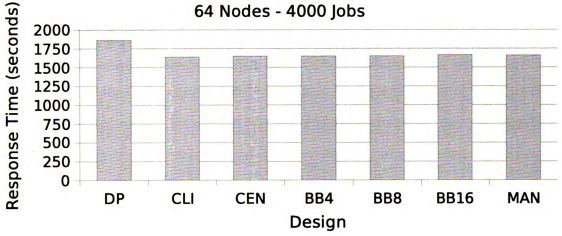

Figure 4.2: Base experiment - 64 PE - 4,000 jobs - response time.

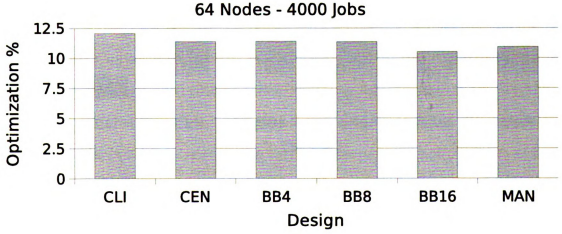

Figure 4.3: Base experiment - 64 PE - 4,000 jobs - optimization ratio.

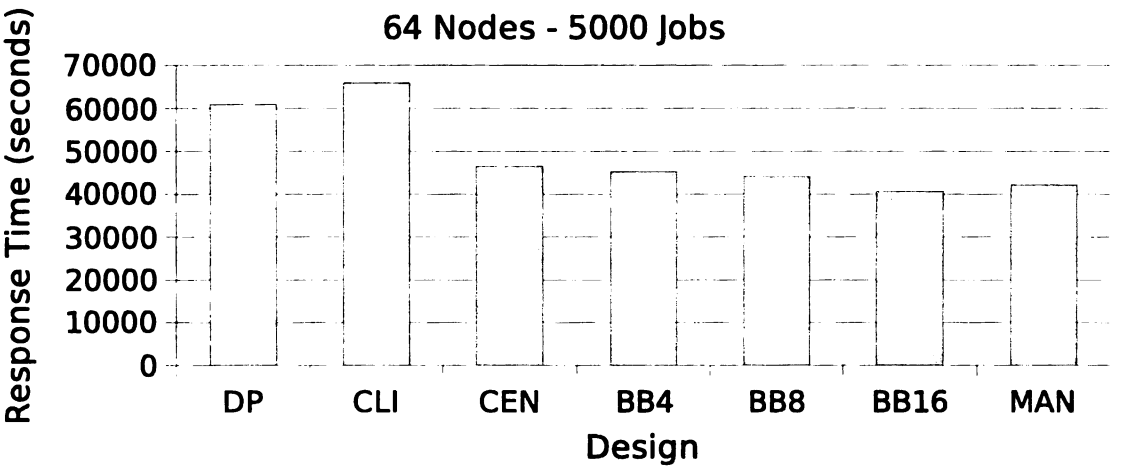

Figure 4.4: Base experiment - 64 PE - 5,000 jobs - response time.

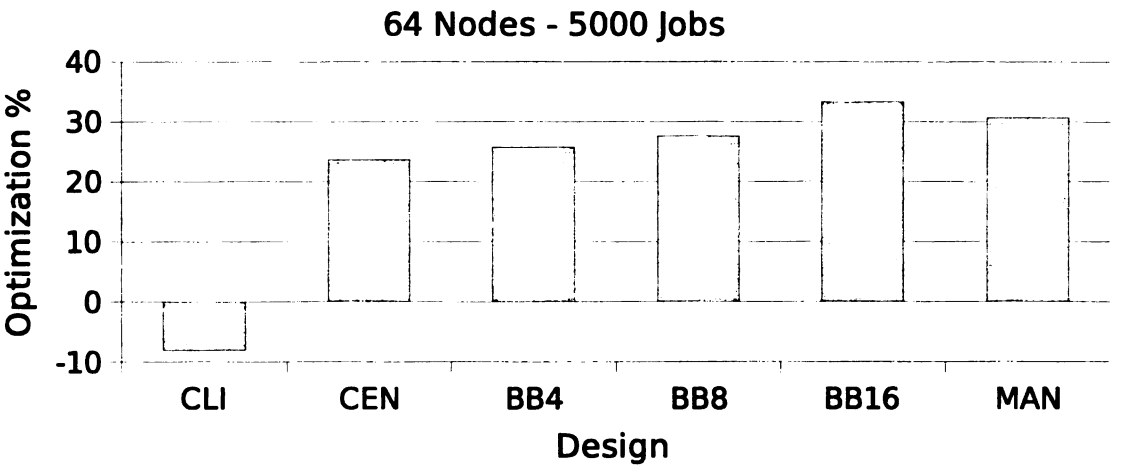

Figure 4.5: Base experiment - 64 PE - 5,000 jobs - optimization ratio.

servent coordination.

The average memory usage for DP was 6.88 MB, with a standard deviation of 4.96 MB. All PNR designs had a standard deviation around 4.4 MB. CLI did not lead to significantly uneven memory allocation (as was expected).

Figure 4.4 demonstrates that, for the 5,000 job workload, the PNR designs are more differentiated. CLI was the only design with an average response time higher than DP's. Figure 4.5 shows that BB16 is the best design at a 33.38% speedup over DP. This figure shows us that BB16 is at the top of a curve, where performance of the coordinating designs is worst at CEN, gets better as the solution becomes more distributed (BB4 and BB8) and degrades past a certain point of distribution (MAN). These results are more in line with our initial hypothesis that a hybrid of centralized and decentralized approaches would have the best performance.

All non-CLI PNR methods experienced approximately 70% or less the number of page faults DP experienced. This can be attributed to the fact that, since PNR jobs finish faster, they experience fewer PCSs. This, in turn, leads to less page loading due to memory contention by multiple processes.

We observe some interesting statistics for memory allocation for each design. DP had highest average memory allocation at 17.87 MB per node with standard deviation of 5.17 MB. All PNR designs had lower allocation averages - ranging from 14.45 MB to 16.37 MB. BB16 had the lowest average at 14.45 MB, but it also had the highest standard deviation of all designs at 5.76 MB! In fact, each PNR design except for MAN had a higher standard deviation than DP - ranging from 5.18 to 5.76 MB (with MAN having 4.98 MB).

This results comes as a surprise, since PNR should be smoothing out memory usage, not creating more hotspots! Further, it may be possible that BB16 had the lowest response time only by chance - since the hotspots created were not in heavy use by the system.

4.1.2 RAM Variations

For both the 4,000 and 5,000 job workloads, we see response times for all designs converge as RAM is increased (figures 4.6 and 4.9). In the 4,000 job workload, response time converges earlier. In the 5,000 job workload, the designs converge later (due to backlog) at RAM ratio 135%. Data is omitted from the figures when optimization rate is 0%. As RAM is decreased, all designs increase response time exponentially. Presumably, response times will also converge when so little RAM is available that allocating network RAM is impossible.

At the 100% RAM ratio and above in the 4,000 job workload all PNR methods use 0 MB of disk space on average. Standard deviation for each is 0 MB. This accounts for similarity of results for the PNR methods.

For the 4,000 job workload, we see a very large initial boost in performance as RAM is decreased. The optimization ratio is the among the highest observed for all experiments - in the 75% to 100% range (figure 4.7). This advantage quickly evaporates at 65%, where most designs actually underperform DP. At 50%, most of the PNR designs recover, but have a lower performance benefit (between 25-50%).

CLI is the clear loser in each of the experiments where the RAM ratio is less than 100%. MAN consistently outperforms or performs as well as other methods in each scenario. The other methods are more difficult to judge, since their performance fluctuates.

In the 5,000 job workload, one immediately obvious feature is the big performance boost PNR receives as RAM is initially increased (figure 4.9). The 4,000 job workload experienced no such benefit. The extra RAM must help PNR efficiently clear out the backlog of jobs. As RAM is increased past a certain point (around 135%) PNR and DP converge.

Interestingly, despite these favorable response times at RAM ratio 115% and 125%, the average allocated memory and standard deviation metrics are not unusually

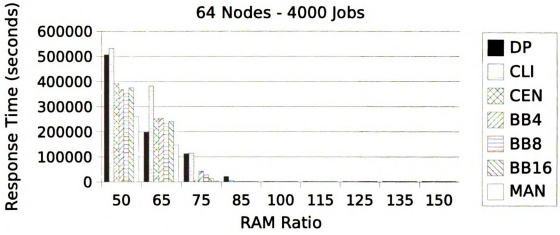

Figure 4.6: RAM experiments - 64 PE - 4,000 jobs - response time.

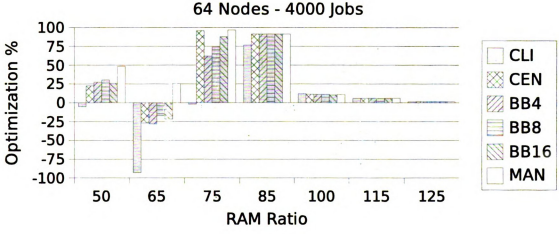

Figure 4.7: RAM experiments - 64 PE - 4,000 jobs - optimization ratio.

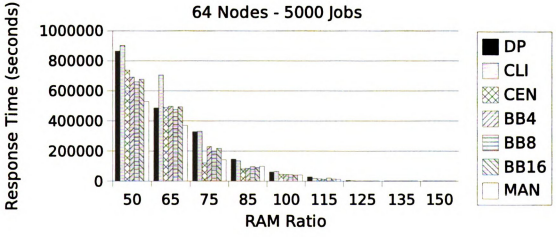

Figure 4.8: RAM experiments - 64 PE - 5,000 jobs - response time.

favorable. In fact, some PNR methods have an average and standard deviation higher than DP. The differentiating characteristic between DP and PNR must be the average disk usage. While DP uses about 1.25 MB on average, each PNR method uses about 0.5 MB or less at RAM ratio 115%. Similarly, at 125% RAM ratio, DP uses 0.32 MB whereas each most PNR methods use less than 0.1 MB.

When RAM becomes more scarce MAN offers consistent performance benefits but this time it is not always the best design. At 85% and 75% we observe smaller-scale versions of the performance boosts observed in the 4,000 workload. At 65% there is another performance hit - but, again, at a smaller scale. At 50%, modest improvements are back.

4.1.3 Network Performance Variations

For all cluster setups and workloads, the performance of PNR at 10 Mbps of bandwidth was so poor, it was not included in the figures for these experiments. To get an idea of the performance, the total simulated time elapsed for the 5,000 workload rose an order of magnitude from 10^6 to 10^7 seconds. This result shows that PNR is extremely sensitive to low-performance networks.

Figures 4.10 and 4.11 show the 4,000 job workload, where the methods with the most coordination overhead suffer at bandwidth 100 Mbps. However, as bandwidth is increased beyond 100 Mbps, all PNR methods become equivalent in terms of response times and optimization ratios.

No PNR method in these experiments uses disk paging at any bandwidth. The limiting factor here is coordination overhead. It makes sense that as communication becomes less costly, coordination overhead becomes such a small factor that the various methods converge to the same response time.

Figure 4.12 and 4.13 show that, for the 5,000 job workload, the general curve observed in the base test is preserved until bandwidth 100,000 Mbps, where the results

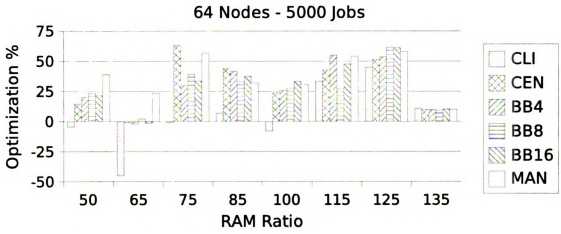

Figure 4.9: RAM experiments - 64 PE - 5,000 jobs - optimization ratio.

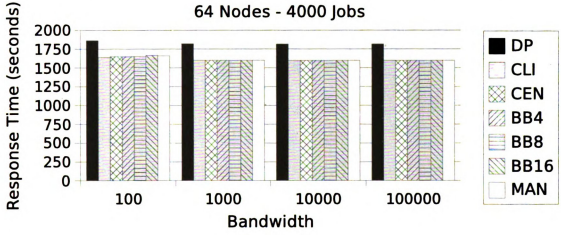

Figure 4.10: Network experiments - 64 PE - 4,000 jobs - response time.

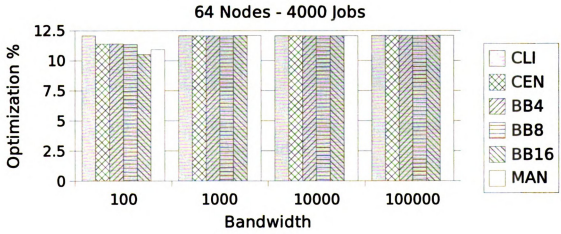

Figure 4.11: Network experiments - 64 PE - 4,000 jobs - optimization ratio.

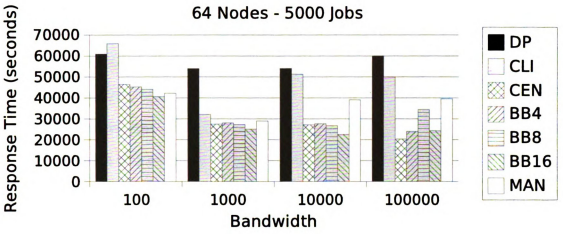

Figure 4.12: Network experiments - 64 PE - 5,000 jobs - response time.

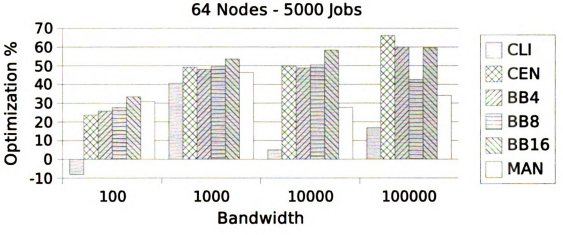

Figure 4.13: Network experiments - 64 PE - 5,000 jobs - optimization ratio.

are noisy. Increases in network performance benefits CEN the most but also tends to increase the performance of each method.

CLI performs worse than every other PNR method in all cases. However, for bandwidth greater than 100 Mbps, CLI still manages to beat DP. In fact, CLI even has lower average memory usage (5.46 MB) and standard deviation (2.02 MB) compared to DP (6.09 and 2.71 MB).

For this experiment, synchronization overhead must not be the only limiting factor. The backlog of jobs must be influencing the experiment and producing noisy results. The phenomena may be explained if we consider that if processes are becoming backlogged in the 5,000 job workload, and if network performance results in higher job turnover, then longer-running jobs will tend to start running together in greater frequency than under lower-performing networks. Longer-running jobs tend to demand more memory than shorter jobs so, given this situation, memory load and response times will rise.

4.1.4 Network Topology Variations

For the 4,000 job workload (figures 4.14 and 4.15), PNR designs are superior to DP on all topologies. For the bus topology, it appears that less coordination overhead results in higher performance. This is not too surprising since the potential for congestion on a bus network is very high. For the connected network, we observe results similar to those at the high bandwidth network. Each PNR design has roughly equivalent performance.

Since the connected network does not increase bandwidth, and since the optimization ratio is similar to that found on the high-performance networks, we can conclude that the significant factor in this experiment and in the network experiments is congestion. The problem with low-performing networks was not simply the message service time, but the queuing delay introduced by the slower-moving messages.

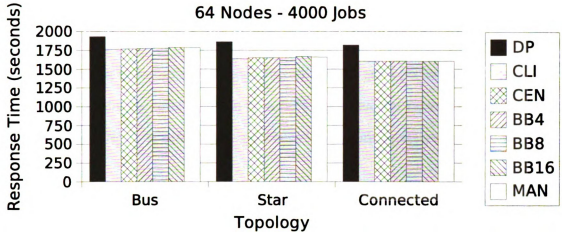

Figure 4.14: Topology experiments - 64 PE - 4,000 jobs - response time.

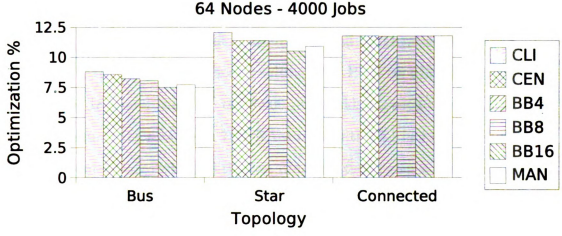

Figure 4.15: Topology experiments - 64 PE - 4,000 jobs - optimization rate.

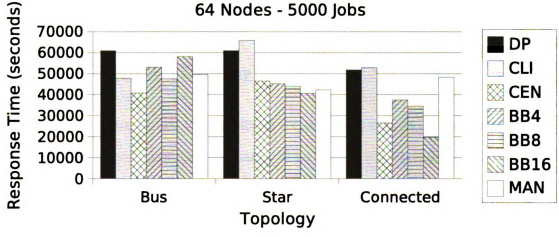

Figure 4.16: Topology experiments - 64 PE - 5,000 jobs - response time.

For the 5,000 job workload (figure 4.16) we have noisier results. In the bus topology, the general trend still appears to be the less synchronization, the better. For the connected topology, all designs (including DP) get a performance boost except MAN. BB16 is the superior method at a 60% optimization ratio. Notably, CLI still underperforms DP. It is difficult to say there is a definite pattern in these results.

4.1.5 Space Sharing Scheduler Experiments

For all workloads, recall that no individual job is larger than the CM-5's available RAM. So, with a space-scheduler, no two jobs will be loaded on one node and, at RAM ratio 100%, no node should ever be overloaded. All designs should experience nearly identical performance. This result is confirmed in all RAM ratio 100% space-sharing experiments (with trivial differences resulting from PNR coordination methods).

Figure 4.18 shows that, for the 4,000 job workload, each method has approximately the same performance at 75% RAM ratio with CLI having a slight edge. For 50%, CLI loses its edge while all the rest of the methods gain a performance boost.

Figure 4.21 shows the same results for the 5,000 job workload. Each method has approximately the same performance at 75%. CLI again loses its performance benefits at 50% while the rest of the PNR methods approach a respectable 45% optimization ratio.

4.2 128 Node Cluster

This section describes the results of experiments run on a simulated 128 node cluster. These experiments are primarily done to test scalability as compared to the 64 node cluster.

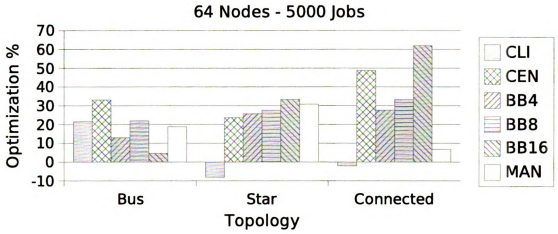

Figure 4.17: Topology experiments - 64 PE - 5,000 jobs - optimization ratio.

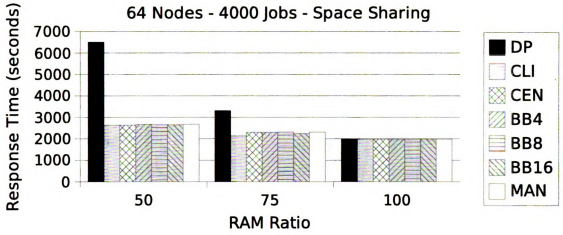

Figure 4.18: Space sharing experiments - 64 PE - 4,000 jobs - response time.

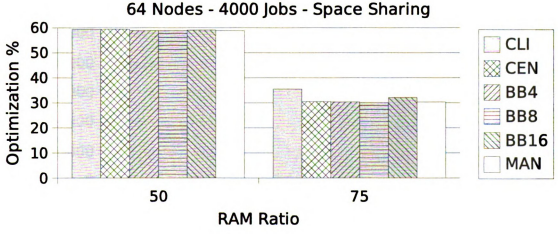

Figure 4.19: Space sharing experiments - 64 PE - 4,000 jobs - optimization ratio.

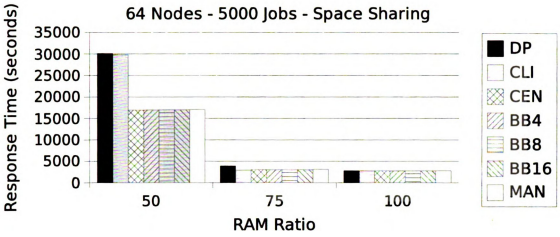

Figure 4.20: Space sharing experiments - 64 PE - 5,000 jobs - response time.

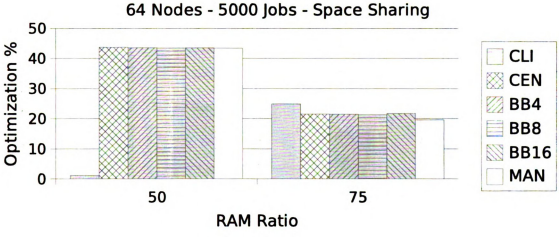

Figure 4.21: Space sharing experiments - 64 PE - 5,000 jobs - optimization ratio.

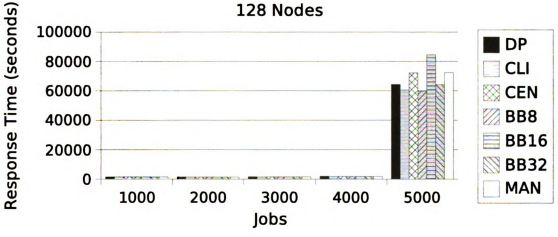

Figure 4.22: Base experiment - 128 PE - several workloads - response time.

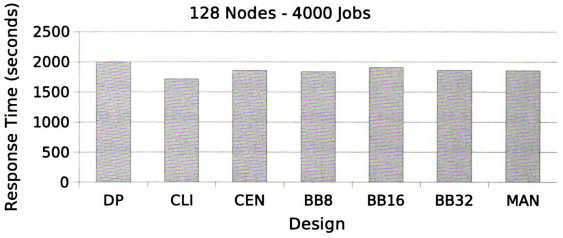

Figure 4.23: Base experiment - 128 PE - 4,000 jobs - response time.

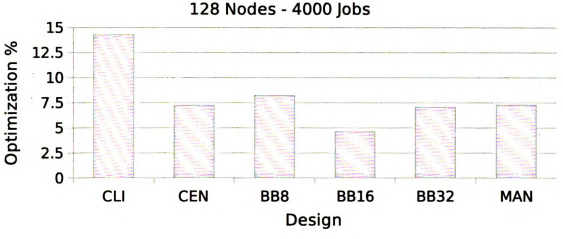

Figure 4.24: Base experiment - 128 PE - 4,000 jobs - optimization ratio.

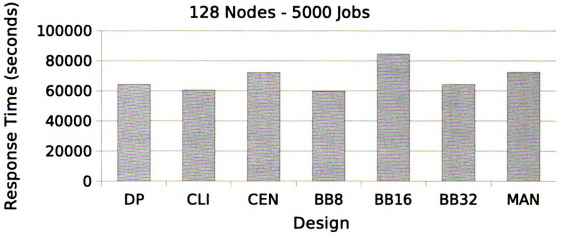

Figure 4.25: Base experiment - 128 PE - 5,000 jobs - response time.

4.2.1 Base Experiments

When we run the base set of experiments over various workloads (shown in figure 4.22), we see the same exponential curve in response time as in the 64 node system.

In the 4,000 job workload, the CLI optimization ratio beats the other PNR designs by a significant margin (figure 4.24). While each other PNR design is between 5% and 7.5% optimization ratio, CLI approaches 15%. CLI also had the lowest average memory allocated and the lowest standard deviation at 5.52 MB and 4.35 MB, respectively. DP had an average of 7.54 MB and standard deviation of 5.23 MB and MAN, for instance, had an average of 6.28 MB and a standard deviation of 4.65 MB.

In the 5,000 job workload there appears to be scalability problems. CLI and BB8 are the only methods that experience significant performance enhancement (about 5%). Others (CEN, BB16, MAN) experience performance degradation.

This result is not unexpected since the underlying network topology is a star. The bottleneck resource, the central switch, becomes more congested as the network becomes larger and as more messages are sent. As observed earlier, PNR is very sensitive to poor network performance and it makes sense that PNR performs poorly here. CLI has a natural advantage in larger networks because individual peers do not synchronize with each other. With each other design, coordination overhead must be incurred.

Interestingly, each PNR design had a lower standard deviation in memory usage than DP. Each PNR design also used less disk space than DP. However, some PNR designs had a significantly higher average memory allocation. All PNR designs still experienced fewer page faults. The number of page faults ranged from 50% to 80% that of DP.

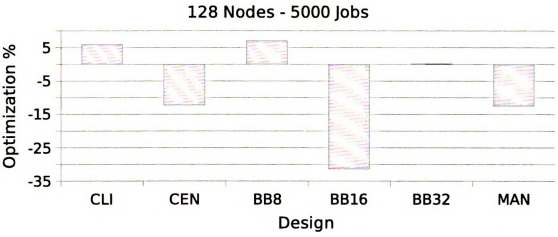

Figure 4.26: Base experiment - 128 PE - 5,000 jobs - optimization ratio.

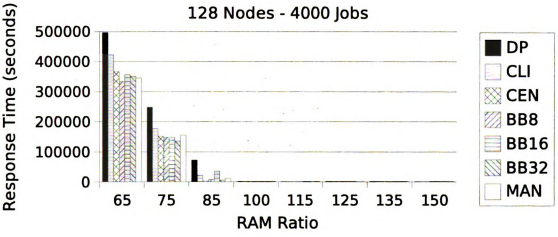

Figure 4.27: RAM experiments - 128 PE - 4,000 jobs - response time.

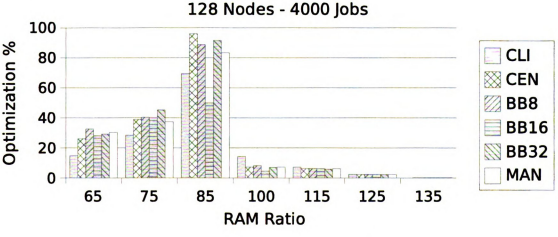

Figure 4.28: RAM experiments - 128 PE - 4,000 jobs - optimization ratio.

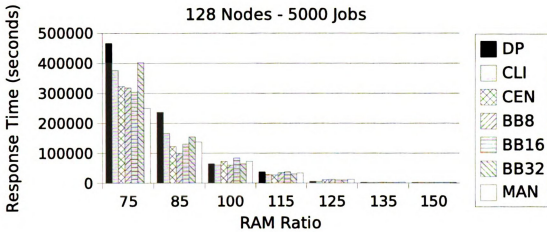

Figure 4.29: RAM experiments - 128 PE - 5,000 jobs - response time.

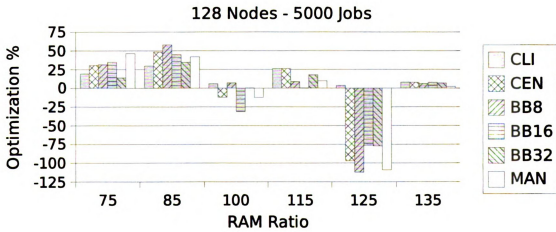

Figure 4.30: RAM experiments - 128 PE - 5,000 jobs - optimization rate.

4.2.2 RAM Variations

Note that there is data missing at 65% and 50% RAM ratios for both the 4,000 and 5,000 job workloads due to simulation time constraints.

For the 4,000 job workload (shown in figures 4.27 and 4.28) PNR converges as expected as RAM is increased. As in the 64 node cluster, large performance increases are observed for 85% and 75% RAM ratios (approaching 100% and 50% improvement, respectively) The improvement at 75% RAM ratio is not as pronounced as in the 64 node system. Interestingly, the $\frac{1}{8}$ backbone solution experiences a performance hit in both the 64 node and 128 node systems. The performance penalty is more pronounced in the 128 node system.

For the 5,000 job workload (in figures 4.29 and 4.30), we observe the PNR optimization ratio converging as expected. However, for 125% RAM ratio, there is a large performance penalty felt by all of the PNR designs (with CLI being affected the least). All PNR designs except CLI experience at least a -100% optimization ratio. All designs improved average response time from 115% to 125%, but DP improved more than each PNR method. It is not known if this is a statistical fluke or something more interesting. More tests need to be performed.

As RAM becomes more scarce, the optimization ratio of the PNR methods don't reach as high as in the 64 node network (not much above 50%). As in the 64 node network, the best performance gains are observed at 85% and 75% RAM ratios.

4.2.3 Network Performance Variations

For the 4,000 job workload, PNR performance improves as soon as network performance is improved. For network bandwidths of 1,000 Mbps and higher, the average response time of each PNR design is equivalent (figure 4.31). Each PNR design is superior to using DP. The optimization ratio settles at about 12.5% (figure 4.32).

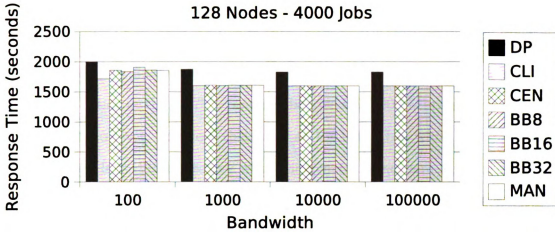

Figure 4.31: Network experiments - 128 PE - 4,000 jobs - response time.

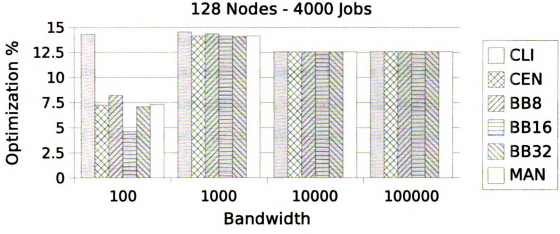

Figure 4.32: Network experiments - 128 PE - 4,000 jobs - optimization ratio.

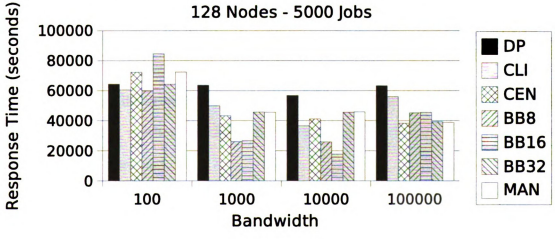

Figure 4.33: Network experiments - 128 PE - 5,000 jobs - response time.

For the 5,000 job workload, each method improves dramatically with higher bandwidth. Similar to the 64 node system, each PNR design outperforms DP at network bandwidth 1,000 Mbps and higher. BB8 and BB16, in particular, have optimization ratios in the 50-75% range at bandwidth 1,000 Mbps and 10,000 Mbps (figure 4.34). However, they lose their lead at 100,000 Mbps while CEN, BB32 and MAN all gain. It is difficult to say what the general trend here is, other than that better network performance tends to elevate the performance of each PNR design.

4.2.4 Network Topology Variations

For the 4,000 job workload, we do not show the results for the bus topology in figure 4.35 or 4.36. This is because the bus response times were orders of magnitude higher than the star and connected topologies. For the CEN method, for instance, the optimization ratio was a staggering -2,500%! Obviously the bus network does not scale well and PNR is hit hard by this. With the connected topology, each method is close to equivalent.

Figures 4.37 and 4.38 show the 5,000 job workload results. Just as in the 4,000 job workload, the bus network causes all PNR designs to behave poorly. CLI is least effected and CEN experiences a huge performance hit - almost reaching -300% (figure 4.38)! It is interesting that this negative performance is better in comparison to the 4,000 job workload. In the 4,000 job workload, the PNR methods all use less disk space than the PNR methods do in the 5,000 job workload. We speculate that the bus network actually drives average network service time higher than disk service time, and the PNR methods in the 4,000 job workload use network RAM more than in the 5,000 job workload.

The connected network gives results similar to the high bandwidth experiments. Here, all PNR designs have better performance than DP. BB8 and BB16 are the clear winners at an optimization ratio greater than 50%.

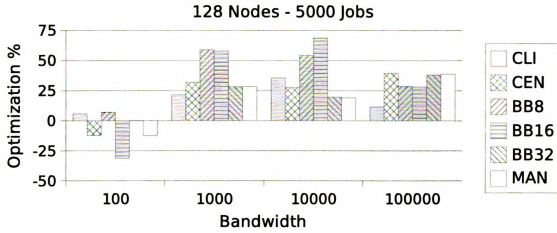

Figure 4.34: Network experiments - 128 PE - 5,000 jobs - optimization ratio.

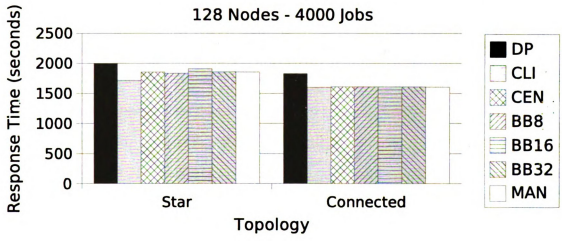

Figure 4.35: Topology experiments - 128 PE - 4,000 jobs - response time.

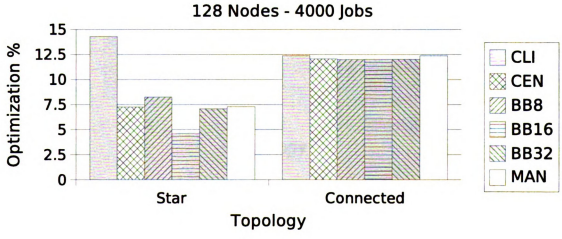

Figure 4.36: Topology experiments - 128 PE - 4,000 jobs - optimization ratio.

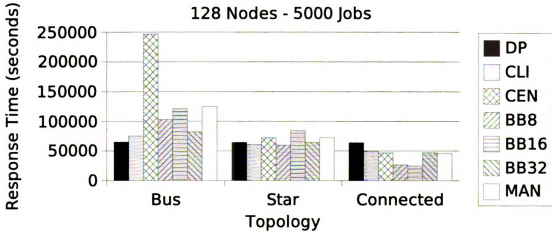

Figure 4.37: Topology experiments - 128 PE - 5,000 jobs - response time.

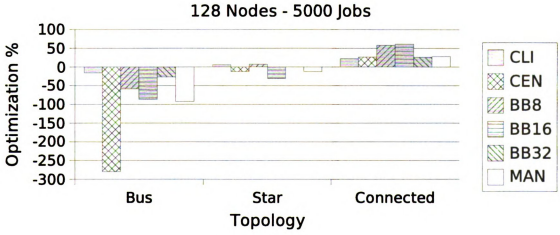

Figure 4.38: Topology experiments - 128 PE - 5,000 jobs - optimization ratio.

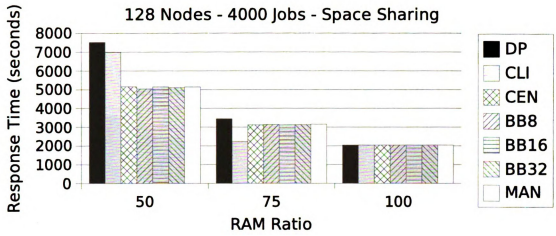

Figure 4.39: Space sharing experiments - 128 PE - 4,000 jobs - response time.

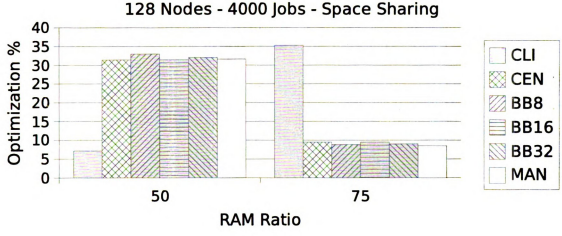

Figure 4.40: Space sharing experiments - 128 PE - 4,000 jobs - optimization ratio.

4.2.5 Space Sharing Scheduler Experiments

The results for both workloads echo those of the 64 node network and are shown in figures 4.39, 4.40, 4.41, and 4.42. In both sets of results, designs are equivalent at RAM ratio 100%, CLI has an advantage at RAM ratio 75%, and CLI loses its advantage at RAM ratio 50%. At RAM ratio 75% and 50% all non-CLI PNR designs experience almost identical performance.

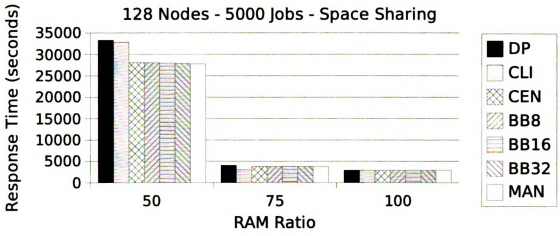

Figure 4.41: Space sharing experiments - 128 PE - 5,000 jobs - response time.

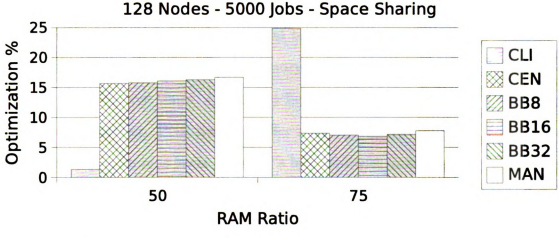

Figure 4.42: Space sharing experiments - 128 PE - 5,000 jobs - optimization ratio.

Chapter 5

Analysis

5.1 Discussion

We have described our results in chapter 5. We now interpret these results to discover the general models of performance PNR follows. Also, we identify platforms and configurations on which PNR would be useful.

5.1.1 Memory Load

Given our observations, it is clear that both PNR and DP follow an exponential curve as memory load is changed. As memory load increases, PNR and DP both tend toward infinite response times. As memory load decreases, PNR and DP will converge on a constant number. PNR does not offer a fundamentally difference performance curve, but PNR's curve is lower than DP's by a constant factor.

This model implies that adding PNR to lightly loaded systems (with high performance networks) should not harm performance and under moderate memory loads PNR can lead to large improvements over disk paging. As observed, this improvement can approach 100%. However, as memory becomes too scarce to share, PNR performance will converge to DP performance.

One surprising result of our simulations is that the proposed PNR designs do not necessarily smooth out memory usage as measured by the standard deviation of system memory usage. For some experiments, PNR memory usage was even more non-uniform than DP's memory usage. This indicates that more work must be done to ensure PNR itself does not create more overloaded nodes.

Also, some of the exceptions to our proposed model are troubling. Specifically, the 125% RAM ratio experiment on the 128 node cluster unexpectedly had most PNR methods with an optimization ratio of -100% or more. Other experiments had occasional incidents where all PNR methods do very well except for one (e.g. 85% RAM Ratio on 128 node system with 4,000 job workload). To determine if this is a problem with PNR or just an artifact, we need to re-run our experiments with different random number seeds, different portions of our collected workload, and with different workloads.

5.1.2 Network

PNR is very sensitive to network performance. The general shape of results is another exponential curve, where PNR response time tends toward infinity as network service time is increased and converges to some constant number as service time decreases. Since low network performance results in low PNR performance, PNR should not be considered on systems with low bandwidth or high message RTT times. A standard 100 Mbps network should be considered a minimum for satisfactory performance.

Topology is also an important consideration. Our experiments show that congestion is a major source of waiting time. In order for PNR to scale, it will have to run on a network that scales well as nodes are added. Bottlenecks, such as the central switch in the star topology, will quickly degrade PNR performance as nodes are added. Switching to a network topology that has no bottlenecks has the same

effect as upgrading a topology with bottlenecks to a very high speed network. To get maximum performance, PNR should be used on a fast network with few (or no) bottlenecks.

When network performance is high and RAM is relatively plentiful, we observe that each PNR design has equivalent performance. This is because coordination of memory resources is not a crucial issue when RAM is plentiful, and coordination overhead is less important with a fast network. These are the two main differentiating factors among the PNR designs, and when they are eliminated, they all perform similarly.

5.1.3 Scheduling

On a space sharing system, as RAM becomes more scarce, each PNR method except CLI appears to perform equivalently. The main limiting factor on the space sharing system is network RAM allocation coordination. Network speed is less important because PCSs are eliminated on this system - which eliminates a large portion of network activity (i.e. the large amount of page faults after a PCS). This reduces overall congestion. All of the schemes that use coordination appear to work equally well in each space sharing system tested. Depending on the workload and cluster setup, the best performance gains can be anywhere between 15-60%. If only a light load is expected, CLI is the best choice for a space sharing system.

On the gang scheduling systems, however, network performance is crucial to PNR performance. PCS's introduce large, regular bursts of network activity and PNR designs that use the network to coordinate allocation suffer. When a high-performance network is available, however, PNR can produce pronounced performance gains. Under the 4,000 job workloads, the maximum speed up using PNR was typically in the 10-12.5% range, although when RAM became scarce, optimization ratio could reach as high as 100%. For the 5,000 job workload, the maximum optimization ratio

observed was 75%, but large ratios were more common than in the 4,000 job workload.

It is interesting to note that the absolute response times of DP and PNR on the space sharing scheduler are better than the gang scheduler for the 5,000 job workload. Backlogged jobs introduce a huge paging penalty to both PNR and DP. However, for the 4,000 job workload, gang scheduling results in better response times, since no backlogging is occurring and the benefits of time sharing outweigh paging penalties.

For heavily-loaded systems, PNR can significantly reduce the response time of jobs as compared to DP. However, it may make sense to attempt to avoid paging entirely by using a space sharing system instead. Using a small MPL in conjunction with PNR may produce results better than both alone. For moderately to lightly loaded systems, the advantages of time-sharing outweigh the disadvantages of increased paging, and PNR is recommended to reduce average response times.

5.1.4 PNR Designs

CLI does surprisingly well in certain situations. If RAM is plentiful, CLI benefits since it requires very little overhead to allocate network RAM. When the network is slow, CLI performs better than its counterparts because it contributes relatively little to network congestion. If it is known that a system has a slow network and will be lightly loaded, CLI may be the best method to implement due to its simplicity and low overhead costs.

However, CLI performs poorly in heavy-load situations where RAM is scarce. In these situations, it becomes necessary for network RAM allocation to be coordinated to avoid wasting it. In these situations, that leaves us with CEN, BBX, and MAN. When the network is fast, all three can potentially be used effectively. When the network is slow, only BBX and MAN should be considered since CEN will introduce significant congestion on the network resources that service the central server.

It is difficult to choose between the BBX and MAN methods. There appears

	Low Load	High Load
Fast Network	ANY	CEN, BBX, MAN
Slow Network	CLI	BBX

Table 5.1: Recommended PNR Design Usage.

to be a tendency for BB16 to be the superior method in the 64 node network ($\frac{1}{4}$ of nodes) and BB8/BB16 to be superior in the 128 node network ($\frac{1}{16}$ and $\frac{1}{8}$ of nodes). However, no definite conclusion can be reached from the results gathered. We will tentatively claim that BBX has an advantage over MAN in slow networks because less messages must be broadcast for synchronization purposes.

If the network is fast, and RAM is plentiful, then each of the PNR designs are equivalent. Any one could be used. It may make sense to use CLI in this situation since it eliminates the need for a manager (proxy) entirely.

A summary of these recommendations is in table 5.1.

5.2 Conclusion

In this thesis we identified a novel way of reducing page fault service time and better utilizing memory resources in a cluster system running parallel processes. This method, which we call Parallel Network RAM, uses remote idle RAM as another tier in the memory hierarchy for parallel jobs in clusters. We proposed several different PNR designs and evaluated the performance of each under different conditions. We discovered that different designs are appropriate in different situations and that simply applying PNR to an existing system is not a guarantee of increased performance.

Applied correctly to lightly loaded systems, performance gains between 10-25% can be achieved. Applied correctly to heavily loaded systems, performance gains can be as high as 100%. Applied incorrectly, PNR can result in extreme performance losses. The highest observed in our simulations were approximately -2500%.

Parallel Network RAM is probably best applied in conjunction with other methods to reduce page load. For instance, a combination of job migration, Parallel Network RAM, heuristics used to guess memory usage at the scheduler, and other scheduler techniques would likely result in better performance than PNR alone.

5.3 Future Work

This thesis only took a survey of PNR performance given a variety of cluster configurations and did not focus on any particular experiment. Our experiments' statistical significance should be confirmed by re-running our experiments with different random number seeds and with different subsets of the workload trace.

A surprising result of our work was that no single PNR design worked well under all conditions. This indicates that, to create a single, unified PNR design that works under a variety of cluster conditions, a design that can intelligently change its behavior based on current cluster conditions should be developed. Such an algorithm could simply use the designs proposed in this thesis, but switch among which design is being used based on the current state of the cluster. This would make the implementation of PNR much easier, since only one system needs to be developed for all clusters.

This thesis only used a specific subset of a single trace from a particular parallel system. To our knowledge, traces such as the one we used that include memory information are very rare. It would be invaluable to collect a new trace on a modern parallel system running cutting-edge parallel programs. Not only would this allow us to compare how PNR acts on a modern system, it would also allow us to calibrate our simulator and ensure that the models used (e.g. page fault models) are realistic.

This simulator only explores jobs that use a static amount of memory determined at runtime. It would be useful to know how PNR performs given a more realistic dynamic memory workload. The simulator also only simulates a very simplistic job

synchronization model. A more intense and complex model would give us more realistic results.

Our approach only allocates and deallocates network RAM at job start and stop time. It would be interesting to know if allowing the PNR system to allocate and deallocate memory at additional times would lead to increased performance. A more periodic approach may better accommodate memory usage.

This approach does not consider job migration. A system which is allowed to choose between job migration and PNR to balance load would be very interesting and may potentially combine advantages from both methods.

A comparison of PNR methods against methods which attempt to avoid paging by guessing memory usage would be interesting. One method may be superior to another or, more likely, the combination of the two methods may yield increased performance.

Bibliography

[1] Anurag Acharya and Sanjeev Setia. The utility of exploiting idle memory for data-intensive computations. Technical Report TRCS98-02, 1998.

[2] A. Barak and A. Braverman. Memory ushering in a scalable computing cluster. *Journal of Microprocessors and Microsystems*, 22(3-4):175–182, August 1998.

[3] Anat Batat and Dror G. Feitelson. Gang scheduling with memory considerations. In *14th Intl. Parallel Distributed Processing Symp.*, pages 109–114, 2000.

[4] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. *The Journal of Supercomputing*, 10(1):87–104, 1996.

[5] Steve J. Chapin, Walfredo Cirne, Dror G. Feitelson, James Patton Jones, Scott T. Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 67–90. Springer-Verlag, 1999. Lect. Notes Comput. Sci. vol. 1659.

[6] R.M. Cubert and P. Fishwick. Sim++, version 1.0. Technical report, Department of Computer and Information Science and Engineering, University of Florida, 28 July 1995.

[7] Allen B. Downey. Lachesis: A job scheduler for the Cray T3E. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 47–61. Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[8] G. F. Pfister F. Darema-Rogers and K. So. Memory access patterns of parallel scientific programs. *Performance Evaluation Review*, 15(1):46–57, 1987.

[9] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing global memory management in a workstation cluster. In *Symposium on Operating Systems Principles*, pages 201–212, 1995.

[10] Dror G. Feitelson. Packing schemes for gang scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 89–110. Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[11] Dror G. Feitelson. Memory usage in the LANL CM-5 workload. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 78–94. Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[12] Dror G. Feitelson. Metrics for parallel job scheduling and their convergence. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 188–205. Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[13] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–18. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[14] Dror G. Feitelson and Larry Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *Journal of Parallel and Distributed Computing*, 35(1):18–34, 1996.

[15] Dror G. Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–24. Springer-Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[16] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[17] Paul A. Fishwick. *Simulation Model Design and Execution.* Prentice-Hall Inc., 1995.

[18] Michail D. Flouris and Evangelos P. Markatos. *High Performance Cluster Computing*, chapter 16, pages 383–408. Prentice Hall, 1999.

[19] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Overhead analysis of preemptive gang scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 217–230. Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[20] David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the Maui scheduler. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[21] James Patton Jones and Bill Nitzberg. Scheduling for parallel supercomputing: A historical perspective of achievable utilization. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer-Verlag, 1999. Lect. Notes Comput. Sci. vol. 1659.

[22] Virginia Lo, Jens Mache, and Kurt Windisch. A comparative study of real workload traces and synthetic workload models for parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 25–46. Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[23] Evangelos P. Markatos and George Dramitinos. Implementation of a reliable remote memory pager. In *USENIX Annual Technical Conference*, pages 177–190, 1996.

[24] Yanyong Zhang Anand Sivasubramaniam Hubertus Franke Jose E. Moreira. Improving parallel job scheduling by combining gang scheduling and backfilling techniques. *IPDPS*, pages 133–142, 2000.

[25] John Oleszkiewicz and Li Xiao. Pnrsim: A parallel network ram simulator. Technical Report MSU-CSE-04-13, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, April 2004.

[26] Eric W. Parsons and Kenneth C. Sevcik. Benefits of speedup knowledge in memory-constrained multiprocessor scheduling. *Performance Evaluation*, 27/28(4):253–272, 1996.

[27] Eric W. Parsons and Kenneth C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. In *Measurement and Modeling of Computer Systems*, pages 57–67, 1996.

[28] V. Subramani S. Srinivasan, R. Kettimuthu and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. *Proc. of 2002 Intl. Workshops on Parallel Processing (held in conjunction with the 2002 Intl. Conf. on Parallel Processing, ICPP 2002)*, 2002.

[29] Uwe Schwiegelshohn and Ramin Yahyapour. Improving first-come-first-serve job scheduling by gang scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 180–198. Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[30] Senjeev K. Setia. The interaction between memory allocation and adaptive partitioning in message-passing multicomputers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 146–164. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[31] Kenneth C. Sevcik and Songnian Zhou. Performance Benefits and Limitations of Large NUMA Multiprocessors. In *Proceedings of Performance '93*, pages 183–204. Elsevier Science Publ, Sept 93.

[32] Quinn O. Snell, Mark J. Clement, and David B. Jackson. Preemption based backfill. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 24–37. Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.

[33] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 55–71. Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.

[34] Fang Wang, Marios Papaefthymiou, and Mark Squillante. Performance evaluation of gang scheduling for parallel and distributed multiprogramming. In

Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 277–298. Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[35] William A. Ward, Jr., Carrie L. Mahood, and John E. West. Scheduling jobs on parallel systems using a relaxed backfill strategy. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 88–102. Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.

[36] Y. Wiseman and D. Feitelson. Paired gang scheduling. Technical Report 2001-10, Hebrew University, 2001.

[37] L. Xiao, S. Chen, and X. Zhang. Dynamic cluster resource allocations for jobs with known and unknown memory demands. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):223–240, March 2002.

[38] L. Xiao, S. Chen, and X. Zhang. Adaptive memory allocations in clusters to handle unexpectedly large data-intensive jobs. *IEEE Transactions on Parallel and Distributed Systems*, 15(6), June 2004.

[39] Li Xiao, Xiaodong Zhang, and Stefan A. Kubricht. Incorporating job migration and network RAM to share cluster memory resources. In *HPDC*, pages 71–78, 2000.

[40] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 133–158. Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[41] Bing Bing Zhou, David Welsh, and Richard P. Brent. Resource allocation schemes for gang scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 74–86. Springer Verlag, 2000. Lect. Notes Comput. Sci. vol. 1911.


MICHIGAN STATE UNIVERSITY LIBRARIES
3 1293 02511 0267