2004

Dynamic adaptation to CPU and memory load in scientific applications

Richard Tran Mills
College of William & Mary - Arts & Sciences

Recommended Citation

Mills, Richard Tran, "Dynamic adaptation to CPU and memory load in scientific applications" (2004). *Dissertations, Theses, and Masters Projects*. Paper 1539623457.
https://dx.doi.org/doi:10.21220/s2-my1g-nn08

DYNAMIC ADAPTATION TO CPU AND MEMORY LOAD IN SCIENTIFIC APPLICATIONS

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

by

Richard Tran Mills

2004

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

Richard Tran Mills

Richard Tran Mills

Approved by the committee, November 2004

[illegible]

Andreas Stathopoulos, Chair

[illegible]

Dimitrios Nikolopoulos

[illegible]

Virginia Torczon

[illegible]

Robert Voigt

[illegible]

Richard Wolski

Department of Computer Science

University of California, Santa Barbara

To my family—especially my mother and father—and the Commodore-64 personal computer.

Table of Contents

ACKNOWLEDGMENTS

This work reflects the influence of a great many people on my life, and I regret that it is not possible to name all of them here; all of them deserve thanks.

I wish to thank all of the members of my dissertation committee for their insightful comments and guidance, which have improved this work tremendously. Foremost among them I thank my advisor, Andreas Stathopoulos, for teaching me a great deal about scientific computing and for taking an active and enthusiastic role in guiding my research. This work reflects a great deal of effort on his part. I thank Dimitris Nikolopoulos for serving as almost a second advisor during the memory-related portions of this work. Robert Voigt deserves special thanks not only for his comments on this work, but also for valuable guidance in my career and professional development. He has always being willing to listen to my problems when I encountered bumps on my road through graduate school. I thank Virginia Torczon for her helpful comments, and for teaching me some of the subtleties of scientific computing when I was the grader for her "Scientific Computing" course. Rich Wolski helped motivate this work with his research on Grid middleware, and deserves special thanks for agreeing to serve on my committee without knowing anything about me.

Outside of my committee, other people at William & Mary deserve thanks. Evgenia Smirni taught me me the basics of parallel computing and collaborated on the early stages of the load balancing portion of this work. Xiaodong Zhang impressed upon me the importance of the interplay between algorithms and architecture in his excellent "Advanced Computer Architecture" course. Shiwei Zhang of the Department of Physics exposed me to many areas of computational science—of which I might otherwise have remained ignorant—in his excellent course on computational physics. Tom Crockett kept SciClone up and running and was always ready to resolve technical issues with skill and enthusiasm; additionally, he provided much-needed friendly conversation on the upper floor of Savage House. Jim McCombs provided the Jacobi-Davidson code used in this work, collaborated on the load-balancing work, and was an amicable office mate.

I spent the summer of 2003 working at Los Alamos National Laboratory with Peter Lichtner. Although the work that I did there was unrelated to my dissertation research, the experience was important because it rekindled my enthusiasm for research after several years in graduate school had caused it to wane. I thank Peter for serving as a mentor without parallel during an incredibly intellectually stimulating summer.

I thank Uncle Maury for stimulating my interest in physics in general and computational physics in particular; that led me down the road to becoming a computational scientist.

I thank my wife, Saffron, for her companionship, love, and support though some difficult times as we suffered through our graduate studies together. The years that we have spent in graduate school have been very trying, but thanks to her constant companionship they have also been the best ones of my life.

Lastly, but most importantly, my parents deserve special thanks for all that they have done. Besides being wonderful, loving parents, they impressed the importance of knowledge and learning upon me at a young age, and their gentle encouragement is responsible for all of my academic (and other) success. I thank my mother especially for the countless hours she spent teaching me reading, arithmetic, and so on when I was a young child; I am very fortunate to have had such a patient teacher of these fundamentals. My father deserves credit for teaching me to think rationally and scientifically, and to view the world with curiosity. It is his example that inspired me to pursue a career in the sciences. He has served as my first and closest mentor, answering my endless questions when I was four years old, reading great books to me and my sister, encouraging my ambitious science projects in primary and secondary school, helping me select courses to take as an undergraduate, collaborating with me on my senior thesis work, and helping me through some tough times in graduate school. His influence on my thinking has been inestimable.

This work was supported in part by a Department of Energy Computational Science Graduate Fellowship, provided under grant number DE-FG02-97ER25308 and administered by the wonderful folks at Krell Institute. Some of the work was performed using the Sci-Clone computational cluster at the College of William & Mary, which was enabled by grants from Sun Microsystems, the National Science Foundation, and Virginia's Commonwealth Technology Research Fund.

List of Tables

List of Figures

ABSTRACT

As commodity computers and networking technologies have become faster and more affordable, fairly capable machines have become nearly ubiquitous while the effective "distance" between them has decreased as network connectivity and capacity has multiplied. There is considerable interest in developing means to readily access such vast amounts of computing power to solve scientific problems, but the complexity of these modern computing environments pose problems for conventional computer codes designed to run on a static, homogeneous set of resources. One source of problems is the heterogeneity that is naturally present in these settings. More problematic is the competition that arises between programs for shared resources in these semi-autonomous environments. Fluctuations in the availability of CPU, memory, and other resources can cripple application performance. Contention for CPU time between jobs may introduce significant load imbalance in parallel applications. Contention for limited memory resources may cause even more severe performance problems: If the requirements of the processes running on a compute node exceed the amount of memory present, thrashing may increase execution times by an order of magnitude or more.

Our goal is to develop techniques that enable scientific applications to achieve good performance in non-dedicated environments by monitoring system conditions and adapting their behavior accordingly. In this work, we focus on two important shared resources, CPU and memory, and pursue our goal on two distinct but complementary fronts: First, we present some simple algorithmic modifications that can significantly improve load balance in a class of iterative methods that form the computational core of many scientific and engineering applications. Second, we introduce a portable framework for enabling scientific applications to dynamically adapt their memory usage according to current availability of main memory. An application-specific caching policy is used to keep as much of the data set as possible in main memory, while the remainder of the data are accessed in an out-of-core fashion. This allows a graceful degradation of performance as memory becomes scarce.

We have developed modular code libraries to facilitate implementation of our techniques, and have deployed them in a variety of scientific application kernels. Experimental evaluation of their performance indicates that our techniques provide some important classes of scientific applications with robust and low-overhead means for mitigating the effects of fluctuations in CPU and memory availability.

DYNAMIC ADAPTATION TO CPU AND MEMORY LOAD IN SCIENTIFIC APPLICATIONS

Chapter 1

Introduction

The advent of powerful but inexpensive workstations and fast, affordable networking technologies has fundamentally altered the landscape of parallel and distributed computing. Powerful commodity components have led to the emergence of clusters of workstations (COWs) as the primary means of parallel computing at many institutions. On a much grander scale, researchers have been taking steps towards geographically distributed computing, building computational "Grids" that link computing resources between different institutions. Grid environments promise to provide the computational resources necessary for analyzing petabyte-scale data sets and conducting the most computationally demanding of simulations. The impact of these technological developments has been particularly felt in scientific applications, which form a great portion of the high-performance computing workload.

The category of "scientific applications" is large and evolving, and therefore difficult to characterize accurately. We make, however, the following broad observations about scientific applications in parallel and high-performance computing (HPC) environments: First, the applications often work with large quantities of data, because realistic simulation of a problem often requires a large number of degrees of freedom. Second, they tend to follow fairly regular, repetitive data access and communication patterns. Third, they usually have synchronization points at which collective communications occur.

Many modern parallel computing environments possess a degree of complexity that poses problems for conventional scientific applications designed to run on dedicated, uniform resources such as massively-parallel processor (MPP) machines. One source of problems is the heterogeneity that arises naturally in these new settings. For instance, upgrades to a COW may occur in stages, with only specific machines being replaced or added. In other cases, the cluster may simply be a loosely coupled network of personal workstations. In a computational Grid, disparate resources from different institutions may be used in conjunction. Traditional parallel programs attempt to distribute work evenly between all nodes because they target environments in which compute nodes are homogeneous in processing power and memory capacity. When nodes differ considerably in processing speed, such a strategy can result in severe load imbalance as faster processors idle at synchronization points, waiting for their slower brethren to finish. If nodes differ considerably in memory capacity, nodes with higher memory capacities may be underutilized.

Beyond architectural heterogeneity, competition for non-dedicated, shared resources poses even greater challenges. Various economic and practical considerations have made multiprogramming of resources commonplace: In decentralized systems, such as Grid environments or loose networks of personal workstations, space sharing simply may not be a viable option. Owners of smaller COWs may not be willing to sacrifice the time necessary to administer a batch queuing system, or to tolerate the slower turnaround times resulting from waiting in the queue, especially during the development and debugging cycles. In shared memory machines, even when CPUs are space-shared, jobs still compete for memory and I/O resources.

Fluctuations in the availability of CPU, memory, and other resources can cripple application performance. Contention for CPU time between jobs may introduce significant load imbalance in parallel applications. Contention for limited memory resources may cause even more severe performance problems: If the requirements of the processes running on a compute node exceed the amount of memory present, thrashing may increase execution times by an order of magnitude or more.

It is our thesis that the dynamic and unpredictable nature of resource contention in modern parallel environments necessitates a degree of system awareness and adaptivity that is not present in today's software. Our goal is to develop techniques that allow scientific applications to achieve good performance in non-dedicated environments by monitoring system conditions and adapting their behavior accordingly. In this work, we focus on two important shared resources, CPU and memory, and pursue our goal on two distinct but complementary fronts:

1. *Application-level dynamic load balancing.* We present some simple algorithmic modifications which can significantly improve load balance in a class of iterative methods. We focus on these methods because they form the computational core of many scientific and engineering applications; improvements to the methods can lead to immediate performance gains in a variety of scientific codes.

2. *Memory adaptation in scientific and data-intensive codes.* We introduce a portable framework for enabling applications to dynamically adapt their memory usage according to current availability of main memory. An application-specific caching policy is used to keep as much of the data set as possible in main memory, while the remainder of the data are accessed in an out-of-core fashion. This allows a graceful degradation of performance as memory becomes scarce.

The combination of system-aware mechanisms for dynamic load balancing and memory adaptation has the potential to change the way computational science is done by enabling efficient use of shared resources that have hitherto been underutilized. We provide some specific tools and techniques for implementing these mechanisms in some specific cases, but our broader goal is to establish an archetype for building system-aware, adaptive applications.

1.1 Contributions

The contributions of this work can be outlined as follows:

1.1.1 Dynamic load balancing of inner-outer iterative solvers

We have studied a novel strategy for dynamic load balancing of a class of inner-outer iterative methods. Our goal has been to enable parallel, iterative, sparse matrix methods—which form the core of many scientific applications—to perform well in heterogeneous and multiprogrammed environments. Our techniques are designed for inner-outer iterative solvers that employ local solvers for the inner iterations, though they are also applicable to other algorithms that possess what we term a *flexible phase* (defined in Chapter 3). We have written a C-library to facilitate use of our load balancing technique, and have used it to implement our scheme in some different solver codes.

We have experimentally demonstrated that our scheme can cope with severe load imbalance under unpredictable external loads in a coarse-grained, block Jacobi-Davidson eigensolver that independently calculates a different correction vector on each processor. This work has been published in [63]. We have also used our load-balancing scheme to enable effective use of a collection of heterogeneous sub-clusters of workstations in a version of the Jacobi-Davidson solver that fully employs the so-called *multigrain* parallelism; see our paper [57]. Finally, we have demonstrated that our method can be used in an additive-Schwarz preconditioned linear system solver to smooth load imbalances due to differences in conditioning of the subdomains, and also to smooth out small load imbalances due to differences in processor speeds.

1.1.2 Dynamic memory adaptation in scientific applications

We have explored a general framework that enables scientific and other data-intensive codes that repetitively access large data sets to dynamically adapt their memory requirements as memory pressure arises. We have developed a modular supporting library, MMLIB, that makes it easy to embed memory-adaptivity in many scientific kernels with little coding effort. The biggest technical hurdle we had to overcome to realize our framework is the lack of information about memory availability provided by operating systems. We have developed an algorithm that can effectively judge memory availability with minimal reliance on operating system-provided information.

We have used our library to inject memory-adaptivity into three scientific application kernels: a conjugate-gradient linear system solver, a modified Gram-Schmidt orthogonalization routine (surrounded by a GMRES-like wrapper), and a Monte-Carlo Ising model simulation. Though simple, these kernels are representative elements of scientific simulations spanning a diverse range of fields. Experimental evaluation of these memory-adaptive kernels in Linux and Solaris environments has shown that our techniques yield performance several times better than that obtained by relying on the virtual memory system to handle memory pressure. Furthermore, we have demonstrated that these performance gains are not only due to enabling use of application-specific replacement policies, but from avoidance of virtual memory system overheads and antagonism. We have also shown that multiple jobs employing our memory-adaptation techniques can coexist on a node without ill-effects. Our work is presented in [61] and [62].

1.2 Relevance

This work is relevant to both computer scientists and practitioners of computational science. From a computer science perspective, it is significant because it explores adaptation to fluctuating system conditions completely at the application-level; traditionally, adaptation approaches have been more system-oriented. An application-centric approach allows us to exploit knowledge about the application that system software does not possess.

In our work on load balancing parallel computations under unpredictable external loads, we utilize knowledge of the numerics of an important class of iterative methods to dynamically modify the computation—without affecting the end result—to achieve good load balance with very little overhead. This departs from conventional load balancing approaches, which adaptively schedule a computation but do not modify it.

In our work on dynamic adaptation to high levels of memory pressure, we use knowledge of the memory-access characteristics of an application to manage memory using an appropriate replacement policy and to use units of data transfer suited to the granularity of memory access. Our memory adaptation work is additionally significant because it has explored the interplay of system and application in multiprogrammed systems under extreme levels of memory demand. System and application behavior under such conditions can be extremely unpredictable and has been relatively unexplored; application-specific virtual memory management schemes, for instance, have usually been designed without multiprogramming in mind.

From the perspective of computational science practitioners, this work is relevant because it has the potential to increase their productivity. Many researchers do not have access to dedicated supercomputers or high-end clusters and must meet their computing needs by using either local shared resources such as networks of workstations and small SMPs, or resources shared across a computational grid. The methods we present can enable some important classes of scientific applications to effectively use such resources. These techniques can also improve the productivity of researchers who do have access to dedicated supercomputers by allowing them to run small to moderate-sized jobs on readily available non-dedicated resources, allowing them to conserve their supercomputer allocations and avoid long wait times in compute-center queues.

1.3 Organization

The remainder of this document is organized as follows: Chapter 2 relates our work to the current state of knowledge in both load balancing and mechanisms for dealing with memory shortage. Chapter 3 explains our load balancing scheme and presents experimental results demonstrating its effectiveness in balancing CPU load. In chapter 4, we illustrate how the load balancing scheme can be extended to attenuate memory pressure, and we outline the shortcomings of this approach. In chapter 5, we present the essential details of our memory adaptation framework, and in chapter 6 we provide an overview of our software library, MMLIB, that facilitates use of the framework. In chapter 7 we present experimental evaluations of the memory adaptation strategy as used in three scientific application kernels. Finally, conclusions are presented in chapter 8.

Chapter 2

Background and related work

The goal of our work is to allow some classes of scientific applications to perform well in multiprogrammed environments where CPU and memory availability can vary unpredictably. To meet this goal our techniques must maintain good load balance as external CPU loads fluctuate, and adjust memory utilization gracefully as memory availability changes. In this chapter, we present a brief overview of related work in load balancing and mechanisms for addressing memory shortage. Additionally, we describe related work on application-centric adaptive scheduling techniques that dynamically adapt schedules according to resource availability.

2.1 Load balancing

Maintaining good load balance is essential to ensure good performance and efficient utilization of processors by a parallel application. Otherwise, CPU cycles will be wasted as lightly loaded processors sit idle waiting for their more heavily loaded brethren to finish. A significant amount of research has focused on the development of methods and software for load balancing within parallel applications, and we outline some of those approaches in this section.

Load balancing is essentially a task-mapping problem, and is thus inextricably linked with the manner in which a computation is decomposed into tasks for parallel execution. The problem is to map a number of tasks onto a set of processors in a way that ensures that each processor is assigned an amount of work commensurate with its speed. This mapping usually must satisfy other objectives as well, such as limiting the amount of communication required. There are many different strategies for accomplishing these goals, but we can place them into two broad categories based on whether they take a task-oriented or a data-oriented approach to mapping the tasks onto processors. The type of approach that is appropriate is determined by the amount of data required to describe and perform each task, and the cost of moving that data to the appropriate processor. (So the choice between task or data orientation depends on both the properties of the problem to be solved and the properties of the target parallel machine.) If processors can inexpensively obtain the data required to perform a task, a task-oriented approach can be used. Otherwise, a data-oriented approach must be employed because the cost of moving the data for a task is the main limiting factor.

2.1.1 Pools of independent tasks

The task-oriented approach is often known as the *pool of tasks* paradigm. The problem to be solved is broken into a pool of independent tasks. Processors that need work fetch tasks from the pool, and if additional tasks are generated during the computation, they are placed into the pool. If the size of the tasks is small enough, the pool of tasks approach usually results in excellent load balance. (Tasks that are too large may result in one processor continuing to work while others sit idle because all other tasks have been processed.) In the simplest implementation (master/slave or manager/worker), processors fetch or receive tasks from a single processor that acts as centralized manager of the pool. In addition to being very easy to implement, the master/slave approach maintains global knowledge of the computation state, which facilitates even distribution of tasks and greatly simplifies termination detection. The master/slave approach cannot scale to large numbers of processors, however, because communication with the master becomes a bottleneck.

To improve scalability, a hierarchical approach can be employed, in which workers are divided into disjoint sets managed by a local sub-master that reports to the central master. If greater scalability is needed, a completely decentralized scheme can be used, making the pool of tasks a distributed data structure. In a decentralized scheme, when a processor runs out of work it sends a potential donor processor a work request. The donor processor to be polled may be chosen in a number of ways: The simplest way is to choose a donor randomly, with all processors having equal probability of being selected. Alternatively, donors can be chosen in a round-robin fashion. The round-robin scheme can employ a global list of processors, or might be restricted to the nearest neighbors of each processor in the network interconnect. Note that if a round-robin scheme is completely decentralized, a donor may receive work requests from multiple processors simultaneously. To prevent this, a hybrid centralized/distributed system may be employed. In such a scheme, the task pool is a distributed data structure, but one processor maintains a global variable that points to the processor to which the next work request should be sent. This improves the distribution of work requests, though contention for access to the global variable somewhat limits scalability. Explanations and analysis of several decentralized pool of tasks schemes and appropriate termination-detection algorithms can be found in [70, 52, 51].

If a computation is decomposed into a large number of small tasks, fetching only one piece of work at a time can result in excessive scheduling overhead. In such situations, an allocation method in which a free processor fetches a chunk of several tasks at a time is usually employed. This reduces scheduling overhead, but care must be taken not to make the chunks too large, because the coarser task granularity may result in increased load imbalance. Kruskal and Weiss [50] used probabilistic analysis to determine an optimal chunk size that balances the trade-offs between very large and very small chunk sizes. Their analysis relies on the assumption of a number of tasks large enough to cancel out variations in the task execution times. Because in practice this assumption often does not hold, other developments in chunk-scheduling employ schemes that start with larger chunk sizes and then gradually decrease the chunk sizes as the computation progresses to prevent processors from finishing at different times. Such approaches include guided self-scheduling [68], factoring [39], and trapezoidal self-scheduling [85].

The pool of tasks approach can maintain excellent load balance, even in the face of factors such as large variation in the cost of each task, processor heterogeneity, or dynamic variation in external processor load on a time-shared system. Unfortunately, the approach is inapplicable to many scientific problems, since they cannot be broken into independent tasks that can be executed asynchronously. Additionally, the independent tasks often cannot be described with an amount of information that is small enough to prevent excessive communication between master and slave processes.

2.1.2 Data partitioning

If the tasks in a computation cannot be described with a small amount of information, or if dependencies between tasks necessitate frequent inter-processor communication, a data partitioning approach to load balancing should be employed: Rather than thinking in terms of mapping tasks to processors, we map the data objects in the computation (e.g. mesh points or particles) onto processors. Processors then execute those tasks associated with their local data. To load balance a computation, we partition the objects into approximately equal partitions (ensuring load balance) while minimizing the data dependencies between partitions (thus reducing communication costs). If processors vary in speed, the partition associated with a given processor is weighted by its relative speed. What we have called data partitioning is usually referred to as domain partitioning, because it is usually employed to assign portions of a computational domain to processors. Some of the techniques we describe here can be applied to any abstract object that can be modeled as a graph, so we use the more general term "data partitioning".

The partitioning approaches can be grouped into two broad categories: geometric and topological. A geometric approach divides a domain based on the locations of objects in a simulation, and is therefore appropriate for applications such as solid mechanics or particle simulations, where interactions between computational objects tend to occur between neighbors. A topological method explicitly bases its partitioning on the connectivity of objects in a computation. Typically a computational domain is represented as a graph which is to be equipartitioned in a way that minimizes the edge-cut, which approximates the amount of communication required.

2.1.2.1 Geometric partitioning

The geometric approaches can be classified into two major types: recursive bisection approaches and octree approaches. These categories could also be described respectively as approaches that work with coarse-grained or fine-grained divisions of the geometry. The basic idea of recursive bisection approaches is to cut a domain into two parts such that each part contains an approximately equal number of objects. Each part is then further cut into two smaller parts, and so on, until the required number of partitions have been generated. The simplest such method is Recursive Coordinate Bisection (RCB) [10], also known as Coordinate Nested Dissection (CND). RCB recursively divides the set of objects using a cutting plane orthogonal to the coordinate axis along which the objects are most spread out. The cut is orthogonal to the long axis to reduce the size of the boundaries between subdomains, across which communication will have to occur. RCB is simple, fast, easily parallelizable, and requires little memory, but also tends to produce lower-quality partitionings. A modest improvement is Unbalanced Recursive Bisection (URB) [42], which divides the geometry, rather than the objects, in half. This minimizes the geometric aspect ratio of the partition and thus reduces communication volume, as the amount of information that must be exchanged across boundaries is approximately proportional to their length or area. The effectiveness of both RCB and URB are limited by the somewhat artificial restriction that cuts are always made along coordinate axes. Recursive Inertial Bisection (RIB) [83] eliminates this restriction by calculating the principal axes of inertia of the collection of objects as if they were a collection of particles in a physical system. The domain is then cut by a bisecting plane orthogonal to the principal axis associated with the minimum moment of inertia, which is a naturally "long" direction across which to cut. Geometric bisection methods need not be limited to the use of cutting planes. Based on some interesting theoretical results [34], Gilbert, Miller, and Teng [32] describe a geometric bisection scheme in which circles or spheres are used to bisect domains.

Octree techniques (also known as space-filling curve techniques) for geometric partitioning start with a fine-grained division of the computational domain, and then aggregate these fine-grained pieces into sets that will make up the partitions. The fine-grained division is obtained by recursively dividing a space into a hierarchy of octants. For a 3D domain, the root octant is a cube that encloses the entire domain. This octant is divided into child octants by simultaneously cutting the root octant in half along each axis (forming eight child octants in 3D space). Child octants that contain multiple objects are then recursively divided into sub-octants, and so on, until each object is enclosed by a terminal octant (i.e., one that is not further subdivided). The relationship of octants in the hierarchy is described by a tree data structure known as an *octree*. Octrees are used to represent space in mesh generation, computer graphics, and n-body simulations, among other applications. They also provide a convenient way to partition geometry among processors: By ordering the terminal octants according to their positions along a space-filling curve such as a Peano-Hilbert curve [55], a k-way partitioning can easily be generated by cutting the list into k equal parts [88]. The speed and quality of octree partitioning is roughly equal to that of geometric recursive bisection techniques.

2.1.2.2 Topological partitioning

Geometric partitionings can be calculated very quickly and can be quite effective for computations in which interaction between objects is a function of geometric proximity, such as in n-body or solid mechanics problems. If geometric proximity is a poor indicator of interaction between objects, however, using a more expensive topological method that explicitly considers the connectivity of the objects can yield far superior partitionings.

One of the simplest topological methods is Levelized Nested Dissection (LND) [31]. LND starts with an initial vertex (a pseudo-peripheral vertex is the best choice), and then visits neighboring vertices in a breadth-first manner. When half of the vertices have been visited, the graph is bisected into the set of vertices that have been visited, and those that have not. This process is applied recursively until the desired number of partitions have been created. The idea behind LND is actually quite similar to geometric bisection methods: distance from the initial vertex is used to cut across a "long" dimension of the graph.

Spectral methods [69, 78] use a very different approach for computing partitionings. The partitioning problem is really a discrete optimization problem: minimize the edge-cut while dividing the graph into k equal parts. This discrete optimization problem is NP-complete, but the problem can be approximated by a continuous optimization problem that is more easily solved. Spectral methods solve the continuous problem by finding some extremal eigenvalue/eigenvector pairs of a matrix derived from the connectivity information of the graph. Spectral methods tend to produce high quality partitions, but solving the eigenproblem can be computationally expensive.

Partition refinement methods refine a sub-optimal partitioning of a graph, and are often used as graph partitioners themselves by applying them to a random partitioning. Kernighan and Lin [48] developed an iterative algorithm (known as the KL algorithm) that improves a graph bisection using a greedy approach to determine pairs of vertices to swap between partitions. Variations of the KL algorithm are used for refinement in several graph partitioning contexts, some of which we discuss below.

Some of the most popular graph partitioning methods being used today are multilevel methods [37, 45]. Multilevel methods form a sequence of increasingly coarse approximations of the original graph by collapsing together selected vertices from finer representations of the graph. The graph is initially partitioned at the coarsest level using a single-level algorithm such as those described above. This partitioning is then propagated back through the finer approximations, being refined at each level via a KL-type algorithm. Multilevel approaches are very effective for two reasons. First, the initial partitioning can be easily computed for the coarsest graph, because it hides many edges and vertices of the original graph. Second, incremental refinement algorithms can quickly make large changes to the partitioning by working with coarser versions of the graph. Chaco [36], METIS/ParMETIS [43, 44], and JOSTLE [87] are some popular software packages that implement multilevel methods.

2.1.2.3 Load balancing adaptive computations

If external load is not a factor, and if the amount of work per object is known and does not vary during the course of the computation, a static partitioning can be used. The partitioning can be computed as a sequential preprocessing step at the beginning of a simulation, using a package such as METIS or Chaco, or in parallel using a package such as ParMETIS or JOSTLE. In many computations, however, the amount of work assigned to each processor can vary unpredictably, either because the number of objects in a partition can vary, or because the amount of work per object is unknown. For example, if the objects are grid points in an adaptive mesh refinement (AMR) calculation [12, 11], the amount of work for a partition may grow as the mesh is refined. If objects are cells in a particle-in-cell simulation, the amount of work associated with an object varies as particles advect into and out of the cells. Some simulations utilize adaptive physics, in which the physics model used at a mesh point—and thus the work associated with that point—may change as the calculation progresses. In all of these cases, the computational domain must be repartitioned as the computation progresses, because a partitioning that results in good load balance at the beginning of the computation may yield very poor load balance later.

Dynamic repartitioning can be done by simply computing a new global partitioning from scratch. However, this generally incurs an unacceptable amount of costly data migration, because the new partitioning can be very different from the existing one. Instead, the old partitioning should be adjusted in a way that mitigates load imbalance, while also minimizing the amount of data migration needed to arrive at the new partitioning. Scratch-remap repartitioners [66] attempt to do this by calculating a new partitioning of the graph from scratch, and then permuting the labels of the new partitions so their overlap with the old partitions is maximized.

Scratch-remap approaches result in excellent load balance and low edge-cut, but still involve a relatively high amount of data movement. For this reason, incremental approaches that make a series of small changes to the partitions to restore load balance are usually favored instead. Most of these methods employ a diffusive approach, in which objects move from heavily loaded processors to more lightly loaded neighbors by a diffusion-like process. The number of objects to be moved from a processor is based on the processor workload, while the choice of which objects get moved is usually made using KL-like criteria. In the simplest incarnations [21], a first-order finite difference discretization of the diffusion equation is used to calculate the amount of work that should be moved between processors at each iteration of the method. Because the first-order stencil is compact (only involving nearest neighbors), such methods can be executed locally by a processor. These methods can be slow to converge to a load balanced state, however, so more complex methods use some global information to accelerate convergence to load balance. The method of Hu, Blake, and Emerson [38] (implemented in ParMETIS and JOSTLE) uses information from all processors to perform an implicit solve for the steady-state solution of the diffusion equation, and then migrates data in one step to reach this state of load balance.

Although local methods are usually slower than global methods to achieve load balance, they can be executed asynchronously, whereas global methods cannot. This is a significant advantage for two reasons. First, load balance can be corrected as it arises, without having to wait for the next synchronization. Second, although most scientific applications have natural synchronization points, certain applications [28, Chapter 14] do not, so the use of a global method introduces artificial (and costly) synchronizations.

Parallel partitioning packages such as ParMETIS or JOSTLE provide algorithms for computing incremental repartitionings, but once the new partitioning is obtained, performing the actual data migration can be a difficult chore for the application programmer. Packages such as Zoltan [22] and PREMA [7, 8] provide tools and frameworks for managing and automatically migrating the objects in adaptive computations. PREMA uses multithreading to support asynchronous load balancing, and thus is particularly suited to very irregular computations.

2.1.3 Load balancing under variable external load

In our discussion so far, we have only considered the load that is internal to an application. In a multiprogrammed environment, however, the total load on a processor can vary unpredictably due to load from processors external to a given application.

Provided that the granularity of the tasks is not too coarse, the pool of tasks paradigm can work well (and automatically) in the presence of external load. Data partitioning approaches, however, face major difficulties in dealing with such loads. External load can conceivably be accounted for by weighting each processor according to its load and then sizing the partitions based on those weights. However, the high cost of repartitioning combined with the dynamic and unpredictable nature of external workloads makes this impractical. If external load conditions stay fairly constant, the cost of repartitioning to reflect that load can be worthwhile. If the decision to repartition proves wrong—that is, if the external workload changes dramatically soon after partitioning—it can be very costly. Because of the difficulties involved in external load prediction, repartitioning is generally not used to cope with external load.

In Chapter 3 we present a load balancing scheme for a very specific but important class of scientific algorithms, inner-outer iterative methods that employ local inner iterations. These algorithms are not amenable to a pool-of-tasks approach, and—for the reasons discussed above—load balancing by dynamic repartitioning is impractical. We present a scheme that provides good load balance for these algorithms in the face of dynamic and unpredictable load, without any reliance on data migration or performance predictions.

What we call our "load-balancing" scheme differs fundamentally from other load-balancing schemes in the conventional sense of the word. The pool-of-tasks and data partitioning approaches we have described distribute a *fixed* set of tasks in a balanced way across a set of processors. Our load balancing scheme also distributes tasks in a balanced manner across a set of processors, but the set from which these tasks are drawn is not fixed. That is, the set includes *optional* tasks: each task (hopefully) helps move the algorithm to its target, but not all tasks must be completed for this target to be reached. The tasks to execute are chosen to maximize load balance and reach the target as quickly (in wall-clock time) as possible: tasks are associated with data on a particular processor, and slower processors execute fewer of the associated tasks. This flexibility in choosing which tasks to execute stems from the intrinsic properties of the inner-outer iterative methods.

2.2 Addressing memory shortage

Jobs in non-dedicated environments contend not only for CPU time, but for memory resources as well. Memory pressure can cause extreme degradation of performance. Its effects are especially pronounced if an affected node is part of a synchronous parallel job, as the immense slowdown experienced by that node can cause incredible load imbalance. In many cases this imbalance cannot be corrected by load balancing techniques that may be in place within the application. For instance, if the job is running on a shared-memory computer, migrating work to other processors on that machine fixes nothing. Even in distributed memory environments, problems remain: The usually excellent load balance provided by a pool-of-tasks scheme may disappear as the slowdown on an affected node increases the granularity of its tasks to excessive levels. If load balancing is performed through domain repartitioning, memory pressure may render the very cost of computing a new partitioning prohibitive! Even when calculating the new partitioning is practical, deciding how to weight each partition is a difficult problem, as operating systems typically provide very little information about memory usage. For these reasons and others, explicit consideration of memory pressure and its mitigation is needed in non-dedicated environments, in addition to good CPU-load balancing approaches.

In this section we discuss several approaches that have been used to remedy memory pressure in dedicated and non-dedicated environments.

2.2.1 Out-of-core algorithms

Since the earliest days of the computer age, researchers have been faced with problems too large to solve using the main memory of a computer. In response to these problems, many out-of-core algorithms have been developed. An out-of-core algorithm [84, 86, 25, 71] is one that has been designed to provide acceptable performance despite the slow nature of accessing secondary storage. Such an algorithm may be obtained by taking a conventional (in-core) algorithm, implementing it to access large, contiguous blocks of data, and re-scheduling its independent operations in a manner that maximizes the re-use of data that have been brought into main memory. Some out-of-core algorithms go beyond simply rescheduling independent operations, altering conventional algorithms so that their data dependencies are more amenable to data re-use. These alterations may sacrifice some numerical stability or require more CPU operations, but admit much better out-of-core schedules.

Out-of-core algorithms bear many similarities to cache-optimized algorithms, as the goal of both is to maximize use of faster levels of a memory hierarchy by scheduling operations in a manner that provides good spatial and temporal locality. Both types of algorithms benefit from many of the same optimization techniques: Structuring loop nests to eliminate strided accesses is important at all levels of the memory hierarchy. Many dense matrix algorithms are optimized for out-of-core use by structuring them to operate on blocks of a matrix at a time. Such blocking techniques are also used to achieve cache optimality in numerical libraries such as BLAS [53, 24, 23] and LAPACK [2]; they are so important, in fact, that computer vendors make great efforts to precisely tune blocking factors in their implementations of these libraries, and the ATLAS project [89, 90] provides software to automatically tune these factors. Cache-oblivious algorithms [30, 15], such as the one employed by the popular FFTW library [29], operate using a divide-and-conquer approach that splits a problem into subproblems that fit into the cache; this approach eliminates the need to tune hardware-specific parameters. Although these algorithms have been proposed with cache-optimality in mind, these techniques are also beneficial in out-of-core settings [20].

Despite their similarities, there are important differences in the design of out-of-core versus cache-optimized algorithms. These stem from two fundamental differences between the levels of the memory hierarchy that the two types of algorithms target. First, the main-memory/disk bandwidth ratio is usually much higher than the cache/main-memory bandwidth ratio in a computer system. This means that modifications that increase CPU operations but improve out-of-core performance by reducing disk I/O may perform worse than unmodified versions when executed in-core, despite possibly higher cache efficiency. Second, cache memory usually has limited associativity—that is, blocks brought from main memory cannot be placed anywhere in the cache, but must be placed in a limited number of specific locations in the cache. Many cache-optimization techniques are designed to reduce so-called "conflict" misses that stem from limited cache associativity. Because main-memory is fully associative (i.e., blocks from disk may be placed anywhere), such optimizations do not improve out-of-core performance.

Out-of-core algorithms have been in use for decades to allow computers to solve problems too big for their physical memory. These codes have been designed to operate with a fixed amount of memory, however, and are unsuitable to deal with the transient memory shortages encountered on multiprogrammed systems. An out-of-core code could certainly be used to avoid memory pressure, but the utility of such an approach is dubious: Though the algorithm would work efficiently when memory is scarce and an in-core algorithm would thrash, when memory is plentiful the out-of-core algorithm will not take advantage, continuing to use the slow disk instead of the much faster main memory.

2.2.2 Virtual memory system modifications

A virtual memory (VM) system that can transparently handle a high degree of memory pressure is highly desirable, as this could free the application programmer from having to worry about memory shortages. Transparent VM systems fall short, however, because no generic page replacement policy that is appropriate for the memory access patterns of all applications is known. Most systems employ an LRU-like policy. Under such a policy, an application that repeatedly loops through a data set, for example, will incur thrashing even if memory shortage is slight. Recently proposed general-purpose replacement algorithms, such as LRFU [54] and LIRS [40] and the related CLOCK-Pro algorithm [41], eliminate some of the shortcomings of LRU at the expense of introducing some additional complexity. None of these algorithms has yet made its way into a production operating system.

In addition to replacement policy problems, the generic prefetching policy and the fixed page size employed by most VM systems may be inappropriate for the granularity of data access in an application. This is especially true when memory pressure forces pages to be written to swap space, which can be highly fragmented and generally does not support prefetching. The shortcomings of generic VM systems have led many researchers to propose operating system modifications that expose portions of the VM subsystem to applications. The Mach microkernel [1] provided an external pager interface that allowed applications to control the movement of pages between main memory and secondary storage. It did not export control over page replacement policy, however. McNamee and Armstrong [60] extended Mach's external pager interface to allow application programmers to control page replacement. Harty and Cheriton [35] designed and implemented an external page-cache management scheme in the V++ kernel. In their scheme, the VM system provides a cache of physical pages to an application, and it is the responsibility of the application to manage that physical memory. Krueger et al. [49] devised a similar scheme, but gave it a modular, object-oriented design to allow programmers to use application-defined replacement policies without having to re-implement large portions of the VM system. Cao et al. [18] implemented a system to give applications control over file caching, prefetching and disk scheduling.

Although application-specific virtual memory management schemes are useful in some situations, they suffer some shortcomings. These schemes have generally not been designed with multiprogrammed environments in mind, and assume a fixed amount of physical pages will be available to the process at runtime. Additionally, such schemes are not useful to the typical computer user precisely because they require significant modifications to an operating system kernel. This precludes their use in most computing settings because the operating system does not support such a scheme and cannot be modified to do so. Even if such a system is available to a code developer, writing an application to use such a system incurs severe portability penalties. Memory pressure solutions that do not require OS modification are more desirable to the typical application programmer.

Nikolopoulos [64] developed a runtime system that transparently adapts the memory footprint of running processes as memory availability fluctuates. This solution requires no modification of the operating system and is targeted specifically at multiprogrammed environments. This approach bears many similarities to the memory-adaptation framework we present in Chapter 5, although our framework is even more application-centric: memory adaptation decisions are based solely on measurements local to the application, and we use application-specific replacement policies and units of data transfer.

2.2.3 Memory-adaptive algorithms

Not a great deal of literature on memory-adaptive algorithms exists. Most of it comes from the field of database management, in which dealing with huge data sets is commonplace. Pang et al. [67] conducted one of the earliest memory adaptation studies, investigating external sorting. External (viz., out-of-core) sorts are frequently used in database systems to order query results and to perform certain join operations. An external sort is conducted in two phases, a "split" phase and a "merge" phase. In the split phase, an in-core algorithm is used to arrange unsorted data into a set of sorted runs. In the merge phase, the runs are merged into a single sorted run. Pang et al. discussed how the algorithms used for the split phase could adapt to memory shortage by writing out a run of sorted tuples to reduce buffer usage. To respond to a memory increase, more relations should be brought into the buffer for sorting. They also introduced an adaptation strategy for the merge phase, which they termed *dynamic splitting*: If memory shortage arises during a merge, it should be split into smaller steps that can be performed with the available memory. If there is a memory surplus, then more merges are run at once. Pang et al. did not implement any of their proposed strategies, however, choosing to only utilize simulation in their study. A later study by Zhang and Larson [92] also addressed memory adaptation for external mergesort; it included experimental evaluation of their strategy. Their implementation was designed only to address memory pressure arising from contention between concurrent sorts. Because competing sorts all obtain memory through the same buffer manager, determining the amount of available memory is not a problem. In a general-purpose time-shared system, however, one of the chief difficulties in implementing memory-adaptive algorithms is devising an effective way to determine the amount of available memory.

Barve and Vitter [9] have developed a theoretical framework for proving lower bounds on the resource consumption of memory-adaptive algorithms. Their motivation comes from database management systems, but they also prove bounds for standard matrix-matrix multiplication, fast Fourier transform computation, and LU factorization. Their model assumes that a memory-adaptive algorithm is aware of the amount of memory available to it at all times. Although operating systems do not provide such information, in Chapter 5 we present an algorithm that uses a "gray-box" approach [4] to closely approximate this information with minimal reliance on system-provided information.

2.3 Application-centric adaptive scheduling

Schedulers of parallel programs in non-dedicated environments must consider current and future system loads in an attempt to match a job with a set of resources that will meet its requirements. This is a remarkably difficult task, however, and conventional, system-oriented schedulers usually yield poor results in such settings. In response to the limitations of such systems, newer, more application centric approaches have emerged. Though very diverse, these new approaches follow a common thread of interaction between systems and applications to dynamically adapt their schedules according to resource availability.

Polychronopoulos and Nikolopoulos [65] have presented extensions to the kernel scheduler combined with a user-level scheduling module that coordinates the scheduling of communicating threads of parallel jobs and also prevents thrashing by suspending threads at memory-allocation points if memory becomes overcommitted. Such a scheduler could also be used with memory-adaptive applications (such as those presented in Chapter 7) that could modify their memory requirements instead of suspending when an allocation request cannot be met.

Projects such as AppLeS [13, 14] and Active Harmony [46] present complete frameworks for matching applications to appropriate resources and updating application schedules as resource availability changes during execution. **AppLeS** (**App**lication **Le**vel **S**chedulers) use a completely application-level methodology. An AppLeS-enabled application has its own, specific application-level scheduler that discovers and identifies viable resource sets, determines candidate schedules, selects a schedule that best satisfies the user's performance criteria, and executes the application using that schedule. For long-running applications, the AppLeS agent may refine and re-deploy the schedule as resource availability changes. AppLeS does not provide a custom runtime system; instead, it relies on the existing infrastructure (e.g. Globus [27], NetSolve/GridSolve [3], MPI [79]) for discovering and utilizing compute resources, along with the Network Weather Service [91] to gauge current and predict future resource utilization. Because the AppLeS methodology is entirely applicationdriven, it is especially suitable to non-centrally controlled computing environments.

Like AppLeS, Active Harmony matches applications to appropriate resources and reschedules them as resource availability fluctuates. In Active Harmony, however, decisions about application scheduling are made by a centralized resource manager rather than the applications themselves. Centralized decision making is employed in an effort to maximize overall efficiency and throughput. Active Harmony provides a "tuning interface" through which applications export information about possible application configurations. A manager then dynamically tailors the application configuration according to resource availability. For example, a hybrid client-server database that can process queries on either the client or server side could be configured to process queries at the server when there are few clients, but be reconfigured to process queries on client nodes when many clients are using the system.

Systems like AppLeS or Active Harmony can place quite a burden on the user, as often very detailed resource-utilization information must be provided. For instance, Active Harmony requires that the user specify the resource requirements of different options, the granularity at which configuration changes can be performed, the cost of switching between the configurations, and (optionally) a way in which the response time of a given configuration can be calculated. Chang and Karamcheti [19] have developed a resource management framework that eases the burden on the user by providing a virtual execution environment which permits automatic, off-line generation of resource-utilization information.

The adaptive scheduling approaches we have just cited and the techniques for dynamic adaptation to CPU and memory load presented in this dissertation are complementary. Systems such as AppLeS and Active Harmony attempt to identify the "best" schedule for an application, and to refine that schedule as resource availability changes. Our techniques, on the other hand, provide applications with a degree of malleability that enables them to cope with a set of resources that may at times be far from "best" due to fluctuations in resource utilization. Using a system such as AppLeS to select the optimal schedule, and techniques such as ours to optimize the use of the scheduled resources makes sense. This seems especially true when one considers the overheads involved in re-scheduling onto a different set of resources. Applications that can achieve acceptable performance on less-than-ideal resource sets can reduce the need for re-scheduling and thus avoid some of the associated costs.

Chapter 3

Load balancing flexible-phase iterations

In the first phase of this work, we have investigated a novel dynamic load-balancing technique for parallel inner-outer iterative methods that execute inner iterations locally on each processor. Our approach takes advantage of what we term the *flexibility* of the inner phase, using the fact that the amount of work done by the processors during the inner phase can vary (within bounds) without detriment to the end result: we limit all processors to the same time in the inner phase (which performs the bulk of the work), virtually eliminating load imbalance there. We explore this idea in the context of Krylov-type methods, which form the core of a number of scientific and engineering applications. We experimentally demonstrate that our scheme can cope with severe load imbalance in a coarse-grained Jacobi-Davidson eigensolver that applies local corrections to each block. Furthermore, in a version of the eigensolver code that employs multigrain parallelism, we show that our scheme allows effective use of heterogeneous networks of clusters. Finally, we also demonstrate the utility of our technique for smoothing load balance in a linear system solver that employs a domain-decomposition preconditioner.

3.1 Flexible-phase iteration

A common algorithmic paradigm for parallel programs is that of *synchronous iteration*, in which processors perform local work to complete an iteration of a loop, and then synchronize before proceeding to the next iteration. The following illustrates synchronous iteration on a single program, multiple data (SPMD) computer where the unique ID of each processor is denoted by `my_rank`:

```
while(target state has not been reached) {
    body(my_rank);
    $\boxed{\text{Synchronous interaction(s)}}$
}
```

We are interested in a very specific but important case of synchronous iteration: one in which the amount of work completed by each processor during an execution of the body may be varied arbitrarily without detriment to the end result. With smaller amount of work per iteration, the target can still be reached, but more iterations are required. A common example is inexact Newton iteration [47, pp. 95–112], where solving the Jacobian system to a lower accuracy may require more Newton steps, although eventually the same result will be reached. We decompose algorithms in this category into two phases: 1) A *control phase*, during which synchronous interactions update global knowledge of the current state, allowing each processor to make better decisions later. 2) A *flexible phase*, during which local execution of the body occurs. It is "flexible" insofar as each processor can vary the amount of work that it does during this phase. We designate this type of parallelism *flexible phase iteration.*

Although it is very specific, several important algorithms used in the sciences and engineering fit this model. One class of such algorithms includes stochastic search optimizers such as genetic algorithms and simulated annealing. In their synchronous formulations, processors independently perturb an initial set of configurations to search for more optimal ones. After a certain number of successes are obtained by each processor, they synchronize to decide upon new configurations, and then continue their search. Because synchronization could occur before each processor has had a certain number of successes, the search phase is a flexible phase. The decision phase is the control phase.

Another important class of algorithms that are amenable to a flexible phase iteration structure are Krylov-like iterative methods [77]. These methods are widely employed to solve systems of linear equations, eigenvalue problems, and even non-linear systems. Their main iteration involves vector updates and synchronous dot-products, which allow for little flexibility. However, flexibility can be introduced with preconditioning. At each outer, Krylov iteration, preconditioning improves the current solution of the method by finding an approximate solution to a correction equation. Flexible variants of Krylov methods can solve this equation iteratively [75]. In parallel implementations, this allows different processors to apply different number of iterations on a local correction equation [73]. Thus, the inner, preconditioning step is a flexible phase, and the outer (Krylov) iteration, where the processors update their corrections, is a control phase.

Since each processor need not do the same amount of work as its peers, near-perfect load balance can be achieved during the flexible phase: If all processors are limited to the same time T for an execution of the flexible phase, any differences in the speed of the processors—even due to changes in the external load—cannot cause load imbalance. We have developed an object-based C library, LBLIB, to facilitate the use of this load balancing approach. Appendix A describes its basic functionality and illustrates its usage.

3.2 Coarse-grain Jacobi-Davidson implementation

Any iterative method which employs independent, local preconditioners on each node is a candidate for the load balancing approach that this work investigates. For the bulk of our experiments, we have adapted a recent implementation of a coarse-grain parallel, block Jacobi-Davidson solver for eigenvalue problems, JDcg [80]. JDcg employs the

Jacobi-Davidson (JD) method to approximate a few extreme eigenpairs (λ_i^*, x_i^*) satisfying $Ax_i^* = \lambda_i^* x_i^*$, where A is a large, sparse matrix. For simplicity of presentation, we assume A to be symmetric. The JD method is a Krylov-like method that starts with an initial vector and successively builds a basis V for a space K from which approximations (λ_i, x_i) for the eigenpairs are obtained, usually with the Rayleigh-Ritz procedure [26]. At each step, the residual $r_i = (A - \lambda_i I)x_i$ is computed, and the solution ϵ_i of the correction equation

$$(I - x_i x_i^T)(A - \lambda_i I)(I - x_i x_i^T)\epsilon_i = r_i \tag{3.1}$$

for some approximate eigenpair (Ritz pair) (λ_i, x_i) is obtained and then used to expand the basis. A block version of JD proceeds in a similar manner, but begins with k initial vectors and builds the basis by a block of k vectors at a time, with each vector being the solution of one of k correction equations, each for a different Ritz pair.

In our implementation, we use the iterative method BCGSTAB to solve the correction equation (3.1) because of the indefiniteness of the systems, and because BCGSTAB utilizes a recurrence with only a small number of terms, allowing us to avoid having to store a large number of vectors. In addition, available preconditioners can be used in solving the correction equations.

The JD method can be easily implemented using data parallelism. Each block vector is broken into subsets of rows, and each processor keeps one of those subsets. This requires a global reduction (summation) to be performed for each inner product. A parallel matrix vector multiplication routine and preconditioning operations are provided by the user. Figure 3.1 outlines the data parallel JD algorithm.

In our coarse-grain Jacobi-Davidson code, JDcg, the method is not implemented in a data-parallel fashion. Instead, it employs a hybrid coarse/fine-grained ("multigrain") approach [80, 81] that reduces communication costs and, coincidentally, introduces flexibility into the inner phase. This approach was motivated by the observation that, in many situations, every processor can have access to the entire matrix A. This is often possible because

```
JD
V starting with $k$ trial vectors, and let $W = AV$
  While not converged do:
  1. $H = V^T W$ (local contributions)
  2. Global_Sum($H$) over all processors.
  3. Solve $Hy_i = \lambda_i y_i,\ i = 1 : k$ (all procs)
  4. $x_i = Vy_i,\ z_i = Wy_i, i = 1 : k$ (local rows)
  5. $r_i = z_i - \lambda_i x_i,\ i = 1 : k$ (local rows)
  6. Correction equation Solve eq. (3.1) for each $\epsilon_i$
  7. Orthogonalize $\epsilon_i, i = 1 : k$. Add in $V$
  8. Matrix-vector $W_i = AV_i,\ i = 1 : k$
  end while
```

Figure 3.1: The data parallel Jacobi-Davidson algorithm.

of the large main memory sizes available today (e.g., a 12th order, 3-D finite difference matrix of 1 million unknowns can be stored on a PC with 512 MB) or, more importantly, because many applications do not explicitly store the matrix, providing instead a function for computing the matrix vector product. In JDcg, the matrix A is available on all processors, but the basis vectors V are still partitioned by rows. The main JD step (*projection phase*), or outer loop, is applied in the traditional data parallel (fine grain) fashion, ignoring the matrix redundancy. The inner loop (*correction phase*) is then performed in a control parallel (coarse grain) manner, with each node gathering all the rows of one of the block vectors and performing the matrix-vector product and preconditioner in BCGSTAB independently of the other processors. The following pseudocode describes the coarse-grain interface:

```
Coarse grain correction equation
All-to-all: send local pieces of $x_i, r_i$ to proc $i$,
    receive a piece for $x_{myid}, r_{myid}$ from proc $i$
Apply $m$ steps of (preconditioned) BCGSTAB on eq. (3.1) with the gathered $x_{myid}, r_{myid}$
All-to-all: send the $i$-th piece of $\epsilon_{myid}$ to proc $i$,
    receive a piece for $\epsilon_i$ from proc $i$
```

Although an expensive all-to-all communication is required, the coarse grain interface allows us to arbitrarily improve the parallel speedup of the method because of the independent solution of the correction equations. By increasing the number of BCGSTAB iterations (m), we can shift work from the outer, projection phase to the inner, correction phase, thus increasing the amount of work performed between communication operations. Larger values of m and block size k typically reduce the number of outer JDcg iterations but increase the total number of matrix-vector products, so the algorithm is not work conserving. Large values of m, however, are required for the solution of numerically difficult problems, and the coarse granularity presents an ideal opportunity to hide the large communication latencies of slow networks.

3.3 Load balancing JDcg

JDcg fits the flexible-phase iteration model: The corrections ϵ_i need not be computed to the same accuracy, so the correction phase is flexible. The highly-synchronous projection phase is the control phase. Thus we can load-balance JDcg by restricting each processor to a fixed time T in the correction phase [63]. Even though imbalances will persist during the brief projection phase, overall load imbalance is virtually eliminated, because the correction phase dominates the execution time. Also, some vectors ϵ_i may be computed to lower accuracy, but this only increases the number of outer iterations and often decreases the amount of total work.

To determine an appropriate T, we follow the commonly used guideline that BCGSTAB be iterated to a convergence threshold of 2^{-iter}, where *iter* is the number of outer iterations that have been performed [26]. Using classical convergence bounds for Conjugate Gradient [77], we determine heuristically an "optimal" number of iterations m that corresponds to the 2^{-iter} threshold. To avoid hurting convergence by too large an m, we set a maximum bound *maxits* for m. T is then the time required by the fastest processor to complete m BCGSTAB steps. The algorithm for the load-balanced correction phase proceeds as follows:

Load-balanced correction phase of the JDcg

1. In the first JDcg iteration, do no load balancing. Each processor performs *maxits* BCGSTAB iterations, calculates the rate at which it performed them, and communicates its rate to all other processors.

2. In subsequent JDcg iterations, use the rate measured in the previous iteration to rank the processors from fastest to slowest. In the all-to-all communication of step 8 of the JDcg, faster processors gather the extreme-most eigenpairs and residuals.

3. Use the fastest rate to determine T, and then iterate on the correction equation for this time.

In the load balanced correction phase, faster processors gather the extreme-most eigenpairs and residuals, and the time T is calculated using the rate of the fastest processor. This ensures that the extreme eigenpairs get most of the work, and is necessary because Krylov-like methods naturally find extreme eigenpairs first; time spent on interior eigenpairs benefits convergence relatively little compared to time spent on exterior ones.

3.3.1 Experimental evaluation

We have run the load balanced code on machines loaded with additional "dummy" jobs which introduce external sources of static and dynamic load imbalance by doing meaningless but CPU intensive calculations. Experiments were run using four Sun Ultra-5 Model 333 machines with 256 MB of RAM, connected via switched fast Ethernet. Two large, sparse, symmetric test matrices available from MatrixMarket[1] were used. The first matrix is NASASRB, of dimension 54870 with 2677324 non zero elements. In the experiments using NASASRB, we seek its lowest eigenvalue. Because the lowest eigenvalues of NASASRB are closely clustered, this is a difficult problem that benefits from using a large number of

[1]`http://math.nist.gov/MatrixMarket`

correction equation iterations (we use $maxits = 150$). The second matrix is S3DKQ4M2, of dimension 90449 with 2455670 non zero elements. We use JDcg to find the largest eigenvalue of S3DKQ4M2, which is an easier problem requiring only a small number of correction phase iterations (we use $maxits = 20$). For both cases, BCGSTAB is preconditioned with a local ILUT(0,20) preconditioner [76]. To aid our experimental evaluation of the load balancing scheme, we instrumented JDcg with profiling code that time-stamps the beginning and end of each communication operation. Synchronizing the initial timestamps allows us to determine precisely where each JDcg process is spending its time, and thus to precisely quantify the amount of load imbalance: for each communication that occurs, the time b_i wasted by processor i due to load imbalance is the time the processor took to complete the communication minus the minimum of the times taken by all processors.

Tables 3.1a and 3.1b summarize the performance of JDcg when run against a static set of dummy jobs. The results for both matrices show that the load balancing scheme performs quite well, cutting execution times roughly in half, even for the most heavily loaded cases.

We also ran our JDcg codes against dynamic workloads. Each processor was loaded with an additional dummy job which executes an endless loop that sleeps for a random amount of time and then performs register based computations for a random amount of time. The duration of the sleep and computation phases are sampled from exponential distributions with means λ and μ seconds, respectively. Tables 3.2a and 3.2b summarize our results for matrices NASASRB and S3DKQ4M2, respectively. Since the behavior of the dummy jobs varies between experiments, ten trials were run for each set of parameters, and we report average timing and standard deviations. The load balancing scheme works well in the presence of a dynamic load imbalance. Performance of the scheme generally worsens as λ and μ decrease. This is not surprising, because as the average duration of the computation done by the dummy jobs decreases, the ability to forecast the speed of a node based on its performance during the previous correction phase is lessened. Note, however, that because our scheme forces each processor to spend the same amount of time T in the correction phase, this lessened ability does not result in poor load balancing, but only worse overall convergence due to poorer estimation of the optimal value of T.

A

	Without load balancing				With load balancing			
Load	Its	Mvecs	Time	% Imbal	Its	Mvecs	Time	% Imbal
1111	19	9813	1745	4.3	18	9205	1666	3.8
2111	19	9813	3509	38.5	19	8517	1783	5.6
2211	19	9813	3526	26.4	19	7381	1788	4.4
2221	19	9813	3599	15.5	21	6941	1997	3.6

B

	Without load balancing				With load balancing			
Load	Its	Mvecs	Time	% Imbal	Its	Mvecs	Time	% Imbal
1111	24	1959	601	4.0	25	1921	564	5.5
2111	24	1959	1096	38.0	25	1720	601	8.3
2211	24	1959	1120	27.4	28	1686	712	10.4
2221	24	1959	1121	15.5	35	1706	892	10.4

Table 3.1: Performance when using JDcg to find the smallest eigenpair of NASASRB (A) and S3DKQ4M2 (B) on a system subject to various static external loads. The "Load" column indicates the number of jobs (including JDcg) running on each node. For example, the notation "2211" indicates that two processors are running an additional job and the two other processors execute only the JDcg job. "Its" denotes the number of outer iterations, "Mvecs" the total amount of matrix-vector multiplications, "Time" the total wall-clock time in seconds, and "% Imbal" the percentage of time wasted due to load imbalance over the lifetime of the the application.

3.4 Load balancing a fully multigrain Jacobi-Davidson solver

3.4.1 Benefits of multigrain parallelism

The hybrid coarse/fine grain parallelism employed by JDcg is a special case of what we term "multigrain" parallelism. The projection phase is carried out using a data parallel algorithm suitable for fine-grained parallel architectures, while the correction phase is carried out in a coarse-grained fashion, with each processor working independently to compute its own correction. It may not be appropriate (or even possible), however, for each processor to compute a correction: If we wish to use many processors, it becomes inappropriate because the block size must equal the number of processors, and large block sizes are algorithmically inefficient, dramatically increasing the number of matrix-vector products that must

A

		Without load balancing		With load balancing			
λ	μ	Time	% Imbal	Its	Mvecs	Time	% Imbal
400	400	3135 (141)	23.6 (3.1)	21.0 (1.3)	8586 (486)	2094 (192)	5.4 (1.0)
300	300	3256 (165)	25.9 (2.3)	21.8 (1.1)	8800 (568)	2161 (160)	5.2 (0.5)
200	200	3094 (107)	25.4 (1.7)	21.5 (1.4)	8873 (531)	2122 (189)	5.5 (0.8)
200	100	2778 (115)	23.5 (1.5)	20.9 (1.1)	9250 (569)	2010 (124)	4.8 (0.4)
100	200	3356 (113)	21.1 (1.4)	22.7 (0.7)	9019 (328)	2539 (191)	6.4 (0.9)
100	100	2977 (185)	21.6 (1.8)	22.6 (0.9)	9448 (486)	2313 (86)	5.5 (0.6)

B

		Without load balancing		With load balancing			
λ	μ	Time	% Imbal	Its	Mvecs	Time	% Imbal
200	200	1074 (38)	25.2 (3.0)	28.6 (1.3)	1731 (37)	732 (46)	10.3 (0.9)
100	100	1028 (58)	25.6 (3.4)	28.5 (2.8)	1735 (58)	733 (100)	9.9 (1.9)
100	50	866 (67)	24.0 (3.8)	27.8 (1.6)	1800 (86)	684 (42)	10.3 (1.4)
60	60	1018 (47)	25.2 (1.8)	29.3 (1.2)	1775 (59)	751 (43)	10.1 (1.3)
50	100	1089 (32)	21.0 (1.8)	30.2 (3.0)	1792 (141)	857 (100)	10.3 (1.7)
30	30	974 (45)	23.6 (1.7)	30.4 (4.3)	1861 (196)	776 (132)	10.4 (2.3)

Table 3.2: Performance averages and their standard deviations (in parenthesis) for 10 different runs on NASASRB (A) and S3DKQ4M2 (B) with come-and-go dummy jobs on each node. Dummy jobs execute a loop in which they sleep for γ seconds and then perform computations for ξ seconds. γ's and ξ's are sampled from exponential probability distributions with averages λ and μ, respectively. In cases without load balancing, 19 outer iterations and 9813 matrix-vector multiplications a performed for NASASRB; for S3DKQ4M2, 24 iterations and 1959 mat-vecs.

be calculated. In other cases, the matrix may simply be too large to fit in memory on a single compute node. When it is not practical for each processor to compute a correction, fully multigrain parallelism should be employed, in which a subset of processors—a "solve group"—rather than a single processor, are used in parallel to solve a correction equation. The only real difference from JDcg is that a data parallel algorithm is used within each solve group to solve the correction equation corresponding to the block vector gathered onto it. The fully multigrain algorithm is implemented in JDmg [58, 59] and is advantageous in several ways: First, it allows the number of processors P to exceed the block size k. Second, it eliminates the requirement that processors be able to hold the entire matrix in primary storage. Third, it facilitates use of computational resources connected by heterogeneous networks. For example, if two clusters both utilize Myrinet internally, but the interconnect between the clusters uses much slower Fast Ethernet, a block size of 2 could be used with each cluster forming a solve group. The brief projection phase would be limited by the slow connection between the two clusters, but during the correction phase each cluster could take advantage of its fast internal network.

3.4.2 Load imbalance in the full multigrain case

Though the multigrain algorithm has several benefits, it can accentuate or even introduce load imbalance. For instance, if three identical processors are used to run JDmg with a block size of two, one solve group will contain two processors and the other only one. Since the latter group is roughly only half as fast as the former, the load imbalance is now 33%! Clearly the utility of the multigrain scheme is limited if not used in conjunction with a load balancing scheme. Fortunately, like the coarse grain version, JDmg fits the flexible-phase iteration model and can be load balanced in the same manner [57]. The only real difference is that solve groups, rather than individual processors, are the entities that work independently during the correction phase. Processor 0 of each solve group is responsible for coordinating the load balancing: those processors track the speed of the subgroups, determine the time T to iterate on the correction equations, and inform the other members of their solve group when they should halt the correction phase.

We used LBlib to load balance JDmg and tested the performance of the scheme using SciClone, a heterogeneous cluster of workstations at the College of William & Mary. Figure 3.2 details its architecture at the time of the experiments. To enable measurement of load imbalance in the multi-grain experiments, we timestamp synchronous communication calls in JDmg, much as we did with JDcg. We do not timestamp communications internal to the solve groups during the correction phase: there is no need, as load imbalance does not arise because the machines in a solve group are homogeneous. Table 3.3 summarizes the performance of the multigrain code, with and without load balancing, on a variety of solve group configurations. Without load balancing, load imbalance is quite severe in

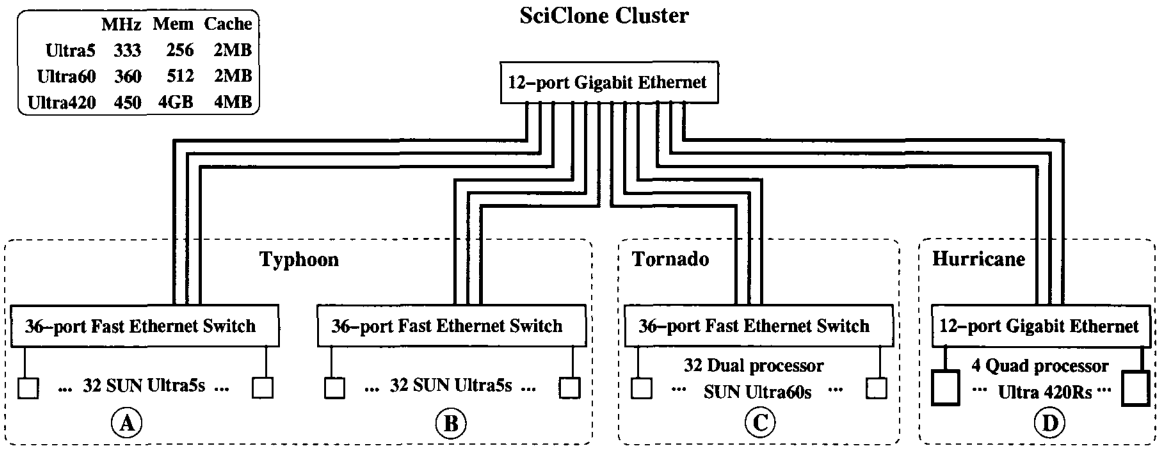


Figure 3.2: SciClone: The William and Mary heterogeneous cluster of three homogeneous clusters: Typhoon, Tornado (also called C), and Hurricane (also called D). We distinguish between A and B, the subclusters of Typhoon, because their intercommunication passes through the Gigabit switch. There are three levels of heterogeneity: node architecture, number of nodes, and networks.

many of the configurations. Employing load balancing significantly improves performance, sometimes doing better than halving execution times. Load imbalance is almost always held below a tolerable level of 10%.

3.5 A load balanced Additive-Schwarz preconditioner for FGMRES

So far we have applied our load balancing method only to the coarse/multi-grain Jacobi-Davidson eigenvalue solver. In this section we demonstrate its applicability in a different setting. The Generalized Minimal Residual (GMRES) method [72] is one of the most popular iterative methods for solving unsymmetric linear systems. A variant known as Flexible GMRES (FGMRES) [75] allows a different preconditioner to be used at each step. In particular, an iterative method may be used. We have applied our load balancing scheme to a parallel FGMRES solver that employs a one-level additive Schwarz method (ASM) preconditioner. ASM is an iterative method, the basic idea of which is to break the problem domain into (possibly overlapping) subdomains and to approximately solve for a correction vector on that domain while ignoring dependencies upon the other subdomains. The correc-

	Without load balancing			With load balancing		
Nodes	Time	Mvecs	%imbal	Time	Mvecs	%imbal
AD	3265	13058	36.17	1746	10515	4.47
$A_{16}A_{16}D_8D_8$	4022	16910	38.96	1692	11208	5.14
$C_{32}D$	3282	13058	39.57	1631	10478	5.01
$A_{16}A_{16}C_{16}C_{16}$	1405	12424	11.02	1546	14698	5.24
$C_{16}C_{16}D_8D_8$	4037	16910	41.71	1544	10711	6.22
AC_{32}	1585	12730	9.46	1450	12833	2.05
CD	3495	13996	52.37	1381	9608	7.68
$C_{32}C_{32}D_8D_8$	3132	13124	58.32	1284	11202	9.94
(AB)C	1198	12656	11.97	1214	13653	5.98
(AB)D	3500	13996	55.42	1126	8870	8.97
$ABC_{32}C_{32}$	981	12240	21.00	991	14167	8.99
ABD_8D_8	3152	13124	61.58	941	8680	11.78
$(AB)C_{32}C_{32}D$	1870	14534	52.64	724	9481	15.05

Table 3.3: Performance of the multigrain JD running on different node configurations, with and without load balancing. "Time" is wall-clock time in seconds and "Mvecs" is the number of matrix-vector products computed. "%imbal" is the percentage of time wasted due to load imbalance. Strings within the "Nodes" column specify what nodes are used for an experiment: For each subcluster that is utilized, its letter is given. If a subscript n is appended to that letter, it indicates that only n processors of the subcluster are utilized; if no subscript is present, all processors are utilized. For instance, "C" means that all 64 processors of cluster C are used, while $C_{32}D$ indicates that 32 processors from cluster C are used together with all the processors from cluster D. For each subcluster utilized, its letter is given. When multiple subclusters are assigned to one block vector, they are grouped together with parentheses. E.g., "(AB)" indicates that subclusters A and B work together on the same block vector (are in the same solve group), whereas "AB" indicates that subclusters A and B work on different block vectors (each composing their own solve group).

tions are then added to the current approximation to the solution to obtain the next iterate. Because the components of each subdomain are not updated until the end of an iteration, the subdomain solves can be performed independently, and therefore in parallel. Because the solves can proceed independently and need not be performed to the same accuracy, ASM fits the flexible phase iteration model.

When one step of ASM is used to precondition an FGMRES iteration, the application of the preconditioner can be load balanced by restricting all processors to spend the same amount of time on their subdomain solves. Using this approach is considerably more complicated for ASM than for Jacobi-Davidson, however. For Jacobi-Davidson, we select the time T for the correction phase by identifying an optimal number of iterations and then estimating the time required for the fastest processor to complete that number. If slower processors complete significantly fewer iterations, this does not prevent convergence. (Indeed, in the proceeding chapter we demonstrate how we cope with memory pressure on a node by having it do no iterations.) Such an approach is not possible for ASM preconditioned FGMRES. Convergence may stagnate if each subdomain solve does not reach a certain level of accuracy. Furthermore, it is difficult to predict how many iterations are needed to reach a given accuracy: some subdomains are much "harder" than others, and on the same subdomain the number of required inner iterations may vary considerably from outer iteration to outer iteration. For these reasons, and to enable better response to dynamic load variations, we determine the time T dynamically by building a consensus: The inner phase ends when all processors have reached a minimum accuracy for their subdomain solve. This incurs some communication overhead, but because domain decomposition methods lose their effectiveness as more subdomains are added, the number of subdomains should not become great enough to cause problems.

We added functions to LBLIB to support determining T by consensus and used the library to write a load-balanced ASM preconditioner for the PETSc framework [5, 6] developed at Argonne National Laboratory. The load balancing functionality is completely encapsulated within the preconditioner and can be used by any application that uses a PETSc solver that supports an iterative preconditioner.

Table 3.4 summarizes the performance of FGMRES when used with the load balanced ASM preconditioner to solve a two-dimensional convection-diffusion problem with a Reynolds number of 1000. Differences in the conditioning of the subdomains cause load imbalance even when no external jobs are present; in this case use of the load balancing scheme reduces execution time by an average of 18%. In the worst-loaded case, execution time is reduced by 24%. Such gains are significant, but it is clear that the load balancing scheme cannot fully correct any substantial load imbalance. This is because increasing the accuracy of a subdomain solve yields diminishing returns: once the solve reaches a certain accuracy, going beyond that does nothing to speed convergence of the outer method. For domain decomposition methods, it appears that repartitioning is the only way to correct substantial external load imbalance. Because repartitioning is costly and cannot be done frequently, our load balancing approach could play the role of reducing load imbalance in between repartitionings. We also note that our method is useful for smoothing out load imbalances due to differences in the condition number of the subdomains; partitioning methods that can ensure that all domains are similarly conditioned are not known.

	non-lb		lb	
Case	Its	Time	Its	Time
1111	56	77	45.0	63.6
2111	56	137	41.3	105.7
1211	56	118	44.0	102.7
2211	56	142	43.0	122.6
1112	56	104	40.7	83.9

Domain	Min its	Max its	Avg its
0	2	34	19.7 (7.7)
1	3	29	16.5 (6.2)
2	4	32	12.3 (5.2)
3	3	25	14.6 (4.5)

Table 3.4: Performance of FGMRES when preconditioned with ASM without load balancing ("non-lb") and with load balancing ("lb"). Four processors from the Typhoon subcluster (A in figure 3.2) were used to solve a convection-diffusion problem with Reynolds number 1000, discretized with a 5-point centered-difference scheme. Subdomains are solved to a relative tolerance of 1e-1 without load balancing and to a higher accuracy on lightly loaded machines when load balancing is used. Some subdomains are harder to solve than others, leading to natural load imbalance. The right table shows the minimum, maximum, and average number (std. dev. in parentheses) of iterations required to solve each subdomain problem to a relative tolerance of 1e-1. The left table depicts performance for different external load cases; the notation used is as in table 3.1. For example, "2111" means that the node 0, which has the hardest domain, is loaded with a dummy job. We ran experiments for the cases which result in the most load imbalance. Because execution time is variable when load balancing is used, we report average times for the load balanced code.

Chapter 4

Load balancing under memory pressure

The load balancing scheme we described in the preceding chapter works well when external load is CPU intensive in nature. In many cases, however, external jobs compete for scarce memory resources as well. If the memory requirements of the processes on a node far exceed the available physical memory, a sizeable number of CPU cycles will be wasted as the system swaps pages to disk. Furthermore, if the page replacement policy of the virtual memory system is ill-suited to the memory access patterns of the jobs, the majority of CPU cycles will be wasted as the system thrashes. Such memory pressure on a node can cause a synchronous parallel job to grind to a standstill under tremendous load imbalance. In this chapter, we explore how our load balancing scheme can be modified to improve performance under memory pressure.

4.1 Load-balanced JDcg under memory pressure

To determine the robustness of our load balancing scheme under memory pressure, we ran some experiments in which we loaded nodes with a dummy program requiring a large amount of memory. The dummy job continuously performs matrix-vector multiplications with a matrix of size M MB. Our initial experiments were conducted on machines running Solaris 7, but we found that the operating system would prevent thrashing by essentially "starving" either JDcg or the dummy job, scheduling one job consistently in favor of another. This policy is documented in [56]. Because of this behavior, we instead conducted our experiments on a system of four 1 GHz Pentium III machines with 128 MB of RAM running RedHat Linux 6.2, which utilizes a version of the Linux 2.2 kernel. The machines were interconnected via fast Ethernet.

We found that a dummy job of size $M = 80$ MB was sufficient to consistently cause excessive swapping. Furthermore, increasing M did not increase the amount of swapping observed, so we used that value in our experiments. Table 4.1 displays our results for NASASRB (S3DKQ4M2 is too large to use on these systems). For load balanced JDcg, the execution time becomes considerably worse as the external load is increased. This is largely due to the fact that in these experiments, a node that is running a dummy job is so much slower than the fastest node(s) that it only completes one BCGSTAB iteration each time through the correction phase (and even with it performing only one iteration, the other nodes still must wait for it). However, when compared with the performance of non-load balanced JDcg, the numbers are very encouraging. Without load balancing, JDcg fares very poorly when multiple nodes are burdened with external jobs (2-2-1-1 and 2-2-2-1 cases)—much worse than when only a single node is burdened (2-1-1-1 case). This is somewhat surprising: since non-load balanced JDcg is limited by the speed of the slowest processor, one might expect the difference between its performance under loads of 2-1-1-1 and 2-2-1-1 to be minimal, as it is in the experiments described in section 5.1. This is perhaps due to extra load imbalance that is introduced because the timing of swapping activity may not coincide with that of other nodes subject to external loads.

4.2 Avoiding thrashing from within the algorithm

Our experiments demonstrate that the load balancing scheme significantly improves performance under memory pressure. It is clear, however, that even when using the scheme, a great deal of time is still lost to disk swapping and the resultant load imbalance.

	Without load balancing		With load balancing			
Load	Time	% Imbal	Its	Mvecs	Time	% Imbal
1-1-1-1	823	5.4	20	10312	843	4.3
2-1-1-1	3288	56.4	21	8464	1566	32.7
2-2-1-1	6677	51.7	20	5225	2067	34.0
2-2-2-1	7500	36.7	25	3535	2719	23.9

Table 4.1: Results for NASASRB with some nodes externally loaded by static, memory intensive dummy jobs. Externally loaded nodes run a dummy job that performs repeated matrix-vector multiplications using a matrix of size 80 MB. In the cases without load balancing, 19 outer iterations and 9959 matrix-vector multiplications are always performed.

In an attempt to improve the performance of load balanced JDcg in the presence of memory-intensive external loads, we have implemented a heuristic that recedes JDcg during the correction phase on a node that is thrashing. The idea is to let the competing job utilize 100% of the CPU and memory resources, hopefully speeding its completion and hence relinquishment of resources.

Obviously, this strategy will only be effective if the lifetime of the competing job is less than that of JDcg. Performance prediction models [13, 91] could be used to estimate these lifetimes, but obtaining useful estimates may be difficult. In the current work, we use no such predictive models and instead use a natural upper limit on how long JDcg stays backed off before it resumes: the time T that each node is given to perform the correction phase. If the lifetime of the competing job is longer than that of JDcg, this means that the anti-thrashing scheme will "starve" the correction phase on a node that is thrashing. The load balancing scheme will, however, ensure that JDcg still converges because other nodes will shoulder the load.

Upon entering the correction phase (excluding the first time, in which no load balancing is done), the anti-thrashing algorithm checks to see if the thrashing was occurring during the just completed projection phase. If so, it recedes for T_{wait} seconds. The choice of T_{wait} is discussed later. After this period, it checks to see if it is appropriate to resume computation or to continue waiting. If the application decides to resume computation, it checks to see if the node is thrashing after the completion of each BCGSTAB iteration. If so, it recedes once again for T_{wait} seconds. The following pseudocode describes the basic steps of the algorithm:

```
Anti-thrashing modification of the correction phase
      Reset the clock $T_{elapsed} = 0$
L1:   If $\left( \boxed{\text{Test for receding}} \right)$ then
          repeat sleep ($T_{wait}$)
          until $\left( \boxed{\text{Test for resuming}} \right)$
      endif until $\left( \boxed{\text{Test for resuming}} \right)$
      while ($T > T_{elapsed}$ + estimated time to complete a BCGSTAB iteration)
          perform one BCGSTAB iteration
          if $\left( \boxed{\text{Test for receding}} \right)$ then goto L1
      end
```

The performance of the algorithm depends on its ability to correctly determine the answers to two tests: (1) should it recede if currently running and (2) should it resume if currently sleeping. That is, it must be able to accurately identify when the system is thrashing and to predict whether or not resuming computation will cause thrashing. While it may be better to answer these tests using performance measurement/prediction middleware, our experiments have shown that one can obtain useful results using some simple performance measures that the operating system makes available.

For the first test (test for receding), JDcg checks the memory page swap-out rate *swapout* and recedes if it exceeds a given threshold *swap_threshold*. We also considered using such information as page fault rates and page swap-in rate, but we found the swap-out rate to be a very reliable indicator. Additionally, it may be desirable to check to see if the CPU idling time exceeds some tolerable threshold, since if there is some paging but little CPU idling, there is little benefit to receding. We did try using an idle time threshold, but, again, we found the swap-out rate alone to be a very reliable indicator so we did not resort to a more complicated criterion. In a following section we describe how we choose an acceptable swap-out threshold. The pseudocode below describes the first test:

```
Test for receding
 Obtain the swap-out rate for the just completed computation phase: swapout
 If ( swapout > swap_threshold )
     then return TRUE
```

Once JDcg has receded, it must be able to determine whether beginning/resuming BCGSTAB iterations (and thus swapping JDcg back into memory) will cause thrashing. This may be as easy as determining that the amount of free memory on a node is enough for JDcg to run. Unfortunately, lower priority processes or the buffer cache may utilize large amounts of memory that they would free if asked by a higher priority job, so the free memory size reported by the system usually greatly underestimates the amount of memory that is actually available to JDcg. Therefore, in addition to checking the amount of free memory, JDcg also checks the idle time of the system: if the memory-intensive external job has finished and there is no other CPU intensive job running on the system, the idle time should be very high because JDcg is recessed. (Note that JDcg considers time spent by the CPU running nice'd jobs as idle time, so low priority jobs that are using lots of CPU time will not interfere with this strategy.) The following pseudocode describes the test for resuming:

```
Test for resuming
 If ($T_{elapsed} + T_{wait} > T$) then
     return TRUE and exit the correction phase
 else
     Obtain the CPU idling percentage over the waiting period: idle
     Obtain the current available free memory: free_mem
     Obtain the current resident memory for JDcg: JD_res_mem
     Estimate the memory JDcg requires to avoid thrashing: JD_req_mem
     If ( idle > idle_sufficient ) OR ( JD_req_mem − JD_res_mem < free_mem )
         then return TRUE
 endif
```

Note that the first line of the above algorithm checks to see if waiting for T_{wait} seconds will cause JDcg to spend more than the allotted T seconds in the correction phase. If so, JDcg exits to the correction phase to ensure that this does not happen.

4.2.1 Choosing the parameters

The sleep time T_{wait} must be short enough to allow JDcg to quickly respond when the system becomes available, but it should also not be so short that an excessive amount of CPU time is spent checking the status of the node. We choose the longest T_{wait} that yields a tolerable ratio T_{wait}/T, which is the upper bound on the amount of time that may be wasted by a node when the competing memory-intensive job finishes while the JDcg process is sleeping.

Page swap-out activity comes in bursts, with the frequency depending on the particular memory access patterns of the programs that are running. This means that the *swap_threshold* is sensitive to the time interval over which *swapout* is sampled. Ideally the computation interval between the two measurements used to calculate *swapout* should be longer than the interval between bursts. It is not feasible, however, to adapt JDcg to ensure that the computation intervals exceed the burst intervals. Alternatively, the burst interval t_b could be estimated and then a program thread could accumulate the swapout rate over the past t_b seconds. In our experiments we found that simply measuring *swapout* over the computation interval was adequate. Note that determining an appropriate value of the *swap_threshold* is not difficult, since even low levels of swap-out activity signify memory contention: a very small *swap_threshold* is sufficient.

The value of *idle_sufficient* should be large enough to ensure that a non-idle system is not mistaken as an idle one, but must also be small enough so that jobs with low CPU utilization are ignored. We have found that a value between 80 and 90% works well.

The swap-out rate, idle percentage, and current amounts of free and JDcg resident memory can all be calculated from statistics provided by system counters that are accessible on most Unix-like systems through the `/proc` pseudo-filesystem.

4.2.2 Experimental results

As stated previously, we have observed on the Sun platforms that the Solaris scheduler prevents thrashing by arbitrarily "starving" one of the processes. This increases the overall throughput on a node but can be very detrimental to parallel jobs. In the case of JDcg, both the projection and correction phases are starved, impeding progress on all of the nodes the JDcg is using.

Because of this behavior, we conducted our experiments with the anti-thrashing scheme using a system of four 1 GHz Pentium III machines interconnected via switched Fast Ethernet and running RedHat Linux 6.2. Our test matrix is NASASRB. JDcg requires about 67 MB of memory when applied to NASASRB, so we chose JD_req_mem = 66 MB, just below that value. We observed that, for a measurement interval $\geq t_b$, average swap-out rates never dipped below 150 KB/s, so we used this value for *swap_threshold.* T_{wait} was set to a sizable 20 seconds. The values of *swap_threshold* and T_{wait} are fairly conservative and may result in JDcg receding fewer times than optimal, but are unlikely to result in JDcg receding erroneously.

All of our experiments were run on four nodes, with node 0 running a dummy job that alternates between sleeping and performing matrix-vector computations. We have found that using a dummy job of size 80 MB is sufficient to consistently cause excessive swapping when the dummy job and JDcg are running. Furthermore, we found that increasing the size of the dummy job did not increase the amount of swapping because the memory access pattern does not change. For these reasons, we have chosen to use a dummy job size of 80 MB in our experiments. The dummy job is described by the following pseudocode:

Code for dummy job:

```
Wait 60 seconds to allow JDcg to initialize.
Create a vector v and a random, dense matrix $A$ of size 80 MB.
For $i = 1$ to $lim$ do
    For $j = 1$ to $nmv$ do
        $\mathbf{b} = A\mathbf{v}$
    End for
    Sleep for $\lambda$ seconds.
End for
```

The dummy job can be viewed as simulating a series of jobs that do a specified amount of work and then terminate. Each of the *lim* passes through the outer loop can be viewed as the execution of one of these "jobs". The inner loop executes the *nmv* matrix-vector multiplications that each "job" performs. The inter-arrival time between the "jobs" is λ seconds. Unlike in the experiments described in previous sections, it is necessary to use a dummy job that does a specified amount of work and then stops, since the idea of the anti-thrashing scheme is to speed up JDcg by speeding the completion of the memory intensive jobs that compete with it. Therefore, we chose values of *lim* and *nmv* so that the (actual) dummy job can complete all of its work within the lifetime of JDcg (without the anti-thrashing scheme). Table 4.2 summarizes our results. Because there is some variability in the performance obtained using a particular set of parameters, we report averages over five trials and their standard deviations in parentheses.

			Without anti-thrashing			With anti-thrashing		
λ	*nmv*	*lim*	Its	Mvecs	Time	Its	Mvecs	Time
n/a	46	1	19.4 (0.9)	8681 (714)	1125 (175)	19.6 (0.5)	9625 (287)	913 (27)
n/a	93	1	19.6 (0.5)	8202 (319)	1256 (92)	19.2 (0.8)	9317 (516)	928 (39)
30	3	4	19.6 (0.9)	9011 (654)	1004 (89)	19.0 (0.0)	8820 (108)	961 (46)
60	7	3	20.4 (0.9)	9088 (649)	1136 (98)	19.6 (0.9)	9088 (222)	943 (15)
60	15	2	20.6 (1.3)	9136 (812)	1157 (131)	19.6 (0.5)	9199 (416)	946 (49)

Table 4.2: Performance of the anti-thrashing scheme for NASASRB with a come-and-go memory intensive dummy job on one node. The tests for receding/resuming in the anti-thrashing algorithm are performed through system measurements. The dummy job performs *lim* repetitions of *nmv* matrix vector multiplications with an 80MB matrix, sleeping for exactly λ seconds between each pass.

Our experiments show that the anti-thrashing scheme yields an appreciable performance gain, often reducing execution times by 20% compared to load balanced JDcg without anti-thrashing. To put these numbers in perspective, it should be noted that the original load balancing scheme often mimics the anti-thrashing scheme, usually completing only one BCGSTAB iteration per correction phase on a node that is thrashing because the performance of the node is so poor. Obviously, the non-load balanced code is simply not viable for such situations.

The performance of the scheme is dependent on its ability to correctly determine when to recede and when to resume. Because our scheme cannot make these decisions perfectly, we obtained an upper bound on the scheme's performance by running some experiments in which we allowed our application to "cheat." Rather than using system statistics to determine when to recede and resume, we modified the dummy job to create a specific file when it begins a computation phase and to delete it when it ends. JDcg could then check for the existence of the file to make its decisions, which would be made perfectly. Our results are summarized in Table 4.3. Comparing these ideal results to those in Table 4.2, we can see that the performance is essentially the same.

			With anti-thrashing		
λ	*nmv*	*lim*	Its	Mvecs	Time
n/a	46	1	20.2 (1.1)	9936 (651)	942 (71)
n/a	93	1	19.4 (0.5)	9233 (313)	920 (39)
30	3	4	19.5 (0.6)	9346 (323)	970 (48)
60	7	3	19.6 (0.9)	9414 (461)	917 (42)
60	15	2	20.0 (0.7)	9774 (440)	961 (45)

Table 4.3: Performance of the "ideal" anti-thrashing scheme for NASASRB with a come-and-go memory intensive job on one node. The JDcg knows exactly when the system is thrashing by communicating directly with the dummy job.

In addition to shortening the execution time of JDcg, the scheme also has the added benefit of increasing total system throughput. Table 4.4 shows that the reduction in the execution time of the dummy job can be dramatic.

One might be concerned with how our scheme performs when the competing memory-

			without anti-thrashing	with anti-thrashing	with ideal testing
λ	*nmv*	*lim*	Time	Time	Time
n/a	46	1	736 (409)	113 (17)	140 (15)
n/a	93	1	1055 (129)	187 (22)	202 (26)
30	3	4	554 (66)	223 (73)	171 (40)
60	7	3	808 (97)	152 (50)	141 (19)
60	15	2	864 (150)	158 (20)	141 (22)

Table 4.4: Improvements in execution time of the dummy job because of increased processor throughput due to our anti-thrashing scheme.

intensive application outlives JDcg. In this case, the scheme cannot reduce the runtime of JDcg, and could conceivably even degrade performance, since a node that is thrashing will never do any BCGSTAB iterations. We ran some experiments with a dummy job that outlives JDcg and verified that, regardless of whether or not the memory-intensive job outlives JDcg, the scheme does not result in any measurable performance degradation. As noted previously, when JDcg runs without the anti-thrashing scheme on a node that is thrashing, typically the node is so slow that it only completes one BCGSTAB iteration per correction phase. Reducing this number to zero with the anti-thrashing scheme does not have a noticeable effect on convergence. The anti-thrashing scheme does, however, significantly improve the overall system throughput.

4.3 Limitations of the memory-balancing approach

We have presented some favorable experimental results using our "memory balancing" scheme that demonstrate the practicality of implementing programs that dynamically adapt to memory pressure. However, we have ultimately found the scheme to possess significant limitations that we cannot remedy.

One of the biggest technical shortcomings is that the tests for deciding when to recede or resume are easily fooled. They work in our controlled experiments, but have proven to be impractical in more realistic situations. The test for receding is fairly reliable, although swap-out activity does not necessarily indicate that our process is experiencing memory pressure—a lower-priority process or one that has been long idle may have pages swapped out while our pages remain untouched, for example. The test for resuming, however, has serious shortcomings. A recessed JDcg process will resume correction iterations if either $idle > idle\_sufficient$ or $JD\_req\_mem - JD\_res\_mem < free\_mem$. Unfortunately, the value of $free\_mem$ is usually meaningless: free memory is almost always close to zero because any memory that would otherwise be free is usually consumed by the buffer cache. The test for resuming, therefore, generally reduces to checking to see if the CPU idle percentage exceeds *idle_sufficient*. This works in our controlled experiments, but in a real-world situation, there is no guarantee that the CPU will stand idle after a competing job has finished its memory-intensive work. Thus our inability to determine the actual amount of memory available to JDcg is a serious obstacle to the practical application of our scheme. Unfortunately, operating systems provide no information that can be used to reliably estimate this quantity.

Another problem stems from the inability of the scheme to remedy load imbalance during the projection phase. When load imbalance is due solely to external CPU load, this is of little concern, as the solver spends little time in this phase. When a system is thrashing however, the time spent in the projection phase can turn into an eternity. In the worst case scenario, which we have observed under Solaris 7, the entire parallel job can grind to a halt as the memory scheduler swaps out the process during the projection phase.

Perhaps the most fundamental shortcoming is that our scheme is applicable only to a class of algorithms far narrower than that to which the load balancing scheme is itself applicable. The flexible phase must be extremely flexible: a node must be able to abdicate all responsibility for performing computations during the phase. This is the case for JDcg, but neglecting one of the subdomains in our load-balanced additive Schwarz preconditioner can result in stagnation of convergence: other processors may continue to work, but the solve cannot progress without the preconditioner completing some work on the subdomain.

For the above reasons and others, we have pursued another, less restrictive approach in which applications gracefully degrade their memory requirements and performance in the face of memory pressure, rather than completely receding their computations. We describe this approach in the following chapters.

Chapter 5

A dynamic memory adaptation framework

In our earliest memory adaptation work, we worked with JDcg and exploited its flexibility. If memory shortage was detected on a node, the node would perform no correction phase iterations. This would allow a competing, memory-hungry job to utilize 100% of the CPU and memory resources, hopefully speeding its completion and relinquishment of resources. Our experiments yielded an appreciable performance gain, often reducing execution times by 20% compared to load balanced JDcg without memory adaptation. The method suffers from several shortcomings, however, the biggest of which is that it is applicable only to a limited subset of flexible-phase algorithms. Additionally, the mechanisms we used for identifying when to recede and resume correction phase iterations are not reliable in general.

In more recent work [61, 62] we have developed a memory adaptation framework which is widely applicable and highly portable. We describe it in this chapter.

5.1 A portable framework for memory adaptivity

Many scientific algorithms, such as iterative methods for linear and nonlinear systems, dense matrix methods, and Monte Carlo methods, operate on large data sets in a predictable, repetitive fashion. To best utilize hierarchical memory, applications often operate in a block-wise fashion to increase locality of memory access. Algorithms designed to run in-core are blocked to effectively utilize L1 and L2 caches, while out-of-core algorithms employ a similar strategy to best utilize DRAM. In the lingo of out-of-core algorithms, blocks are sometimes referred to as panels to distinguish from disk blocks; we adopt that terminology here. With data partitioned into P panels, the processing pattern of a blocked algorithm can be represented schematically as

```
for i = 1:$P$
    Get panel $p_i$ from lower level of the memory hierarchy
    Work on $p_i$
    Write results back and evict $p_i$ to the lower level of the memory hierarchy
end
```

The above structure suggests a simple mechanism for memory adaptation: control the resident set size by varying the number of panels cached in core. If the program has enough memory, it caches its entire data set in core and runs as fast as a standard in-core algorithm. If main memory is scarce, the number of panels cached is reduced, and an application-specific cache replacement policy is used. The magnitude of the reduction varies according to the memory shortage; so performance should degrade gracefully as memory becomes scarce (provided an appropriate cache-replacement policy is used). Thus if the amount of physical memory available is only slightly less than the size of the adaptive program's data set, its performance should be close to its in-core performance. A non-adaptive in-core program, on the other hand, may thrash under the same conditions if the replacement policy of the VM system is inappropriate for its access pattern. This is often the case for scientific applications, which commonly access large data sets in a cyclic fashion. For such access patterns, a most recently used (MRU) replacement policy should be used, but the generic replacement policy used by the operating system is usually an approximation of least recently used (LRU) replacement

The software required to support the above memory adaptation strategy can be easily encapsulated into a software library. An existing blocked code can then be easily modified by the insertion of a few library calls. Essentially, all that needs to be done is to make the code call a library function that returns a pointer to panel p_i before working with it. The function call should handle all memory management in a way that is completely transparent to the application programmer. We have written such a library, which we call MMLIB (for "memory malleability library").

5.2 Elements of the implementation

To use our adaptation strategy we must be able to grow and shrink the memory space of an application. Most implementations of `malloc()/free()` are not appropriate because they do not provide a mechanism to release memory back to the operating system. A successful `malloc()` call can grow a program's heap. When `free()` is used to deallocate memory, however, the heap size may not be decreased, so freed memory is not returned to the operating system. Our solution is to use memory mapping, which is universally available in modern operating systems and provides more explicit control over memory usage. Memory obtained via a memory map can be easily returned to the operating system; additionally, many operating systems allow programmers to provide hints to the OS via `madvise()` about how a mapped region will be used. Using named mappings to files (viz., memory-mapped I/O) confers other advantages as well. Because I/O is handled transparently, codes can be simplified: explicit I/O calls are not needed, and an adaptive code can greatly resemble and in-core one. I/O traffic is optimized because non-dirty pages in mapped regions can be freed without a write, while pages that are dirty will be written, but not to the swap device. Writing dirty pages to swap space incurs software overhead and results in poor data placement on disk, since pages that are part of a contiguous array may be scattered between several non-contiguous blocks on the swap device.

In our implementation, each panel is written to a disk file. A panel to be cached in core is mapped via an `mmap()` call, and the VM system is asked to prefetch that panel via `madvise()`. A panel to be evicted is unmapped via an `munmap()`. A memory adaptation decision is made each time the program calls our library function to fetch a new panel. Based on its current estimates of memory availability, it chooses to increase, decrease, or maintain the number of panels cached in core. If panels must be evicted, victim panels are selected using a user-specified, application specific policy. This allows an optimal replacement policy to be used, in contrast to the general policy employed by a VM system, which may be highly suboptimal for a given application.

5.3 Algorithms for adapting to memory availability

An adaptive application should quickly decrease its memory usage when shortage is detected, while still caching as many panels as possible. When surplus memory becomes available the application should quickly map additional panels in order to utilize that memory. The main challenge involved in implementing our memory adaptation framework is devising a simple, reliable mechanism for answering two basic questions: (1) Is there a memory shortage? and (2) If not, is there a memory surplus? Because many systems export very limited information about memory usage, these questions should be answered using as little system information as possible to ensure portability. We have found ways to answer both questions using only the resident set size (RSS) of the program.

Detecting memory shortage is the easier of the two tasks. Memory shortage can be inferred from a decrease in the program's RSS that occurs without any unmapping on the part of the program. The magnitude can be ascertained by comparing the current RSS to what the program estimates it should be if all panels it is currently trying to cache are actually resident in memory.

Detecting surplus memory is more challenging. Ideally the system would provide an estimate of the amount of available memory, and the program would use this to determine when there is enough room to cache additional panels. The best that most operating systems provide, however, is the amount of free memory, which typically greatly underestimates the amount of additional memory actually available. On many systems, the amount of free memory is usually very close to zero, because any memory that is not needed by running processes is used by the file cache. The system might still service a large memory request, however, by reducing the file cache size. Because the operating system simply does not provide much information, the most reliable way to determine if a quantity of memory is actually available is to try to use it and see if it can be maintained in the resident set.

5.3.1 Detecting memory shortage

Consider an application whose memory is managed by our library. We denote by `Panels_in` the number of panels that are cached in memory. We can calculate what the RSS should be if all panels that we are attempting to cache are actually resident. We refer to this quantity as desired RSS and denote it by `dRSS`:

dRSS = Panels_in * Panel_size + sRSS,

where `sRSS` indicates the size of what we term the *static memory*, that is, memory not managed by our memory-malleability library. (We will explain how we determine `sRSS` in section 5.4.1.) By definition, the program is under (additional) memory pressure when it cannot maintain `RSS` equal to this desired RSS. This suggests the following scheme:

```
if ( RSS < dRSS ) then
    diff = (dRSS-RSS) / Panel_size
    Unmap diff panels
    Panels_in = Panels_in − diff
    dRSS = dRSS − diff * Panel_size
```

This "solution" is not sound, however, because when the program unmaps panels in response to memory pressure, the victim panels may not correspond to the memory paged out by the operating system. Using the above scheme can lead to a cascade of unmappings all the way down to `Panels_in` = 1. Consider an example (illustrated in Figure 5.1) where the replacement policy is MRU and the program's data set is broken into five panels, all of which are currently mapped (`Panels_in` = 5). A memory shortage has caused the system to evict portions of panels 1 and 2 from memory, but there is enough memory available to keep four panels mapped (`diff` = 1). When the program accesses panel 4, the condition (`RSS` < `dRSS`) holds, so `Panels_in` is decreased to 4 by evicting the MRU panel 3. However, panel 3 was fully resident, so its unmapping causes `RSS` to reduce even further by exactly

Panel_size. This is the same amount the dRSS is reduced, so when the program tries to access panel 5, the condition (RSS < dRSS) still holds—despite the availability of memory. The above process repeats until all panels but one are unmapped.

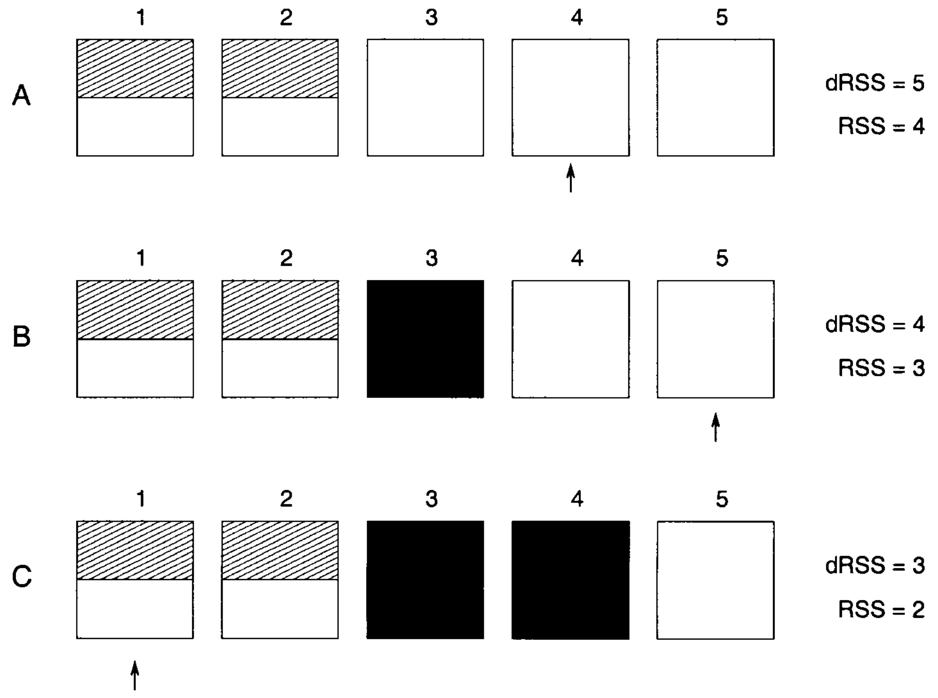


Figure 5.1: Detecting and responding to memory shortage with the naive (and incorrect) "solution". A program runs with 5 panels; each represented with a box at times A, B, and C. White regions of a panel are mapped and resident, while shading represents memory that is mapped but not resident. Black represents a panel that is not mapped. At time A, portions of panels 1 and 2 have been evicted by the operating system, but there is enough memory to keep four panels mapped. Panel 4 is accessed, and since (RSS < dRSS) holds, Panels_in is decreased by one panel (since diff = dRSS - RSS = 1) to the new value of 4 by evicting the MRU panel 3. Because panel 3 was fully resident, unmapping it causes RSS to reduce even further by exactly Panel_size. At time B, we access panel 5. The condition (RSS < dRSS) still holds, so we unmap the MRU panel 4—even though there is actually enough memory to keep four panels mapped. At time C, we access panel 1. Again, RSS < dRSS, so panel 5 will be unmapped, further reducing RSS by Panel_size. This pattern will continue until all but one panel has been unmapped, despite the availability of enough memory to cache four panels. Clearly this is undesirable behavior!

The problem responsible for the cascade of unnecessary mappings described above is that dRSS is not an accurate lower bound (it is too high) on what RSS can be without indicating a memory shortage. To fix this we introduce the variable lastRSS, which tracks the value of RSS at the last panel access. When a panel is unmapped due to memory shortage (i.e., when Panels_in is reduced), lastRSS is set to the value of RSS immediately after the eviction. This accounts for the fact that `RSS` may drop below the new `dRSS`. At panel accesses that are not accompanied by a decrease in `Panels_in`, if `RSS` > `lastRSS` then we set `lastRSS` = `RSS`. Maintaining `lastRSS` in this fashion, we can conclude that there is a memory shortage whenever `RSS` < `lastRSS`. Figure 5.2 summarizes the test for memory shortage, and Figure 5.3 illustrates its use. Note that no explicit reference to `dRSS` is required: If there is enough memory to keep `Panels_in` panels resident, then `lastRSS` will eventually grow to match `dRSS` once all pages evicted by the operating system during the last memory shortage have been touched again.

```
Algorithm: Detect shortage

RSS = Get current RSS
if ( RSS < lastRSS )
    diff = (lastRSS-RSS) / Panel_size
    Unmap diff panels
    Panels_in = Panels_in − diff,  dRSS = dRSS − diff * Panel_size
endif
lastRSS = Get current RSS
```

Figure 5.2: The algorithm for detecting and responding to memory shortage.

We note that some operating systems employ a page-fault frequency (PFF) strategy for preventing thrashing; Windows NT is an example. An OS employing a PFF strategy will sometimes reclaim pages from a process under no memory pressure. If the page-fault rate of the process falls below a certain threshold, the OS removes page frames from the process; if the process then increases its page fault rate, the OS will give frames back. The idea is to prevent thrashing by ensuring that processes do not consume more memory than they require. Our algorithm may interpret this sort of probing by the OS as memory shortage and evict a panel. The page-fault frequency of the program will increase, prompting the OS to allocate more page frames. `Panels_in` will eventually return to its original level, but some performance penalty will have been incurred by the unnecessary eviction. We believe this problem could be eliminated by requiring that (`lastRSS` − `RSS`) exceed a threshold before action is taken, but we have not yet had the opportunity to test our algorithm under

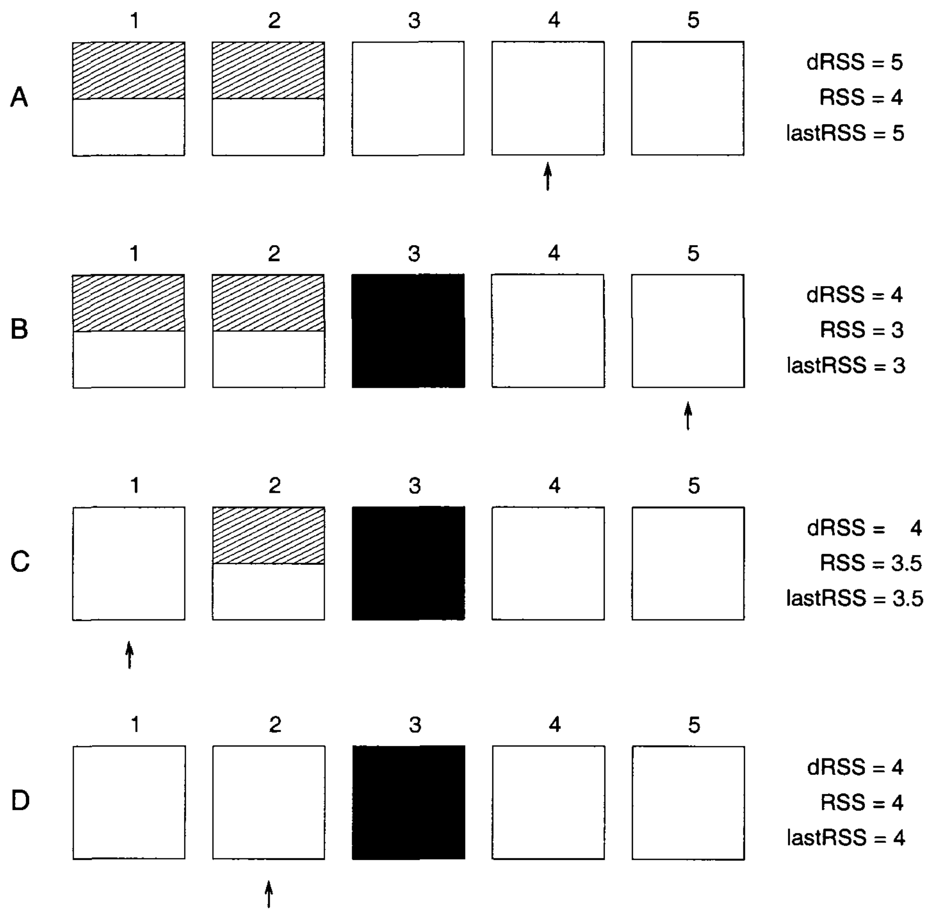


Figure 5.3: An example of detecting and responding to memory shortage using the algorithm in Figure 5.2. A program runs with 5 panels; each represented with a box at times A, B, and C, D. White regions of a panel are mapped and resident, while shading represents memory that is mapped but not resident. Black represents a panel that is not mapped. At time A, portions of panels 1 and 2 have been evicted by the operating system, but there is enough memory to keep four panels mapped. We access panel 4. RSS < `lastRSS`, so we unmap MRU panel 3. At time B, we access panel 5. RSS < dRSS, but RSS = `lastRSS`, so we don't unmap. At time C, we access panel 1. By touching panel 1, the portion of it that was swapped out by the OS is brought back into memory. Now RSS = 3.5, and `lastRSS` < RSS so we set `lastRSS` = RSS = 3.5. At time D, we access panel 2, bringing its contents fully into memory and increasing RSS to 4. We set `lastRSS` = RSS = 4. Note that now `lastRSS` matches dRSS.

a PFF system.

5.3.2 Detecting memory surplus

Because the operating system provides no mechanism for determining memory availability, we must employ an invasive approach. We periodically probe the system, attempting to increase memory usage by one panel. If enough memory is available, `RSS` should grow by one panel. If memory is not available, then `RSS` will remain constant, or decrease as the operating system responds to memory pressure by evicting pages.

The question is **when should we probe?** We should not probe for more memory if `RSS` < `dRSS`. This condition indicates that parts of mapped panels have been paged out by the system. If memory is available, `RSS` will grow as panels are touched and pages are brought back into memory. When `RSS` = `dRSS`, and if there are additional panels to map, then we may probe, performing the next mapping of a panel without replacement. If the new `dRSS` cannot be maintained, `RSS` will eventually decrease below `lastRSS` and the Detect Shortage algorithm will take memory usage back to a safer level.

5.3.2.1 Problems with overly aggressive probing

The simplest policy is to attempt to increase `Panels_in` whenever `RSS` = `dRSS`. This policy is too aggressive, however. It continually pushes `Panels_in` above a safe level, incurring a significant performance penalty each time this happens. Figure 5.4a depicts experimental results that illustrate this. In the experiment, there is room to keep 40% of the panels in memory. Our program is able to temporarily obtain enough memory to hold up to 60% of the panels. Quickly, however, the operating system senses a memory shortage and begins reclaiming pages from the program, sometimes reducing `RSS` significantly below 40% of the panels. The program adapts by decreasing `Panels_in` back to the safe value of 40%. Eventually all mapped pages come back into resident memory, and the cycle repeats. Continual page reclamation by the OS keeps average memory utilization of the program significantly below the desired 40%.

5.3.2.2 Balancing performance penalties with a dynamic probing policy

We can reduce the aggressiveness of our policy by delaying growth of `Panels_in` for a time after it has been reduced by the Detect Shortage algorithm. Choosing an appropriate delay is a balancing act between two sources of performance penalties. If a probe is unsuccessful, this induces what we call an "incursion" penalty because it will induce paging and a subsequent performance decrease. On the other hand, if the program's memory usage stays below the amount of memory available, it suffers an "inaction" penalty because some panels will be loaded from secondary storage when they could instead reside in main memory. We assume a simple model in which the time T to fetch M words from disk depends only on the bandwidth B_w of the disk. Despite its simplicity, the model is appropriate in our case because we access large, contiguous blocks of data; seek times are largely hidden by prefetching. Define `maxRSS` to be the maximum amount of memory currently available to our program. If the program stays at `RSS`, then for each iteration, or cycle through all panels, (`maxRSS` - `RSS`) of data which could have been kept in-core will be brought from disk, incurring an inaction penalty of (`maxRSS` - `RSS`)$/B_w$ seconds. If the program probes beyond `maxRSS`, the operating system responds by decreasing `RSS`. As figure 5.4a shows, in the worst case `Panels_in` may be reduced all the way down to 1. The incursion penalty then is roughly (`maxRSS`$/B_w$), because all of the evicted panels will have to be brought back in.

We attempt to choose a delay that balances the two penalties. This suggests that we consider the quantity

$$R_{pen} = \text{maxRSS}/(\text{maxRSS - RSS}),$$

which is the ratio of the incursion and inaction penalties. When `RSS` is zero, the inaction penalty is as great as the incursion penalty, so we have nothing to lose by probing for more memory; thus when $R_{pen} = 1$ we should probe as soon as possible. When the ratio is greater

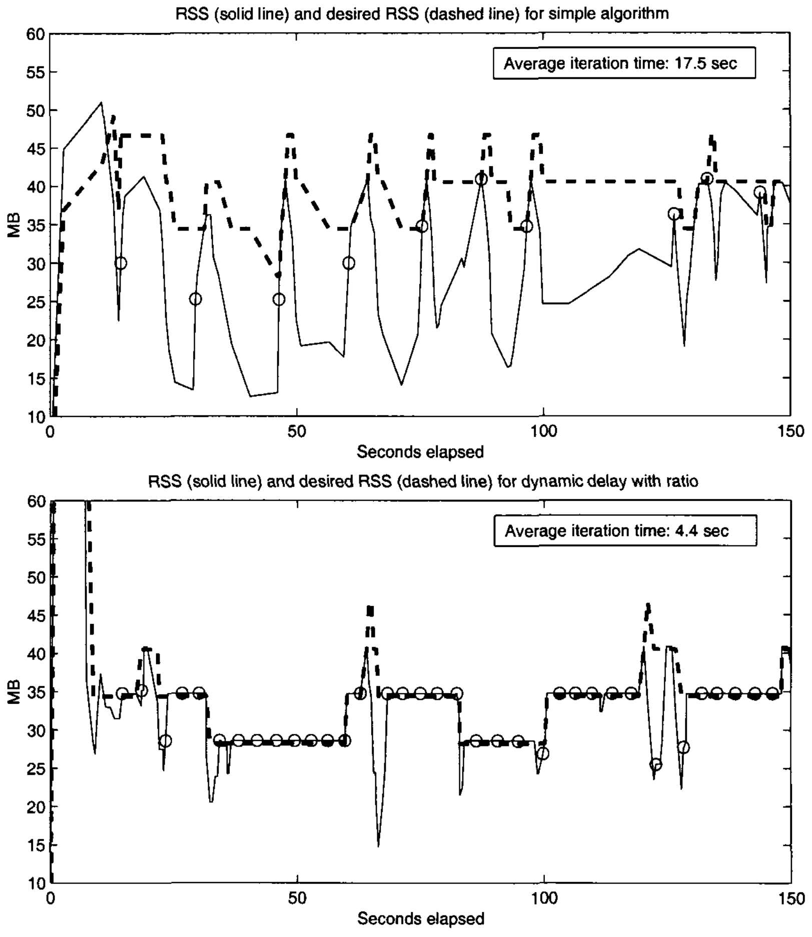


Figure 5.4: Graphs depicting the necessity of a dynamic delay parameter in the algorithm for detecting memory availability. RSS (solid line) and dRSS (dashed line) versus time are shown for two versions of a memory-adaptive program that performs dense matrix-vector multiplications with a 70 MB matrix broken into 10 panels, each consisting of consecutive rows. Circles denote the completion of a matrix-vector multiplication. The program runs against a 70 MB dummy job on a 1 GHz Pentium III machine with 128 MB of DRAM and an IDE hard disk. The top graph (a) uses the original adaptation algorithm with no delay. It is too aggressive, continually pushing against the memory limit. In response, the operating system evicts pages from the program, causing a significant performance penalty. The bottom graph (b) utilizes a dynamically determined delay to reduce this penalty: after a memory shortage is detected, attempts to grow memory usage must wait until the delay has elapsed. Using the dynamic delay, the algorithm settles at what is close to the optimal value for dRSS (dashed line) and diminishes RSS (solid line) fluctuations.

than unity, it indicates that the possible incursion penalty outweighs the possible inaction penalty by that ratio; thus suggests we should wait R_{pen} times as long as we would in the $R_{pen} = 1$ case before probing. Given a base delay time, then, we can scale it by R_{pen} to determine the delay:

delay = (base delay) * maxRSS/(maxRSS - RSS)

We have noted that when RSS is close to 0, we should probe for memory as soon as possible. Since we never probe unless RSS = dRSS, after Detect Shortage causes an unmapping the program may have to wait for a full iteration (cycle through the panels) for RSS to grow to dRSS. Thus the time for an iteration provides a reasonable approximation for the minimum delay, and thus is a natural value for the base delay. We maintain a queue of timestamps for the last P panel accesses to determine the base delay. This delay is dynamic in character, constantly changing to reflect system conditions. For instance, if a competing memory-intensive job exits, the delay will decrease as the system uses the extra space to hold more panels in the buffer cache; as a result, a memory-adaptive job will decrease the time it waits to probe for more memory.

Unfortunately, we do not know maxRSS—if we did, we wouldn't need the algorithm for detecting memory surplus—so we must approximate it somehow. We use peakRSS, which is the most recently observed peak in RSS. To determine it, we introduce the variable max_since_last_surplus. This variable assumes the value of zero upon initialization. Then it is updated in the following manner: Whenever a panel is fetched, max_since_last_surplus is set to max(max_since_last_surplus, RSS). Then, if shortage is detected, peakRSS is assigned the current value of max_since_last_surplus. Otherwise, if surplus is detected (i.e., if the algorithm decides to probe for more memory) we set max_since_last_surplus to the current value of RSS. This simple scheme, illustrated in Figure 5.5, effectively identifies the most recent peak in the RSS versus time curve at negligible expense.

Figure 5.4b demonstrates how the introduction of a dynamic delay parameter dramatically improves performance by greatly diminishing RSS fluctuations while maintaining dRSS

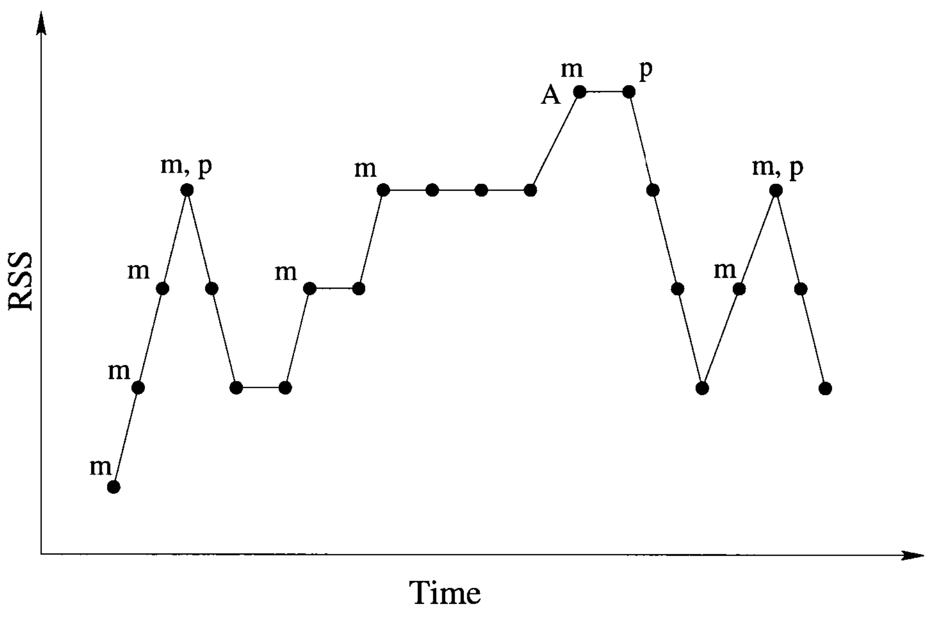


Figure 5.5: An example of how peakRSS is determined. Shown is a cartoon curve depicting RSS versus time. Points where max_since_last_surplus is updated are marked with an "m"; points where peakRSS is updated are marked with a "p". An astute reader may notice that at the point marked A, RSS grows to exceed peakRSS, but because shortage is not yet detected, peakRSS is not updated to match RSS. This is by design; if RSS has grown beyond peakRSS, this means that until the next shortage, probing will no longer be delayed, and the value of peakRSS is therefore irrelevant. The reason will become clear in section 5.4.2, where the calculation of R_{pen} is explained fully.

at close to the optimal value. The combined algorithm for detecting memory shortage and surplus is summarized in figure 5.6.

Algorithm: Adapting to memory variability

```
RSS = Get current RSS
max_since_last_surplus = max(max_since_last_surplus, RSS)

// Check for memory shortage
if ( RSS < lastRSS ) & ( Panels_in > 1 )
    diff = (lastRSS-RSS) / Panel_size
    unmap diff panels,  Panels_in −= diff,  dRSS −= diff * Panel_size
    peakRSS = max_since_last_surplus

// Check for memory surplus
else if ( dRSS == RSS ) & ( Panels_in < $P$ )
    base_delay = Time to access the last $P$ panels
    if (Time since last shortage > base_delay * $R_{pen}$ )
        Panels_in ++
        dRSS += Panel_size
        max_since_last_surplus = RSS
    endif
endif

lastRSS = Get current RSS
```

Figure 5.6: The combined algorithm for detecting and adapting to memory shortage or surplus. It is executed whenever a panel is requested. It is parameter free and requires no system information beyond the resident set size (RSS) of the program, making it highly portable. max_since_last_surplus is initialized to zero. The initial value of peakRSS is immaterial; it is not needed for calculating R_{pen} until after the first shortage is detected, at which point peakRSS will be updated. We have not given the details of how R_{pen} is calculated; that will be explained in section 5.4.2.

5.4 Further details of the adaptation algorithms

The preceding section presented the basic ideas behind the algorithms that enable our memory-malleability approach, but did not discuss some important details. First, we have not explained how the static memory size sRSS, which is required for calculating dRSS, can be determined. Second, we have not fully explained how the ratio R_{pen}—crucial in determining the frequency of probes for surplus memory—is calculated. In fact, R_{pen} can be computed in some different ways, each appropriate for different situations. We fully discuss both of these issues in this section.

5.4.1 Estimating the static memory size

In the description of our algorithm for adapting to memory surplus, we have assumed that the program has an accurate estimate of the size of its static memory, i.e., the memory not associated with objects managed by our memory-malleability library. This information is necessary for calculating the desired RSS and determining the portion of the observed RSS that belongs to the managed objects. Unfortunately, this size may not be easily computed or even available to the program if a large amount of the static memory is allocated within linked, third party libraries. Moreover, for some programs the static memory may fluctuate between different program phases, and even during a pass through the panels. A more serious problem arises when the static memory is not accessed during a computation phase: Under memory pressure, most operating systems will consider the static memory as least recently used and slowly swap it out of DRAM. This can cause a false detection of memory shortage, and the unmapping of as many panels as the size of the swapped static memory.

Our solution to this problem relies on a system call named `mincore()` under Unix-like systems and `VirtualQuery()` under Windows. The call returns information about which pages from a certain memory segment of a process are resident in core. Because our adaptation algorithm has no knowledge of data that it does not manage (i.e., it only knows about the panels), it uses `mincore()` to compute the "managed RSS", `mRSS`, which is the amount of data within all the panels that is actually in-core (resident). This is precisely what is needed—from this we can calculate an accurate estimate of the static memory: `sRSS` = `RSS` − `mRSS`, with `RSS` obtained from the system. With `sRSS`, we can now calculate `dRSS` = `Panels_in` * `Panel_size` + `sRSS`. Furthermore, if we use `lastRSS` to track `mRSS`, instead of `RSS`, we can use `mRSS` to determine if a detected "shortage" is real or simply an artifact of fluctuations in `sRSS`. (We will likewise use `peakRSS` to track the most recent peak in mRSS, rather than RSS, and so on for any other variables that track a resident set size value. We wish to isolate our algorithm from fluctuations in sRSS, because it does not manage the static memory.)

There is a small but non-negligible overhead (see Figure 5.7) associated with checking the residency of the pages of all mapped panels, so using the above technique every time a panel is fetched is impractical. Instead, we adopt a more feasible but equally effective strategy: We measure mRSS (and hence sRSS) at the beginning of each computation phase involving the panels. As long as no memory shortage is detected, we use the calculated sRSS during that phase. When shortage is detected, we recompute mRSS and sRSS to ensure that a detected shortage is actual and not simply due to variation in sRSS. Since shortage is not a frequent occurrence the overall mincore overhead is tiny, especially compared to the slow down the code experiences under memory pressure. In practical terms, reading 128 MB from a 40 MB/s disk takes approximately 3.2 seconds, but it takes only 0.028 seconds to check the residency of a 128 MB region. The overhead of checking the residency of the pages a few times during a computation phase is not a concern.

5.4.2 Calculating R_{pen}

5.4.2.1 A low frequency probing approach

To prevent the adaptation algorithm from probing too aggressively for memory, we scale a base delay (the time for a sweep through the panels) by the ratio R_{pen}, which approximates the ratio of the penalty for unsuccessfully probing and the penalty for failing to utilize all available memory. Ideally we would calculate this ratio as R_{pen} = maxRSS / (maxRSS - mRSS), but because we have no way of knowing maxRSS we instead approximate it with peakRSS. There is a potential problem here, because peakRSS may significantly underestimate the actual value of maxRSS (and it becomes more likely to do so as we move further in time from the last peak). For example, say that peakRSS = 5 panels, but a competing job that was causing memory pressure on the node has recently exited. Our memory adaptive job

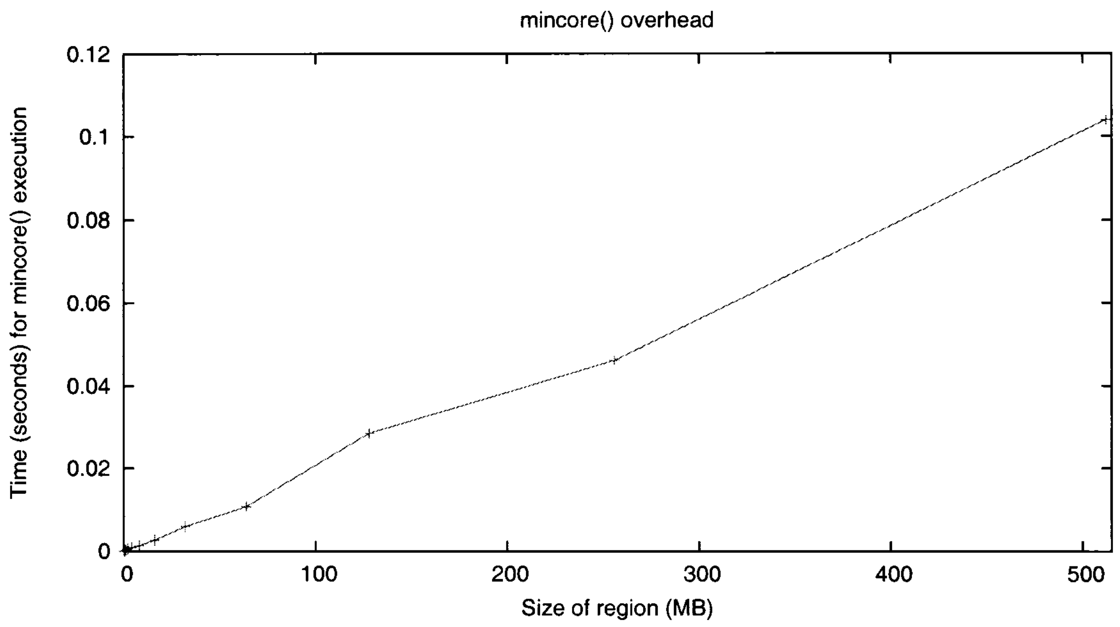


Figure 5.7: Overhead as a function of size of the region upon which `mincore()` is called; the overheads are small but non-negligible. The data pictured were collected on a Sun Enterprise 420R running Solaris 7, and are similar to what is observed on other systems.

begins growing its memory usage, but when it reaches RSS = 5 panels, the denominator in the ratio R_{pen} becomes zero and our calculated delay before probing goes to infinity. This behavior is correct if peakRSS really does equal maxRSS, since then there is nothing to gain from probing for more memory, as RSS is at its maximum. But because peakRSS in general is not equal to maxRSS, we must introduce a mechanism to ensure that RSS does not become forever fixed at the level of the last observed peak. The simplest solution is to set a maximum R_{pen_max} which the value of R_{pen} cannot exceed: that is, we calculate

$$R_{pen} = \min(R_{pen\_max}, \text{peakRSS}/(\text{peakRSS} - \text{RSS})). \tag{5.1}$$

The value of R_{pen_max} should be high enough to prevent overly aggressive probing, but not so high as to cause an unreasonable delay in probing when RSS approaches peakRSS. We have found that once R_{pen_max} becomes sufficiently large, performance becomes relatively insensitive to further increases in its value. This is in part due to the ability of the buffer cache to use available memory to help speed up access to panels that the memory adaptation algorithm does not keep mapped. We have found $R_{pen\_max} = 10$ to work well in practice. In Chapter 7 we present some experiments that examine the choice of R_{pen_max}.

5.4.2.2 Probing with higher frequency

Although the strategy using R_{pen_max} we have just described adjusts to memory availability quite well when external memory load is not highly variable, it has the drawback of being very slow to adjust to large jumps in memory availability: In general, at least R_{pen_max} sweeps will be required, which can be a very long time if a large amount of computation occurs over a sweep. Although we have noted that the buffer cache helps mitigate this problem, better responsiveness to the appearance of surplus memory is desirable because it is better for the memory to be managed by our application-specific policy, rather than the generic OS one. We can achieve better responsiveness by making some changes that reduce the conservativeness of our scheme. The first (relatively minor) change is based on the observation that `maxRSS` (and hence `peakRSS`) usually overestimates the incursion penalty for an unsuccessful probe. We have found that unsuccessful probes typically drop memory usage a sizeable amount, but usually come nowhere near dropping `RSS` to zero unless it is already at a very low level. To remedy this problem, we introduce another variable, `troughRSS`, that tracks the most recent trough or dip in the `RSS` (or `mRSS`, more accurately) versus time curve. `troughRSS` is set to zero at program initialization and is then maintained in the following manner: When a panel is fetched, we update `troughRSS` = min(`troughRSS`, `mRSS`). Whenever memory shortage is detected, `troughRSS` is set to the current value of `mRSS`. Figure 5.8 illustrates how `troughRSS` is maintained. Using `troughRSS`, we now calculate

$$R_{pen} = \min(R_{pen\_max}, (\text{peakRSS} - \text{troughRSS})/(\text{peakRSS} - \text{mRSS}))$$

The introduction of `troughRSS` to better approximate the incursion penalty enables

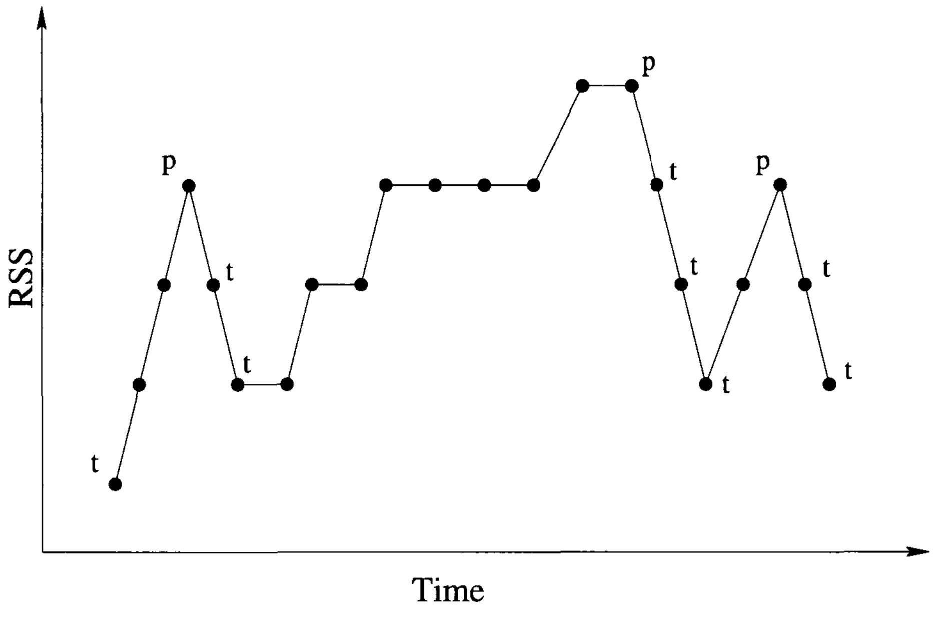


Figure 5.8: An example of how troughRSS is determined. Shown is a cartoon curve (the same as in Figure 5.5) depicting RSS versus time. Points where troughRSS is updated are marked with a "t". Also shown are the points where peakRSS is updated, marked by a "p".

us to respond slightly faster to memory surplus, but in practice this change matters little because we still face the problem of the denominator (peakRSS – RSS) going to zero. To address this problem we employ the following reasoning:

peakRSS is likely a very good approximation to maxRSS immediately after the peak is hit, but as we move further in time from the peak without detecting shortage, the probability that peakRSS is significantly wrong increases. We cannot quantify how this probability increases, but we can say that if RSS becomes close to or exceeds peakRSS and stays there without causing shortage, then it is likely that peakRSS is no longer a good approximation for maxRSS. This is because peakRSS is actually the most recent known point at which the adaptation algorithm probed too high (recall that peakRSS is only updated immediately after a detected shortage). When RSS can be maintained close to or above peakRSS, then, this suggests that the amount of available memory has increased, because the last time RSS hit peakRSS, shortage ensued. Since peakRSS is likely invalid, we replace it by the only other upper bound for maxRSS that we we have: the value mRSS takes if all panels are mapped and resident, which we call `MAXRSS`. So, our heuristic does the following: Once `RSS` and `peakRSS` become very close, we then reset `peakRSS` = `MAXRSS` and recalculate R_{pen}. (We take "very close" to mean that the ratio of penalties evaluates to greater than R_{pen_max}.) When the next shortage is encountered, `peakRSS` will be reset to `max_since_last_surplus`. Note that, using this heuristic, if `RSS` is high (and hence close to `MAXRSS`), then probing will still be quite conservative, but if `RSS` is small, then probing becomes more aggressive. This makes intuitive sense.

We face a dilemma in choosing between the method for calculating R_{pen} we have just described (which we denote method II, or "high frequency probing") and the method shown in equation 5.1 (method I, "low frequency probing"). High frequency probing is indeed quicker to respond to memory surplus, but because it generally results in more frequent probing, it will result in somewhat worse performance than low frequency probing when running under long-lived, static memory pressure (and perhaps under other loads as well). Because which method is most appropriate depends on the characteristics of a system and the workload it experiences, we cannot categorically state that one method is superior to another. We discuss the effects of the choice of methods for calculating R_{pen} in an experimental context in Chapter 7.

We depict our complete algorithm for adapting to memory surplus using low frequency probing in Figure 5.9. All of the details we have discussed in section 5.4 are included. The complete algorithm using high frequency probing is shown in Figure 5.10.

Algorithm: Adapting to memory variability with low frequency probing

```
panel_access_count++
RSS = Get current RSS
if (panel_access_count modulo $P$ == 0) then sRSS = gauge_static_size()
mRSS = RSS − sRSS
dRSS = ( Total size of all mapped panels ) + sRSS
max_since_last_surplus = max(max_since_last_surplus, mRSS)

// Make sure that any "shortage" is not due to variations in sRSS.
if ( mRSS < lastRSS )
    sRSS = gauge_static_size()
    mRSS = RSS − sRSS

// Check for memory shortage
if ( mRSS < lastRSS ) & ( Panels_in > 1 )
    diff = (lastRSS - mRSS) / Panel_size
    unmap diff panels,  Panels_in −= diff,   dRSS −= diff * Panel_size
    peakRSS = max_since_last_surplus

// Check for memory surplus
else if ( dRSS == RSS ) & ( Panels_in < $P$ )
    base_delay = Time to access the last $P$ panels
    $R_{pen}$ = min($R_{pen\_max}$, peakRSS/(peakRSS − mRSS)).
    if ( Shortage has never occurred
        OR Time since last shortage > base_delay * $R_{pen}$ )
         Panels_in++
         dRSS += Panel_size
         max_since_last_surplus = mRSS

RSS = Get current RSS
lastRSS = RSS − sRSS
```

Figure 5.9: The final, complete algorithm for detecting and adapting to memory shortage or surplus, using low frequency probing. It is executed whenever a panel is requested. max_since_last_surplus is initialized to zero. The initial value of peakRSS is immaterial because R_{pen} is not used until after the first shortage is detected, at which point peakRSS will be updated. The gauge_static_size() function uses a system call such as mincore() to calculate the size of the "managed RSS", mRSS, and returns sRSS = RSS − mRSS. Because the overhead of gauge_static_size() means that it cannot be called whenever mRSS is desired, mRSS is sometimes approximated in the algorithm by calculating mRSS = RSS − sRSS using an older value of sRSS.

Algorithm: Adapting to memory variability with high frequency probing

```
panel_access_count++
RSS = Get current RSS
if (panel_access_count modulo $P$ == 0) then sRSS = gauge_static_size()
mRSS = RSS − sRSS
dRSS = ( Total size of all mapped panels ) + sRSS
max_since_last_surplus = max(max_since_last_surplus, mRSS)
troughRSS = min(troughRSS, mRSS)

// Make sure that any "shortage" is not due to variations in sRSS.
if ( mRSS < lastRSS )
    sRSS = gauge_static_size()
    mRSS = RSS − sRSS

// Check for memory shortage
if ( mRSS < lastRSS ) & ( Panels_in > 1 )
    diff = (lastRSS - mRSS) / Panel_size
    unmap diff panels, Panels_in −= diff,   dRSS −= diff * Panel_size
    peakRSS = max_since_last_surplus
    troughRSS = mRSS

// Check for memory surplus
else if ( dRSS == RSS ) & ( Panels_in < $P$ )
    base_delay = Time to access the last $P$ panels
    $R_{pen}$ = (peakRSS − troughRSS)/(peakRSS − mRSS)
    if $R_{pen} > R_{pen\_max}$
        peakRSS = MAXRSS
        $R_{pen}$ = min($R_{pen\_max}$, (peakRSS − troughRSS)/(peakRSS − mRSS)).
    if ( Shortage has never occurred
       OR Time since last shortage > base_delay * $R_{pen}$ )
        Panels_in++
        dRSS += Panel_size
        max_since_last_surplus = mRSS

RSS = Get current RSS
lastRSS = RSS − sRSS
```

Figure 5.10: The final, complete algorithm for detecting and adapting to memory shortage or surplus, using high frequency probing. It is executed whenever a panel is requested. The algorithm is essentially identical to that shown in Figure 5.9, except for the following differences: 1) `troughRSS` is used in addition to `peakRSS` to calculate R_{pen}, and 2) when the ratio used to calculate R_{pen} exceeds R_{pen_max}, `peakRSS` is set to `MAXRSS`, the value that `mRSS` would assume if all panels are fully resident. `max_since_last_surplus` is initialized to zero.

Chapter 6

Design of MMlib: A memory-malleability library

Regardless of its other merits, if our memory-adaptation framework is to see use in application codes, it must be simple to embed in an actual code. To facilitate use of our memory-adaptation strategy, we have developed an object-based C-library, MMLIB, that hides bookkeeping and memory-adaptation decisions from the programmer. We present an overview of the interface design and some of the technical details in this chapter.

6.1 The general framework supported by MMlib

The framework we have described in section 5.1, with repeated, exhaustive passes through one object broken into P panels, is overly simplistic, but served for elucidating our adaptation strategy. In reality, scientific (and other) applications are much more complex, often working with many large memory objects at a time, with various access patterns for each object, sometimes working with only portions of panels, sometimes working persistently with only a few panels, etc. MMLIB is designed around a more general framework that can accommodate these applications; Figure 6.1 depicts this framework.

```
Identify memory objects $M_1, M_2, \ldots, M_k$ needed during this phase
for i = [ Iteration Space for all Objects ]
    for j = [ all Objects needed for iteration i ]
        for panelID = [ all panels in accessPattern($M_j$, i) ]
            panel_ptr = mmlib_get_panel($M_j$, panelID)
        endfor
    endfor
    Work on required panels or subpanels
    for j= [ all Objects needed for iteration i ]
        for panelID = [ all panels in accessPattern($M_j$, i) ]
            if panel panelID was modified
                Write Back(panelID)
            if panel panelID not needed persistently
                mmlib_release_panel($M_j$, panelID)
        endfor
    endfor
endfor
```

Figure 6.1: Extended framework modeling the memory access needs of a wide variety of scientific applications. Note that although write-backs are represented explicitly, in our implementation writes are made to named memory-maps, and thus are handled transparently by the virtual memory system unless the user wishes to force a write-back.

6.2 Data structures

A data set to be managed by MMLIB is broken into a number of panels, for which a backing store is created on disk. Access to panels occur through mmap() system calls executed by MMLIB. Each data set is associated with an MMS ("memory malleability struct") object, which contains all bookkeeping information required for memory-management of the data set, and through which all access to the data set occurs. Because an application may require multiple MMS objects associated with different data sets, and because global knowledge of all managed objects is required to make memory-adaptation decisions, MMLIB maintains a global registry (type MMReg) of all MMS objects.

6.3 Core interface and functionality

Though MMLIB is a complicated piece of software consisting of thousands of lines of code, the core of its interface and functionality can be described by the following few functions:

void mmlib_initialize()

Initializes MMLIB, creating the registry that will track all of the bookkeeping information.

MMS mmlib_new_mmstruct(int type, char *filename, int P, void *context)

Creates a new MMS object of a given type (such as MMLIB_TYPE_MATDENSE for a dense two-dimensional array or MMLIB_TYPE_VECTORS for a one-dimensional array). The filename specifies the name of the backing store, which must already exist. (To allow maximum flexibility, the MMS constructor does not create the backing store itself. We do provide functions to aid in its creation.) `P` is the number of panels into which the data set associated with the MMS will be broken. The final, optional, argument allows the user to provide a pointer to a user-defined context containing private data to be made available to the function implementing the user-specified replacement policy.

void *mmlib_get_panel(MMS mms, int p)

This function is the basic building block of the library. It returns a pointer to the beginning of panel p. Additionally, it transparently handles adaptation decisions, updating the priority queues used to make replacement decisions, checking for memory shortage or surplus, updating the number of panels cached, and performing requisite cache evictions.

void mmlib_release_panel(MMS mms, int p)

A pointer returned by mmlib_get_panel() is guaranteed to remain valid until the panel is released by mmlib_release_panel(). This is necessary because some applications require certain panels to persist throughout the mapping and unmapping of other panels. For example, if an application must compute the interaction of a panel X_m with panels X_i, $i = \{1, ..., m-1\}$, we must ensure that the call to mmlib_get_panel() does not unmap panel X_m when it fetches one of the panels X_i. In this case, the panel X_i must remain "locked" during the computation to ensure its persistence, and afterwards can be released. Note that the release does not evict a panel from the cache; it merely unlocks it so that MMLIB can do so if deemed necessary.

void mmlib_set_update_queue(void (*func) (MMReg, MMS, int))

The MMLIB registry contains a global priority queue that orders the panels of all MMS objects according to the panel-cache eviction policy. When a given amount of space must be freed, the MMLIB eviction function evicts panels according to their ordering in the queue until enough space has been freed. The priority queue is updated each time that mmlib_get_panel() is called, inserting the newly accessed panel in the proper place. mmlib_set_update_queue() allows the user to specify the function that should be called to perform this update and maintain any other data structures that may be required to implement the eviction policy, such as queues local to each MMS object. MMLIB defaults to Most Recently Used (MRU) replacement, as this is suited to the cyclic the access patterns of many scientific applications.

We should note that to provide maximum flexibility, MMLIB also provides an interface for the user to specify the function that performs panel evictions. The preferred method for specifying an eviction policy is to use mmlib_set_update_queue() when possible, however.

6.4 Optimizations and additional functionality

6.4.1 Memory access at sub-panel granularity

Blocking data objects into a set of panels is central to achieving good performance from MMLIB. Fairly large panel sizes are preferred to keep overheads low. However, some programs naturally employ a small granularity of memory access (for example, accessing small vectors instead of a submatrix), and if an application does not already (or cannot) use a blocked access pattern, then injecting memory adaptivity by the get_panel() semantic may be awkward.

More problematically, when get_panel() is called directly by the user, MMLIB has no way of knowing what portion of the panel will be accessed. It can only assume that all of the panel will be touched, which means that the calculated desired resident set size `dRSS` = `Panels_in` * `Panel_size` will be an overestimate if in fact only a portion of the panel is used. Consequently, the program cannot probe for more memory even if all of the pages that are actually in use are resident. This can also cause "shortage" to be detected when in fact none exists!

For example, consider a program with `Panels_in` = 5 and `mRSS` = 5 panels. When the next panel is requested, assume there is no shortage detected, but we do not probe for memory either. At the end of the adaptation algorithm, we set `lastRSS` = `mRSS` = 5. Because we are not probing for more memory, if the requested panel is not already mapped, we will have to evict a panel from the cache to make room. Assume that this is the case. We evict a panel, bringing `mRSS` down to 4 panels, and then map the new panel. Say that only 25% of the new panel is needed. We touch that portion, and then call mmlib_get_panel() to fetch the next panel. There is a problem now: Because we only touched 25% of the new panel, `RSS` = 4.25, but `lastRSS` = 5. The adaptation algorithm will incorrectly conclude that there is memory shortage. Figure 6.2 depicts an example of this problem in the `ISING` application described in Chapter 7.

We solve these problems by introducing some higher-level MMLib functions that serve to decouple the unit of memory access required by the user from the panel size used internally by MMLib. The user still specifies the panel size, but does not have to access the data via mmlib_get_panel() calls. Instead, they can define a smaller unit of data access via the function

void *mmlib_define_unit(MMS mms, int num_units, int N, int data_type)

This function informs MMLib that the specified MMS object is to be broken into `num_units` logical units. Each unit will consist of `N` elements of type `data_type`, where `data_type` is a defined constant such as MMLIB_TYPE_INT, MMLIB_TYPE_DOUBLE, or MMLIB_TYPE_OTHER. The logical units are accessed via the following functions:

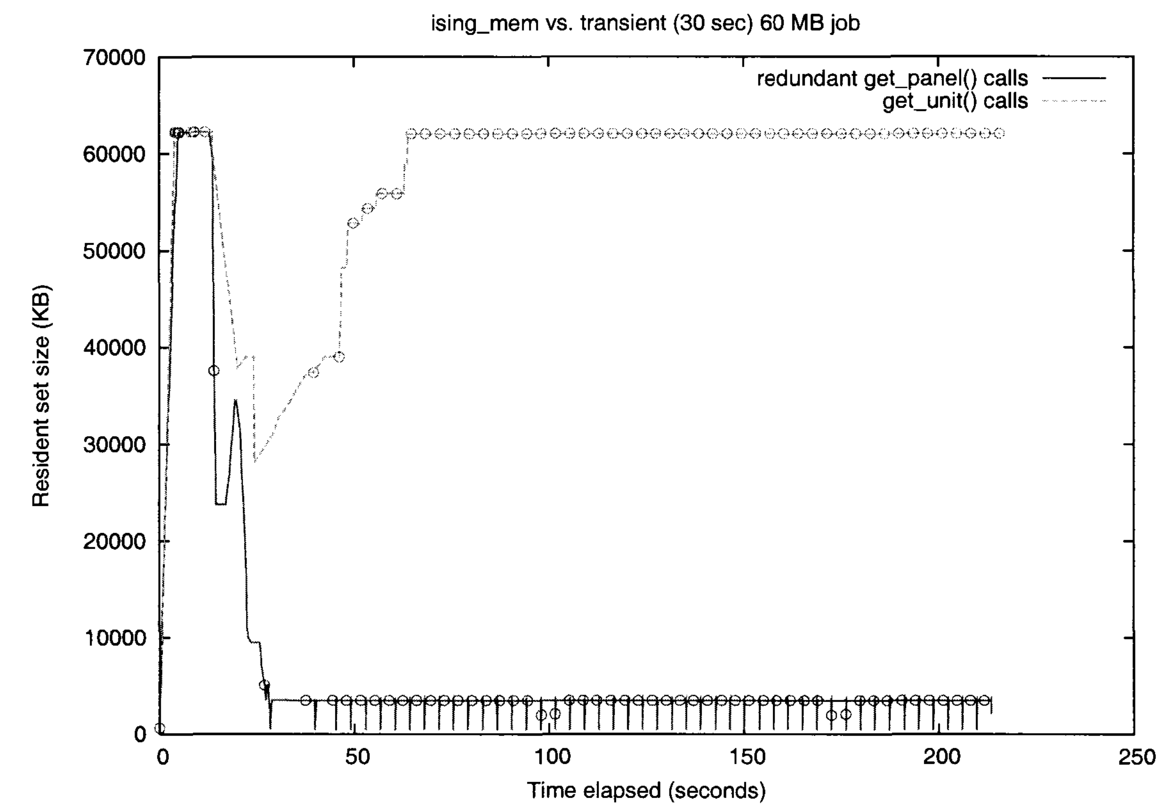


Figure 6.2: Benefits of using smaller memory transfer units: RSS vs. time is shown for two implementations of the ISING code described in Chapter 7. The first (red lower), uses get panel calls without fully using all of the panels, and thus erroneously detects memory shortage and reduces its RSS to a minimum. The second implementation uses `mmlib_get_unit()` calls to access the data in sub-panel sized units and avoids this problem.

void *mmlib_get_unit(MMS mms, int u)

void mmlib_release_unit(MMS mms, int u)

mmlib_get_unit() returns a pointer to unit u and places a lock on that unit (note that locks from multiple gets of the same unit do stack). A pointer to a unit is guaranteed to remain valid until all locks on the unit are released by calls to mmlib_release_unit().

Internally, mmlib_get_unit() calls mmlib_get_panel() when the panel in which u resides is not currently mapped (cached), or when the number of accesses to that parent panel exceeds a given threshold. The threshold criterion ensures that mmlib_get_panel() is called periodically to allow memory adaptation decisions to be made even if all panels are mapped. Without it, once all panels were mapped, memory shortage would never be detected because fetching any unit would not require an mmlib_get_panel() call.

The problems with `dRSS` and false shortages are solved by maintaining an MMLIB registry variable `pages_in`, which tracks the number of pages we are attempting to cache. Whenever a unit is fetched via mmlib_get_unit(), `pages_in` is incremented by the number of pages in that unit; when a panel is evicted, `pages_in` is decremented by the total number of pages in all of the units in that panel that were accessed via mmlib_get_unit(). Maintaining `pages_in` in this manner enables accurate calculation of `dRSS` = `pages_in` * `page_size` even though regions of mapped panels may remain untouched, so long as all of each requested unit is touched. With an accurate value of `dRSS`, the problem of detecting false shortages can also be eliminated. In the example described above, the algorithm will still detect `RSS` < `lastRSS`. However, it will then calculate `mRSS` by checking the residency of all pages in the mapped panels. Only if `mRSS` < `dRSS` will it conclude that the detected "shortage" is real. Figure 6.2 illustrates the effectiveness of this strategy.

6.4.2 Eviction of "most-missing" panels

One of our design goals for MMLIB is to preempt the page replacement policy of the virtual memory system by holding the RSS of an application below the level at which the system begins to swap out its pages. This is not always possible, however, and under high memory pressure the virtual memory system can be quite antagonistic, paging out the data that MMLIB strives to keep resident. In this case, it may be beneficial to "concede defeat" and limit our losses by evicting those panels that have had most of their pages swapped out, rather than evicting according to the application-specific policy, say MRU. The rationale is thus: if the system has evicted LRU pages, we will have to load those pages again when we access the parent panels, regardless of whether we keep those panels mapped or not. Therefore, it is preferable to evict those panels that have been mostly swapped out. The information provided by the `mincore()` calls used to estimate the static memory size (see section 5.4.1) can be used to implement such a "most-missing" eviction policy. This strategy is not at odds with the user-specified replacement policy, as it is not used for routine panel replacements; *it is only used to reduce the panel cache size when shortage is detected* (and antagonism with the VM system policy can occur). In fact, following the user-specified policy and evicting MRU panels in this situation would make things worse: we would have to load the LRU pages that have been paged out, as well as the MRU pages that were resident until we evicted their panels.

We have implemented support for the most-missing eviction strategy in MMLIB (a user-set flag specifies whether or not this policy should be used). Some results indicate that the strategy can have clear benefits. In the scenario shown in Figure 6.3, the job that uses strict MRU eviction exhibits very slow performance initially after external memory pressure begins, because it must load the pages evicted by the operating system as well as the MRU panels that MMLIB has evicted, which usually do not coincide with the panels from which the system has taken pages. The job that employs the "most-missing" policy adapts more nimbly to the sudden increase in memory pressure, because it does not make the mistake of automatically throwing out many panels that have been untouched by the VM system. Note that after the application adapts to the sudden decrease in available memory, it automatically reverts from most-missing to MRU replacement when no further shortage is detected.

Although the most-missing policy does show promise, we do not use it in the experiments we report in Chapter 7. The chief reason is that we do not wish to further complicate the behavior of MMLIB in our experiments; it is difficult enough to analyze its behavior without introducing this additional complexity.

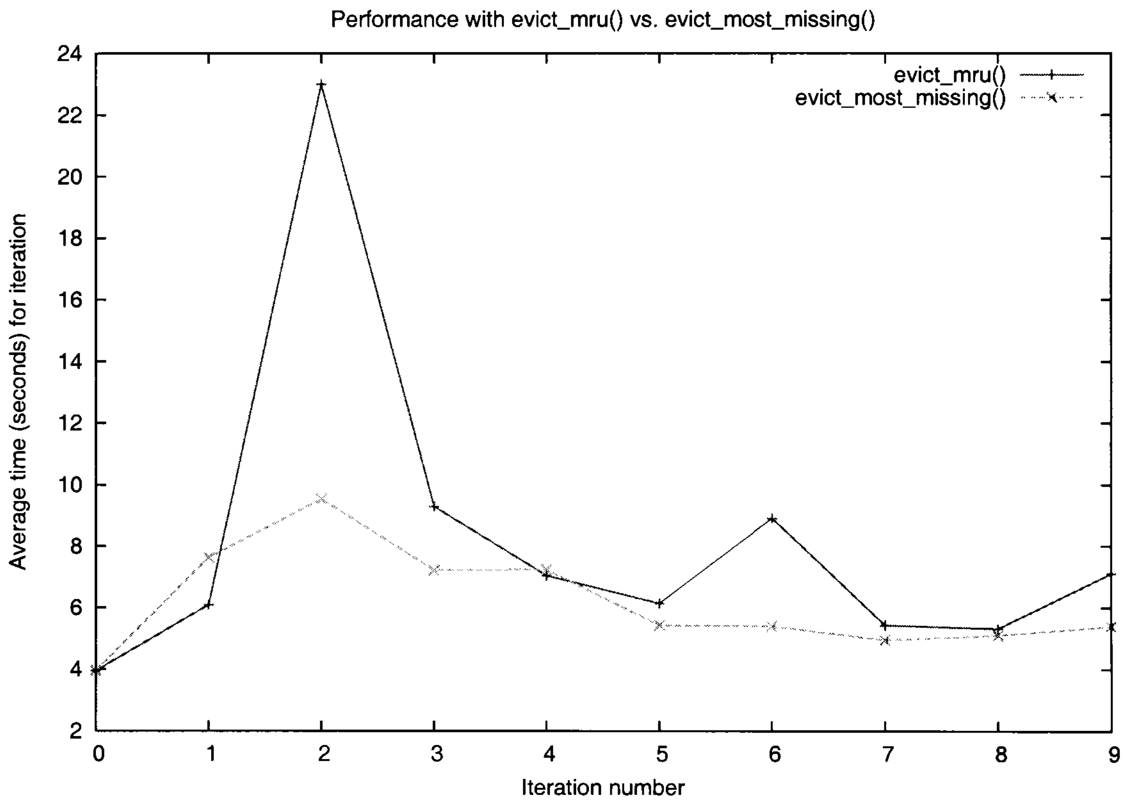


Figure 6.3: Benefits of evicting partially swapped out panels. The chart shows the times for the first 10 sweeps of two versions of the memory-adaptive `ISING` code described in Chapter 7. (Note that sweep 0 is an initialization sweep that runs more quickly than subsequent sweeps, which perform Monte-Carlo steps.) In each case, the code runs with a 60 MB lattice on a Linux machine with 128 MB of RAM. After 12 seconds, a dummy job that randomly writes to 60 MB of memory begins. The memory-adaptive job that evicts panels with the largest number of missing pages, labeled `evict_most_missing`, has lower and less variable times than the job employing strict MRU replacement, labeled `evict_mru`. For both cases, the number of panels $P = 40$ and low frequency probing is used with $R_{pen\_max} = 10$.

Chapter 7

Experimental evaluation of memory-adaptation

7.1 Application kernels

We have used MMLIB to test our memory adaptation technique in three scientific application kernels which together represent a range of scientific applications. All of these application kernels exhibit memory access patterns for which MRU replacement is most appropriate, putting them at odds with the LRU-like policy employed by the operating system. (See also section 7.7, where we modify the `CG` code to use an LRU-friendly pattern.) In other respects, however, the memory access patterns differ between the applications. In this section, we describe these applications and their memory access characteristics.

7.1.1 Conjugate Gradient (`CG`)

The first application is a conjugate gradient (CG) linear system solver using the CG routine provided in SPARSKIT [74]. The computational structure of CG is common among many iterative methods, and these methods constitute the basic kernel in a wide range of scientific applications. (This is why CG is one of the SPEC benchmarks.) Each iteration requires calculation of one sparse matrix-vector product and a few inner products and vector updates. The bulk of the memory demand comes from the sparse coefficient matrix, which we manage with MMLIB. CG also requires storage for four work vectors plus the solution and right-hand side vectors. In principal, these vectors can be managed with MMLIB, and in fact this might be necessary if large and very sparse matrices are used, in which case the vectors make up a greater portion of the memory requirements. In our experiments, however, we do not manage these vectors with MMLIB (so they are "static" memory) as they occupy far less memory than the matrices we use. Still, the size of the work vectors is significant, as they represent around 13.5% of the storage requirements in our CG experiments using a 70 MB matrix.

Our test code, `CG`, does not construct a matrix, but loads from disk a pre-generated sparse matrix in diagonal format. Figure 7.1 depicts the algorithm for matrix-vector multiplication of a sparse matrix in this format; this is the algorithm in which MMLIB will be used to enable memory adaptivity. The matrices used in our experiments are generated from a three-dimensional, eighth order finite-difference discretization of the Laplacian operator on the unit cube using a regular grid and Dirichlet boundary conditions. In the memory-adaptive code, they are partitioned row-wise into panels of consecutive rows. Matrix-vector multiplications sweep through each panel in typewriter fashion, left to right and top to bottom. We note that, as far as MMLIB is concerned, `CG` is a read-only application: writes do not occur to the matrix managed by the library.

7.1.2 Modified Gram-Schmidt (`MGS`)

The second application is a code that progressively builds a set of orthonormal vectors. It uses the modified Gram-Schmidt (MGS) procedure to orthogonalize the vectors, a very common kernel in scientific computing. An example of a very memory demanding application for MGS comes from materials science, where large eigenvalue problems of dimension on the order of one million must be solved for about 500–2000 eigenvectors [82]. For such problems, the storage and orthogonalization of the eigenvectors are the limiting computational factors. MGS sees very common use in implementations of the GMRES algorithm [72], which progressively builds a vector basis from which solutions are extracted. At each step,

```
Algorithm: Diagonal format sparse matrix-vector multiplication

current = 1
for row = 1 to N do
    y[row] = 0
    for i = 1 to nonzeros_per_row do
        col = row + offset[i]
        if (0 < col < N) then
            y[row] = y[row] + A[current] * x[col]
        endif
        current = current + 1
    enddo
enddo
```

Figure 7.1: Matrix-vector multiplication algorithm for a sparse matrix of dimension N consisting of a number of diagonals. x is the input vector and y is the output vector. The array A[] consists of the elements from the first row, followed by the elements from the seconds row, and so on. Note that all rows consume the same number of entries in A[], so some entries will not be used: for example, the first row of the matrix does not contain any elements from diagonals below the main diagonal, so some empty elements will be "stored" in A[]. The offset[] array stores the offset of each of the diagonals with respect to the main diagonal.

an additional basis vector is generated, which must be orthonormalized against the other basis vectors. If the vectors are very large (as required to solve PDEs on high resolution meshes, for instance) or if many vectors must be retained to ensure convergence, then the memory requirements for the MGS orthogonalization may make up the bulk of the memory demand. Our test code, `MGS`, simulates the behavior of a GMRES-type solver (such as Jacobi-Davidson), though it generates new vectors randomly rather than through a matrix-vector multiplication, because our goal is to focus solely on the memory demands imposed by the vectors. At each step, a new vector is generated, orthogonalized against previously generated vectors, normalized, and then appended to them. The code allows a "restart size" *max_basis_size* to be specified: that is, once the basis has grown to *max_basis_size* vectors, it discards all but *min_basis_size* vectors from the basis and begins building a new set. Restarting is commonly employed with GMRES and related solvers because as the basis grows, memory and computational costs may become prohibitive. A remedy is to restart the algorithm, retaining the current approximate solution vector and discarding the basis vectors. We note that MGS is the only one of our test codes whose memory requirements vary considerably throughout its lifetime, as the basis grows or is discarded. Figure 7.2 depicts the algorithm executed by the MGS code.

Algorithm: MGS test code

```
for j=1 to min_basis_size do
    $\mathbf{v}_j$ = random(N)
enddo

for restart = 0 to num_restarts do
    for j = min_basis_size to max_basis_size do
        $\mathbf{w}$ = random(N)
        for i = 1 to j do  // The MGS orthogonalization
            $h = \mathbf{w} \cdot \mathbf{v}_i$
            $\mathbf{w} = \mathbf{w} - h \cdot \mathbf{v}_i$
        enddo
        $\mathbf{v}_{j+1} = \mathbf{w}/\|\mathbf{w}\|_2$
    enddo
enddo
```

Figure 7.2: The algorithm executed by our MGS test code, which simulates the behavior of a GMRES-type solver, generating random vectors of dimension N which are added to an orthonormal basis via modified Gram-Schmidt. After the basis size grows to a set maximum, the basis is discarded and the computation is "restarted". To ensure that a minimum level of memory pressure is maintained, one can specify a minimum basis size, below which the size of the basis never drops.

7.1.3 Monte-Carlo Ising model simulation (ISING)

Our third application is a Monte-Carlo simulation of the two-dimensional Ising model [33, pp. 550–556], which is used to model magnetism in ferromagnetic materials, liquid-gas transitions, and other similar physical processes. We chose the Ising model for several reasons: Though remarkably simple, it captures many aspects of phase-transitions, and after being introduced in 1928 it is still studied by physicists today. Computationally, it is very similar to a great variety of lattice "spin" models used to study phenomena as diverse as molecular-scale fluid flow and order-disorder transitions in binary alloys [33, pp. 589–593]. And because laboratory-scale systems consist of at least 10^{18} spins, people are interested in studying very large lattices, which puts memory at a premium.

The ISING code works with a two-dimensional lattice of atoms, each of which has a spin of either up (1) or down (-1). It uses the Metropolis algorithm [33, pp. 566–568] to generate a series of configurations that represent thermal equilibrium. The memory accesses are based on a simple 5-stencil computation common to many other scientific applications. Figure 7.3 presents a pseudocode summary of the operation of our ISING code. For each iteration the code sweeps through the lattice and each lattice site is tested to determine whether its spin should flip. If the flip will cause a negative potential energy change, ΔE, the flip is automatically accepted. Otherwise, the spin flips with a probability equal to $\exp \frac{-\Delta E}{kT}$, where k is the Boltzmann constant and T is the ambient temperature. Note that this means that the higher the temperature, the more spins are flipped (equivalent to a "melting" of magnetization or evaporation of liquids). In computational terms, T determines the frequency of writes to the lattice sites at every iteration. The memory-adaptive version of the application partitions the lattice row-wise into panels to simplify the panel access pattern, although other partitionings could also be easily implemented. To calculate the energy change at panel boundaries, the code needs the last row of the above neighboring panel and the first row of the below neighboring panel. Note that unlike CG, which performs no writes to the panels, and MGS, which writes only when a vector is added to the basis, ISING performs frequent writes when higher values of T are used. Also, ISING is the only code that requires multiple panels to be active simultaneously, so that interactions across panel boundaries can be computed.

We should note that the version of the ISING code we use here is written in a manner that allows us to avoid the problems with sub-panel access size shown in Figure 6.2 of the preceding chapter. This involves a bit of awkward bookkeeping to ensure that we do not release panels before using all of them. The code can be written in a much more natural manner (with very little impact on performance) using the MMLIB functions for fetching sub-panel sized units, but we choose to avoid that in our experiments here to avoid complicating our analysis of the performance of MMLIB.

```
Algorithm: Metropolis Ising model sweep

for row = 1 to L do
    for col = 1 to L do
        up = spin[i-1, j]; down = spin[i+1, j]
        left = spin[i, j-1]; right = spin[i, j+1]
        $\Delta E = 2 \cdot$ spin[i, j] $\cdot$ (left + right + up + down)
        if random() $\leq$ w[$\Delta E$ + 8] then
            spin[i, j] = -spin[i, j]
            $E = E + \Delta E$
            $M = M + 2 \cdot$ spin[i,j]
        endif
    enddo
enddo
```

Figure 7.3: The algorithm for executing a Metropolis sweep through the LxL spin lattice of the Ising model. Sites in the lattice possess either spin up (+1) or down (-1). Periodic boundary conditions are used to calculate the spins (up, down, left, right) of the four nearest neighbors. The array w[] is a lookup table of Boltzmann probability ratios; these ratios are dependent on the ambient temperature in the simulation. The total energy E and the magnetization M are scalar quantities that track some macroscopic observables of interest; they do not factor into the computations. Note that for each lattice site, we always generate a random number to determine whether the spin should flip. This could actually be avoided by automatically accepting a spin flip whenever $\Delta E < 0$, but by always generating the random number we ensure that the amount of computation is the same at any temperature. This allows us to ensure that performance differences observed at different temperatures are solely due to differences in frequency of writes to memory.

7.2 Experimental environments and methodology

The experimental results we present were obtained using two different Unix/Unix-like operating systems, Linux 2.4 and Solaris 9, which both employ global, LRU-like page replacement algorithms. Experiments under Linux were performed using Linux 2.4.22-xfs on a 1.2 GHz AMD Duron system with a 30GB, 5400 rpm IDE hard disk, 128 MB of RAM (some of which is shared with the video subsystem), and a 128 KB L1 and a 64 KB L2 cache. The Linux system ran the KDE graphical desktop environment during the experiments. Results for Solaris were obtained using the `typhoon` nodes of the SciClone cluster at William & Mary (see Figure 3.2). The `typhoon` nodes are Sun Ultra 5 machines with a 333 MHz UltraSparc IIi processor, with 256 MB of DRAM, 2MB of external cache, and a 9.1 GB, 7200 rpm EIDE internal hard disk. Access to the nodes occurs through a PBS batch system; no memory is consumed by running a graphical user-interface. We note that, by current standards, the amount of memory in both our Linux and Solaris machines is small. However, running some experiments on a Mac OS X 10.1 system with 768 MB of RAM using 500 MB memory-adaptive jobs yielded results qualitatively and quantitatively similar to those obtained on our smaller Linux and Solaris systems, suggesting that the memory-adaptation strategy scales well to larger memory sizes. The smaller available memories in the systems we use here have the benefit of allowing us to perform experiments under high levels of memory pressure in a reasonable time frame.

We induce memory pressure in our experiments in a variety of ways, but when we wish to report statistics, we have mostly used our `memlock` code running on our Linux system (on which we have the root access privileges required by the code). `memlock` uses the `mlock()` system call to pin pages in-core, allowing us to precisely control the amount of memory pressure experienced by an application. This level of control is important because when memory resources are oversubscribed by multiple jobs, the behavior of the virtual memory system can be highly chaotic and unpredictable. For example, Figure 7.4 shows results from two runs of the same experiment on a Linux 2.4 system with 128 MB DRAM, in which a memory-adaptive `CG` job runs with a 70 MB matrix against a dummy job that allocates 70 MB and repeatedly accesses random pages. The dummy job starts first. In one instance the dummy job keeps its entire working set and `CG` utilizes the remaining memory; in the other, the system eventually gives `CG` enough memory to cache its entire working set. Both runs were conducted under identical settings, but our `CG` code experiences completely different levels of memory pressure in the two instances, although in both cases overall CPU utilization remains high and memory-adaptive `CG` makes good progress. We have found no way to predict or control which case `CG` experiences, and this makes compiling meaningful performance statistics very difficult.

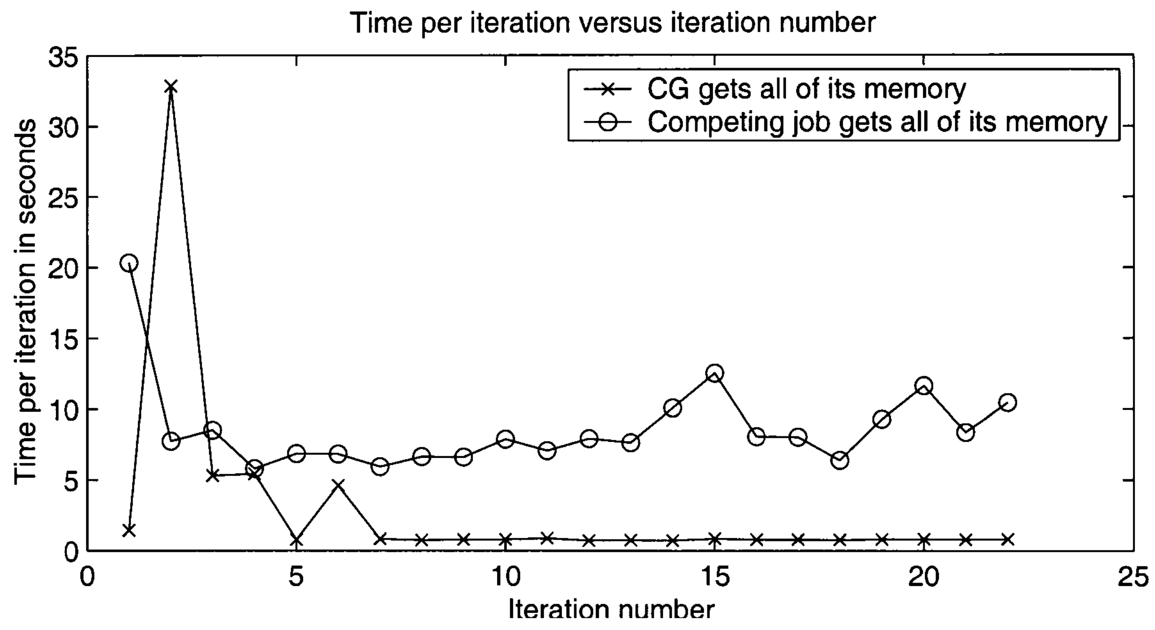


Figure 7.4: Two different modes observed under the same experimental settings. Execution times for each of the first 25 iterations of memory-adaptive CG show two possible modes of behavior: in the first mode, the external load keeps its entire working set while CG utilizes the remaining memory. In the second mode, the scenario is reversed. Resource utilization is high in either case.

Using `memlock` to precisely control the amount of memory available to an application simplifies our statistical analysis, although even when we use `memlock` the variance in the distribution of the times to complete an iteration or sweep through the panels is quite high. As figure 7.5 illustrates, these distributions tend to have a significant right skew, with some iterations taking a very long time to complete. Because of this skewness, when we wish to report confidence intervals for the average time per iteration for a program under given conditions, we run the program under those conditions for 50 iterations and measure the time for each iteration; those 50 measurements comprise one sample. We repeat the run of 50 iterations multiple times to obtain multiple samples, and then calculate the average iteration time as the arithmetic mean of the arithmetic means of the samples. The sample size of 50 is more than enough to ensure normality of the distribution of sample means, allowing us to utilize the Central Limit Theorem and Student's t-distribution to compute 95% confidence intervals.

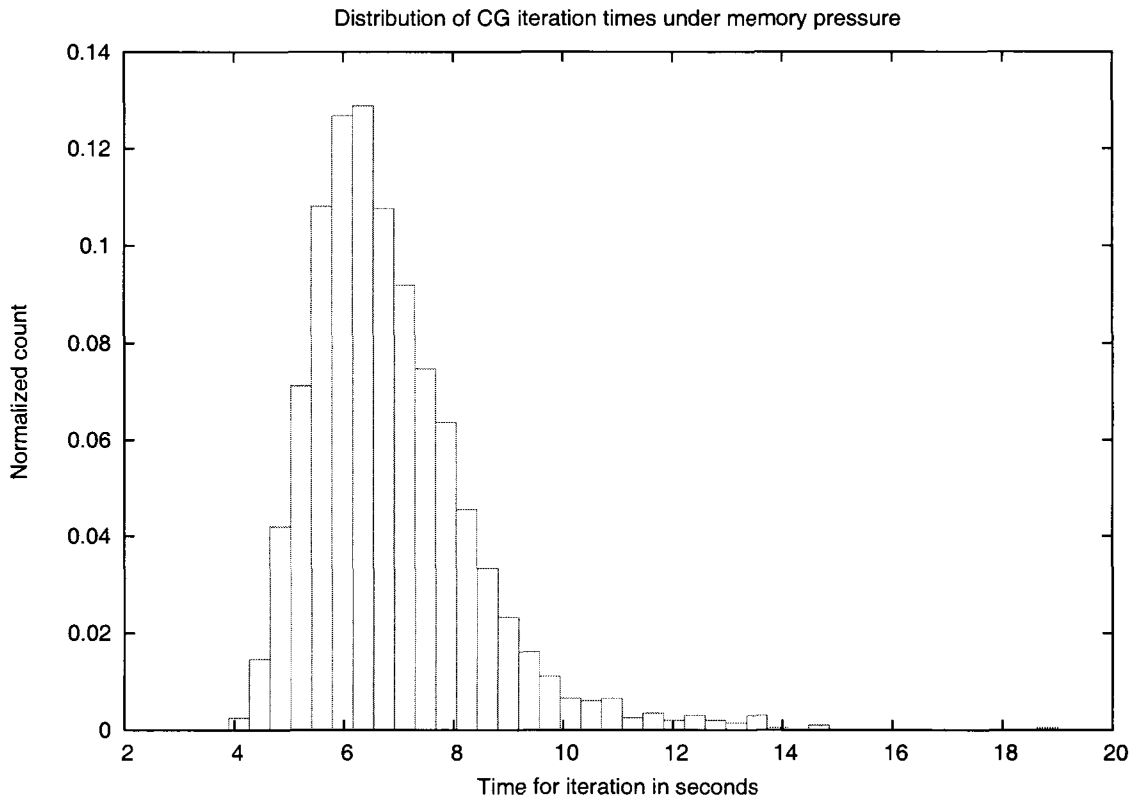


Figure 7.5: Relative frequencies of iteration times for memory-adaptive CG running on a Linux 2.4 machine with 128 MB RAM against a `memlock` job consuming 70 MB. The distribution displays significant right-skewness, with iterations occasionally taking a very long time to complete.

7.3 Choosing the MMlib parameters

We have endeavored to keep MMLIB as free as possible of parameters that must be tuned. The only quantities that must be specified are the panel size and the quantity R_{pen_max} that limits the delay between detection of memory shortage and probing for more memory. Additionally, one must choose between the two types of probing (low or high frequency) discussed in section 5.4.2. In this section we discuss the selection of appropriate values for these parameters and experimentally study the sensitivity of MMLIB to the choice of parameters. Under different workloads than those used in our experiments, the optimal choice of parameters might differ. However, because our experiments reveal MMLIB to be fairly insensitive to the choice of parameters, we think that following the guidelines we present here will result in reasonable (i.e., not far from optimal) performance under a variety of workloads.

7.3.1 Panel size

The choice of panel size involves a trade-off: If a data set is broken into a large number of panels, MMLIB can more exactly tune the level of memory usage, and can cope with higher levels of memory pressure. On the other hand, using many panels results in higher bookkeeping and I/O overheads. Figure 7.6 depicts the performance of memory-adaptive `CG` against different levels of memory pressure, using different values of P, the number of panels into which the 70 MB data set is broken. Figure 7.7 depicts a similar performance curve for memory-adaptive `ISING`. Constant memory pressure is applied via our `memlock` code, which runs as `root` and uses the `mlock()` system call to lock a portion of memory in-core, disabling paging for that region. The performance curves for `CG` show that the choice of $P = 20$ (3.5 MB panels) provides the best performance. With $P = 10$, the overhead associated with managing panels is small, but overall performance is worse because of inability to finely adapt the level of memory usage. Also notice that at the highest level of memory pressure, the $P = 10$ case is markedly worse because the panel size is too large to cope with such a small amount of available memory. For $P = 40$, performance worsens because of increased panel-management overheads.

The performance trends for `ISING` are similar, except that the $P = 20$ curves and $P = 40$ curves are essentially identical except under the highest level of memory pressure, where the larger panels of the $P = 20$ case become a bit too large to allow `ISING` to reduce its memory usage to the optimal amount. We believe that the extra overhead required to manage 40, rather than 20 panels, is not noticeable in `ISING` because the amount of computation per sweep through the panels is relatively high: under no memory pressure, `ISING` requires about 4.2 seconds for a sweep through the panels, compared to about 0.4 seconds for `CG`. Choosing too large a panel size (see the $P = 10$ case) still results in a performance hit because the coarseness of the panels limits the ability to tune memory usage to the appropriate level.

The interaction of many factors, including characteristics of the machine, application, and workload determines what panel size is best for a given situation, and it is impossible to conclude that a certain panel size will always be best. As a rule of thumb, however, we suggest using a value around $P = 20$ (i.e. a panel size that is 5% of the managed data set), unless a need to adapt to very small levels of available memory necessitates more panels. In practice we have found $P = 20$ to offer a good compromise between panel-management overheads and tunability of memory usage. We note, however, a user need not agonize too much over the choice of panel size. The goal of MMLIB is to allow applications to perform better under memory pressure than if they simply relied on the virtual memory system, and as Figures 7.6b and 7.7b illustrate, with any reasonable panel size the performance with MMLIB is far superior.

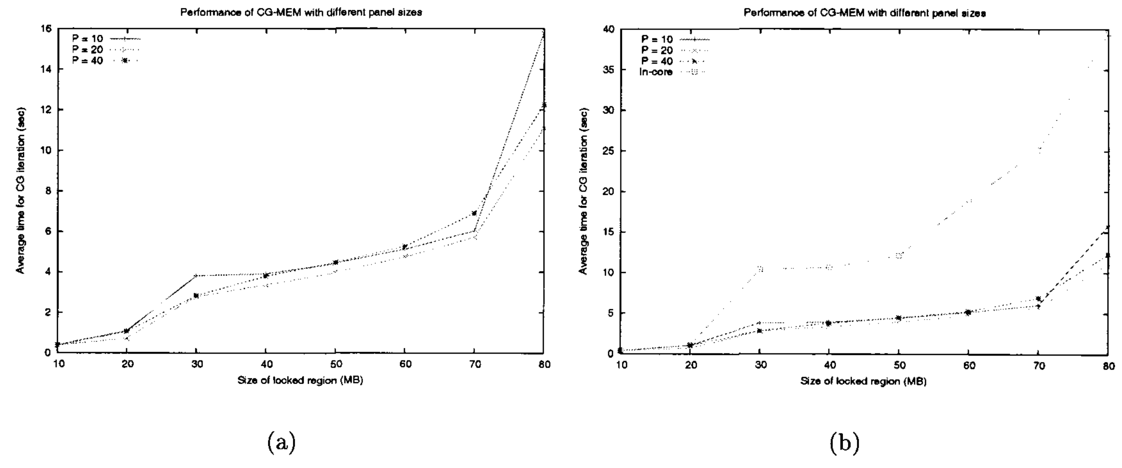
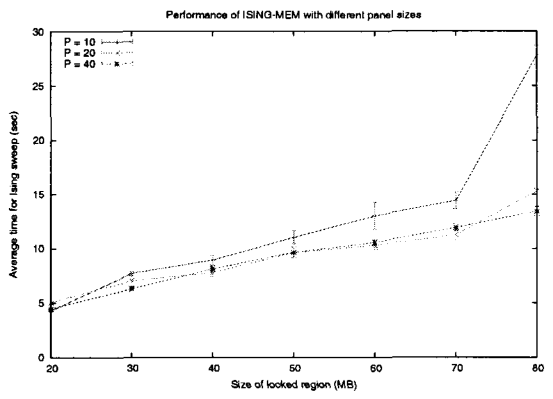
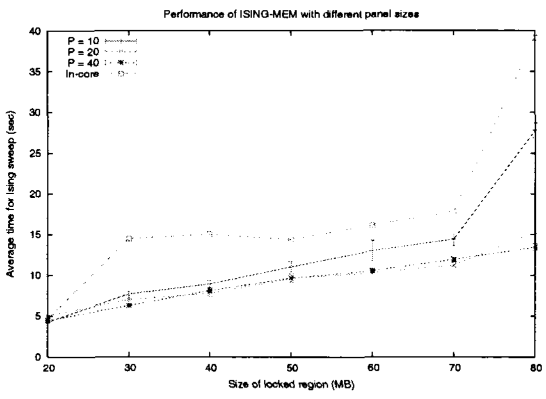


Figure 7.6: Effect of panel size on performance of MMLIB-enabled CG. Chart (a) depicts performance profiles for memory-adaptive CG for some different values of P, the number of panels into which the data set (the coefficient matrix) is broken. CG runs with a 70 MB matrix against static memory load provided by `memlock` on a Linux 2.4 machine with 128 MB of RAM. Chart (b) depicts the same profiles, but includes the performance profile of in-core CG for comparison. Low frequency probing with $R_{pen\_max} = 10$ is used.



(a)



(b)

Figure 7.7: Effect of panel size on performance of MMLIB-enabled ISING. Chart (a) depicts performance of memory-adaptive ISING for some different values of P, the number of panels into which the data set (the spin lattice) is broken. ISING runs with a 70 MB spin lattice against a static memory load provided by `memlock` on a Linux 2.4 machine with 128 MB of RAM. Chart (b) depicts the same profiles, but includes the performance profile of in-core ISING for comparison. The ambient temperature in the simulation is $T = 2$. Low frequency probing with $R_{pen\_max} = 10$ is used.

7.3.2 R_{pen_max}

In section 5.4.2 we explained that we must cap the maximum value that R_{pen}, the ratio used in determining the probing delay, can assume; otherwise MMLIB may fail to ever probe for more memory after detecting a shortage. Here we discuss the choice of R_{pen_max} and MMLIB's sensitivity to its value.

R_{pen_max} in essence specifies the largest amount of time that we are willing to wait to probe for more memory after a shortage has been detected: if the condition RSS = dRSS is satisfied, we will wait at most R_{pen_max} sweeps through the MMLIB-managed data set before probing. R_{pen_max} should be large enough to prevent significant performance loss due to probing too aggressively for memory, but should not be so large as to waste surplus memory by neglecting to try to use it. The challenge is to intelligently pick a value that balances these requirements.

Fortunately, we have found in practice that under static loads MMLIB becomes insensitive to the value of R_{pen_max} once it becomes sufficiently large, which happens at fairly low values. Figure 7.8a depicts performance of memory-adaptive `CG` versus static memory pressure, with different curves corresponding to different values of R_{pen_max}. Performance is clearly inferior for values of R_{pen_max} below ten, but once $R_{pen\_max} = 10$, little performance is gained by increasing its value: the performance curve for $R_{pen\_max} = 100$ is nearly identical, with its 95% confidence intervals overlapping with those of the $R_{pen\_max} = 10$ curve. Figure 7.8b depicts curves similar to those in Figure 7.8a, but for memory-adaptive `ISING`. `ISING` is much less sensitive to the value of R_{pen_max} than is `CG`, perhaps because of the greater amount of work per panel that it performs. It is clear that a value of $R_{pen\_max} = 10$ is sufficient for both codes; in the experiments we report here, we use a value of 10 unless otherwise specified.

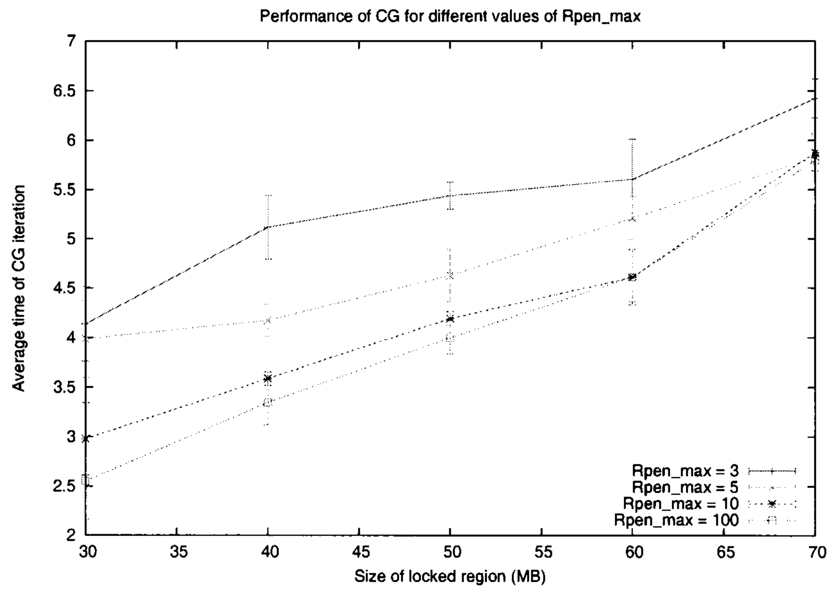
Although we have discussed selection of an appropriate value for R_{pen_max}, we have not yet said anything about the choice between the two types of probing (viz., methods of calculating R_{pen}) described in section 5.4.2. Our assertion that the value of R_{pen_max} should be between 5 and 10 is valid in either case; though Figure 7.8 depicts results using low frequency probing, the curves are nearly identical if high frequency probing is used.

7.3.3 Choice of probing type

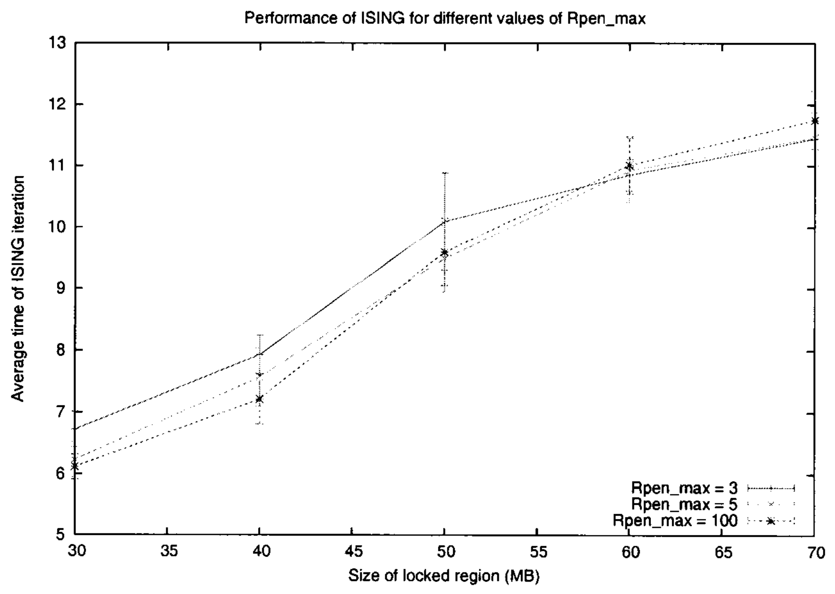
The figures we have presented so far depict performance obtained using low frequency probing (R_{pen} calculated by method I described in section 5.4.2). Here we present some results that compare the two types of probing under some different circumstances.

7.3.3.1 Static memory pressure

Under static memory pressure, we have seen that probing too frequently for surplus memory can incur significant performance penalties. Thus, one would expect high frequency probing to yield worse performance than low frequency probing under static memory loads. Figure 7.9 compares performance between both types of probing on a system subjected to memory pressure from `memlock`. Figure 7.9a demonstrates that our expectation is justi-



(a) Effect of R_{pen_max} on CG performance



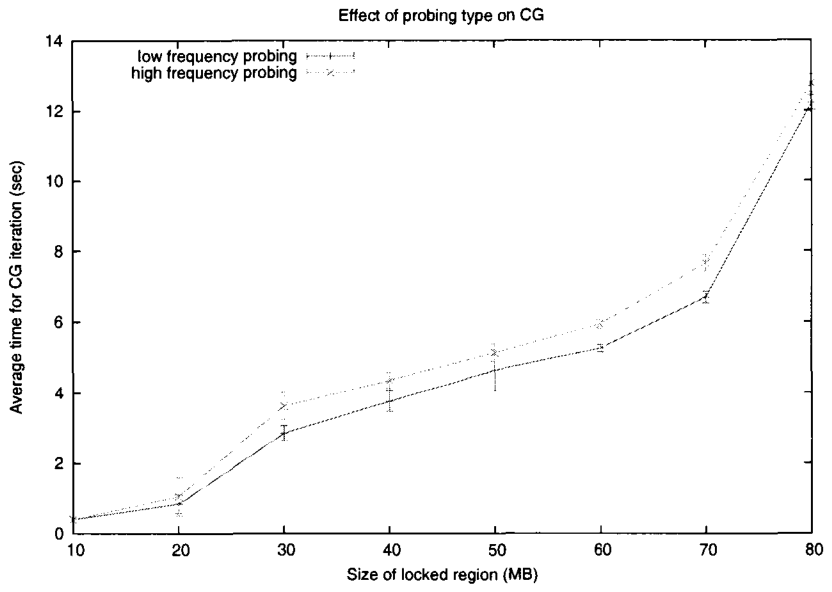
(b) Effect of R_{pen_max} on ISING performance

Figure 7.8: Effects of R_{pen_max} on MMLIB performance. The charts depict performance profiles for a memory-adaptive job running with different values of R_{pen_max} against static memory pressure from memlock on a Linux 2.4 system with 128 MB of RAM. The job in the top chart (a) is CG; in the bottom chart (b) the job is ISING. Error bars represent 95% confidence intervals. In all cases, $P = 20$ and low frequency probing is used. $T = 2$ in the Ising model simulation.

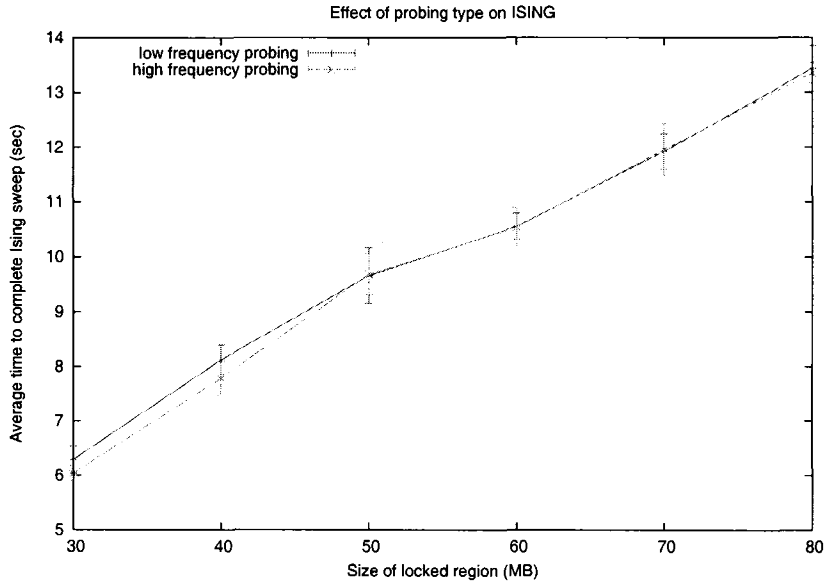
fied for memory-adaptive CG: The performance curves for the different probing types are qualitatively very similar, but the average time per iteration is consistently higher for high frequency probing. Somewhat surprisingly, however, when we compare the performance of the two types of probing in memory adaptive ISING (Figure 7.9b), we can discern no difference between the two methods. We suspect that the disparity between what we see for CG and ISING is due to two key differences between the codes: First, ISING does significantly more work per panel than CG. Second, CG has a significant amount of "static" memory that is not managed by MMLIB, while ISING does not. More frequent probing may have a bigger effect on CG because it causes the static memory to be paged out by the operating system; eviction (and subsequent reloading) of the static memory is less efficient than that of memory handled by MMLIB. Therefore, it seems safer in general to conclude that under constant, long-lived memory pressure, low frequency probing should be used, as high frequency probing offers no performance gains and sometimes (we believe most of the time) results in worse performance. Either method, however, offers far better performance than that obtained using conventional in-core codes.

7.3.3.2 Highly variable memory pressure

High frequency probing was proposed with highly variable memory loads in mind, and here we compare the two probing types under such loads. Table 7.1 presents some experimental results comparing the two in memory-adaptive CG on our Linux 2.4 system. Variable memory pressure is created using our `steplock` code, which works like `memlock` but can increase and decrease the amount of locked memory in discrete steps. We run CG against a `steplock` job that cycles ten times through a sequence in which it locks m_1 MB of RAM for t_1 seconds, then locks m_2 MB for t_2 seconds. We choose the m and t values to create short periods of intense memory usage, followed by longer periods of less intense usage. This is the type of load that low frequency probing should have trouble adjusting to: after dropping its memory usage during intense memory pressure, several iterations may be required to



(a) Effect of probing type on CG performance



(b) Effect of probing type on ISING performance

Figure 7.9: Comparison of performance using the two methods of calculating R_{pen}. Memory-adaptive jobs run on a Linux 2.4 system with 128 MB RAM against static memory pressure from `memlock`. $R_{pen\_max} = 10$ and $P = 20$ in all cases. Chart (a) depicts performance of memory adaptive CG, while chart (b) depicts performance of ISING running with $T = 2$. Error bars represent 95% confidence intervals. Low-frequency probing is clearly superior for CG, but the two types of probing are indistinguishable for ISING.

increase its number of cached panels to take full advantage of the memory available during a period of lesser memory pressure.

Somewhat surprisingly, Table 7.1 shows that low frequency probing delivers equivalent or better performance than high frequency probing in our experiments. As Figure 7.10 illustrates, high frequency probing does tend to take better advantage of jumps in memory availability, but it also pushes memory usage too high, prompting much more page reclamation by the operating system and negating any performance benefits gained by the higher memory utilization. With low frequency probing, CG spends less time fighting with the system for memory, and the lower memory utilization makes little performance difference in practice because the buffer cache speeds up panel "misses" by caching some of those panels that MMLIB does not keep mapped.

While we hesitate to categorically state that high frequency probing should never be used, it has become clear to us that with the systems and workloads used in our experiments, low frequency probing is preferable, as it seems to provide performance equal to or better than that of high frequency probing. In some other situations, the quick response to memory availability that high frequency probing offers might be beneficial. For example, under older versions of Solaris (unpatched pre-2.7 versions), the file cache could interfere with running applications and cause them to thrash; to prevent such problems, file caching might be disabled on some file systems. If panels reside on a disk without file caching, then quick adjustment to memory surplus becomes considerably more important. This is an unusual case, however, and we recommend that low frequency probing be used in most cases.

7.4 Graceful degradation of performance

Our goal is to allow applications to degrade their performance gracefully as memory resources become scarce. Figure 7.11 depicts the performance degradation of each of our three applications under increasing levels of memory pressure. We show one graph per application; each showing the performance of three versions of that application under dif-

Type	m_1	t_1	m_2	t_2	Avg. its	95% low	95% high	stddev
high	70	5	0	5	19.7	17.4	22.0	3.3
low	70	5	0	5	20.2	17.0	23.4	4.4
high	70	5	30	25	59.3	55.7	62.9	5.1
low	70	5	30	25	74.1	72.0	76.2	2.9
high	70	5	50	25	46.0	39.4	52.6	9.4
low	70	5	50	25	60.6	58.1	63.1	3.5

Table 7.1: Comparison of the two types of probing under highly variable memory pressure. Memory-adaptive `CG` runs against a `steplock` job that locks m_1 MB of RAM for t_1 seconds, and then locks m_2 MB of RAM for t_2 seconds, repeating the cycle a total of ten times. `CG` runs with a 70 MB matrix on a Linux 2.4 system with 128 MB RAM; $P = 20$ and $R_{pen\_max} = 10$. We report the average number of iterations that `CG` is able to complete during the time that `steplock` runs, as well as the upper and lower 95% confidence intervals for that average, and the standard deviation. In the "type" column, "high" denotes high frequency probing, and "low" denotes low frequency probing. Low frequency probing equals or out-performs high frequency probing in all cases.

ferent levels of constant memory pressure applied via `memlock`. For each application we test a conventional in-core version (blue top curve), a memory-adaptive version using MMLib with low frequency probing (red lower curve), and an ideal version of the memory-adaptive code that fixes the number of panels cached at an optimal value provided by an oracle. For all three applications, the memory-adaptive version performs significantly and consistently better than the in-core implementation. Additionally, the performance of the adaptive code is very close to the best-case performance, even though the adaptive code has no foreknowledge of memory availability.

7.4.1 Effects of panel write-frequency

One might question whether MMLib becomes ineffectual when applications write to the panels frequently, as many dirty pages must be flushed to disk before a panel can be unmapped. In the case of `CG` the panels are never updated and thus never need their contents flushed. `MGS` does write to panels, but infrequently, doing so only when an orthonormalized vector is added to the basis. `ISING`, however, updates the panels with each sweep through them, and it does so frequently if the ambient temperature T of the simulation is high. To determine if frequent writes to the panels negatively affect performance, we

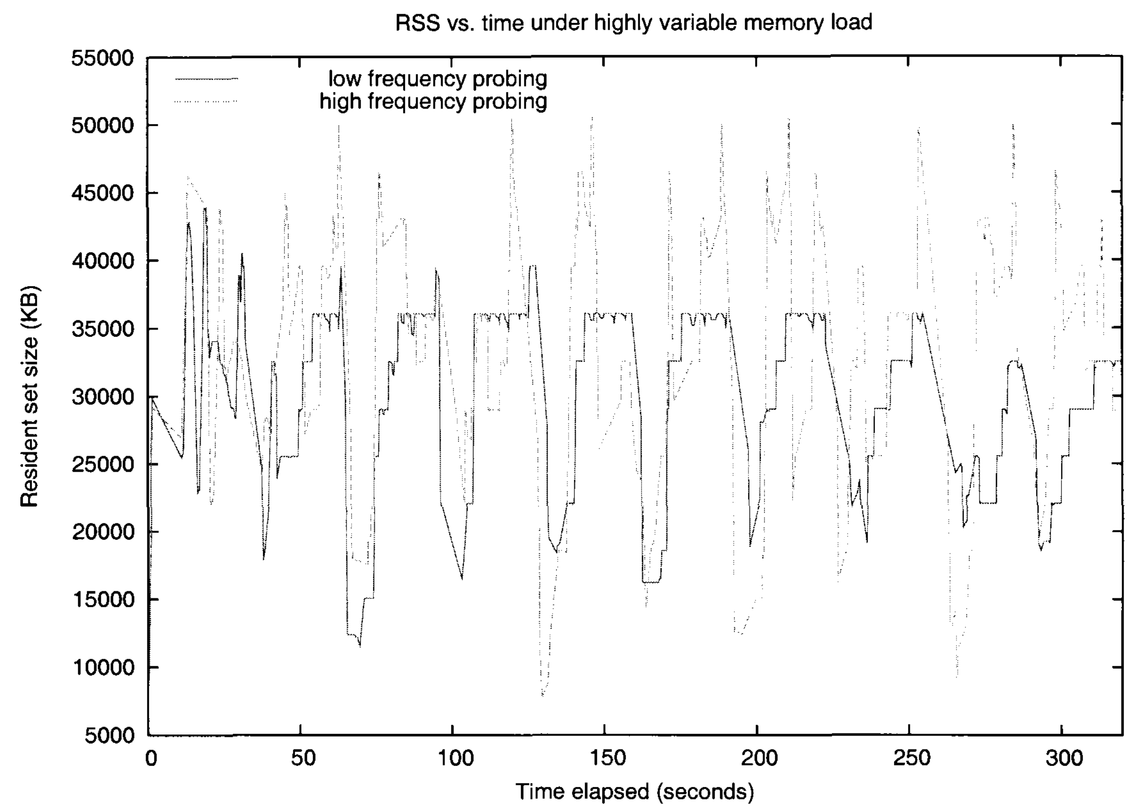
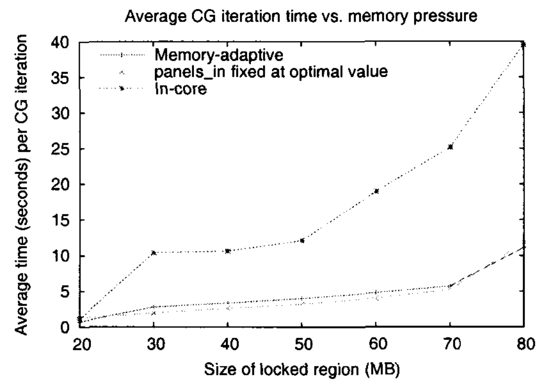
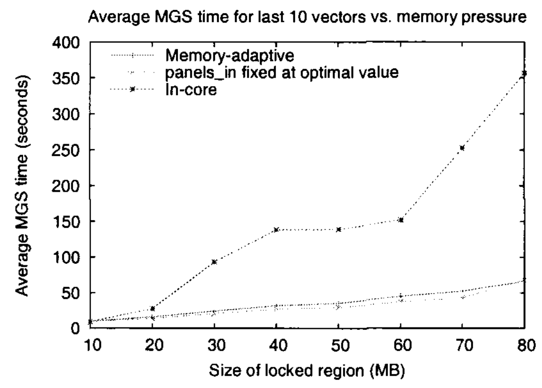


Figure 7.10: RSS versus time for the two types of probing in memory adaptive CG under highly variable memory load. The plot depicts behavior from the $(m_1, t_1, m_2, t_2) = (70, 5, 30, 25)$ case from Table 7.1. Although high frequency probing is able to utilize higher amounts of memory, it also prompts more page reclamation by the operating system.

tested MMLIB-enabled ISING under constant memory pressure for different values of T. Figure 7.12 displays performance curves generated under static memory pressure on Linux 2.4 for temperatures $T = 0$, $T = 2$, and $T = 50$. At $T = 0$, the simulation quickly reaches equilibrium after a few sweeps through the matrix, and afterwards never flips any spins. At the other extreme, $T = 50$, the lattice is in a highly disordered state, with an average of 97.0% of the spins flipping during one sweep through the panels. At $T = 2$, a more modest 16.2% of spins are flipped during a sweep through the panels. The ISING code performs the same amount of CPU work for each sweep, regardless of the simulation temperature. However, we see that as memory pressure increases, ISING at $T = 0$ performs much better than in the $T = 2$ or $T = 50$ cases. This confirms that, not surprisingly, frequent writes





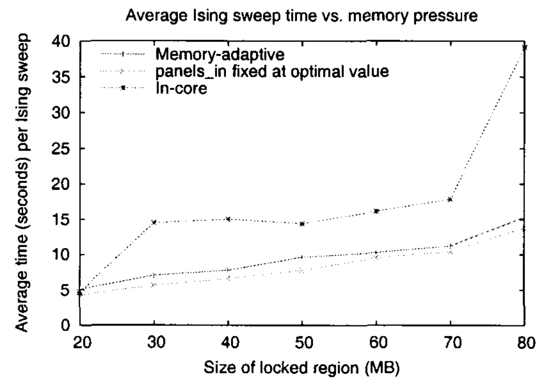


Figure 7.11: Performance under constant memory pressure. The charts show the average time per iteration of a memory-adaptive job running against static memory pressure from `memlock` on a Linux 2.4 node with 128 MB RAM. Low frequency probing with $R_{pen} = 10$ is used in all cases. The top chart shows the average time per iteration of CG with a 70 MB matrix ($P = 20$), which requires a total of 81 MB of RAM including memory for the work vectors. The middle chart shows the time to generate and orthogonalize via modified Gram-Schmidt the last 10 vectors of a 30 vector set. Approximately 80 MB are required to store all 30 vectors, which are stored one vector to a panel ($P = 30$). The bottom chart shows the time required for an Ising code running at $T = 2$ to sweep through a 70 MB lattice ($P = 20$).

to the panels do increase execution times, as there is no avoiding flushing dirty pages to disk when panels are unmapped. Note, however, that even for extremely frequent writes, graceful performance degradation is observed.

One may notice that despite the much higher frequency of writes in the $T = 50$ case, the observed performance is essentially the same as in the $T = 2$ case. This makes sense because although only around 16% of the flips are accepted in the $T = 2$ case, those flips are distributed widely throughout the spin lattice. Because one page can contain over a thousand spins, it is likely that with 16% acceptance, almost every page will be updated and therefore must be flushed to disk. To the memory subsystem, there is essentially no difference between 16% and 97% acceptance.

The impact of frequent writes on performance explains, at least partially, why in Figure 7.11, MMLIB seems to confer less benefit to `ISING` than to `CG` or `MGS` in the sense that lower speedups over the in-core version are observed. The MMLIB-enabled `CG` and `MGS` spend very little time writing to disk, which gives them an advantage over the in-core versions which must write to the swap device. Memory-adaptive `ISING` on the other hand, running at $T = 2$, must devote considerable time to such writes, so it loses some of its advantage over the in-core code. (We could make memory-adaptive `ISING` show better performance in Figure 7.11 by running at $T = 0$, but this temperature is of no scientific interest, so we use $T = 2$, a temperature that physicists might actually wish to simulate.)

7.5 Quick response to transient memory pressure

Our goal is for MMLIB-enabled applications to not only exhibit graceful performance degradation under memory pressure, but to also respond quickly to changes in memory availability. Figure 7.13 illustrates the quick response of our memory-adaptive `ISING` code (using low frequency probing) to transient memory pressure. Depicted is the resident set size versus time for `ISING` running with a 150 MB lattice on a Solaris 9 system with 256 MB of RAM. After running for 120 seconds, a dummy job that randomly writes to a 100 MB region of

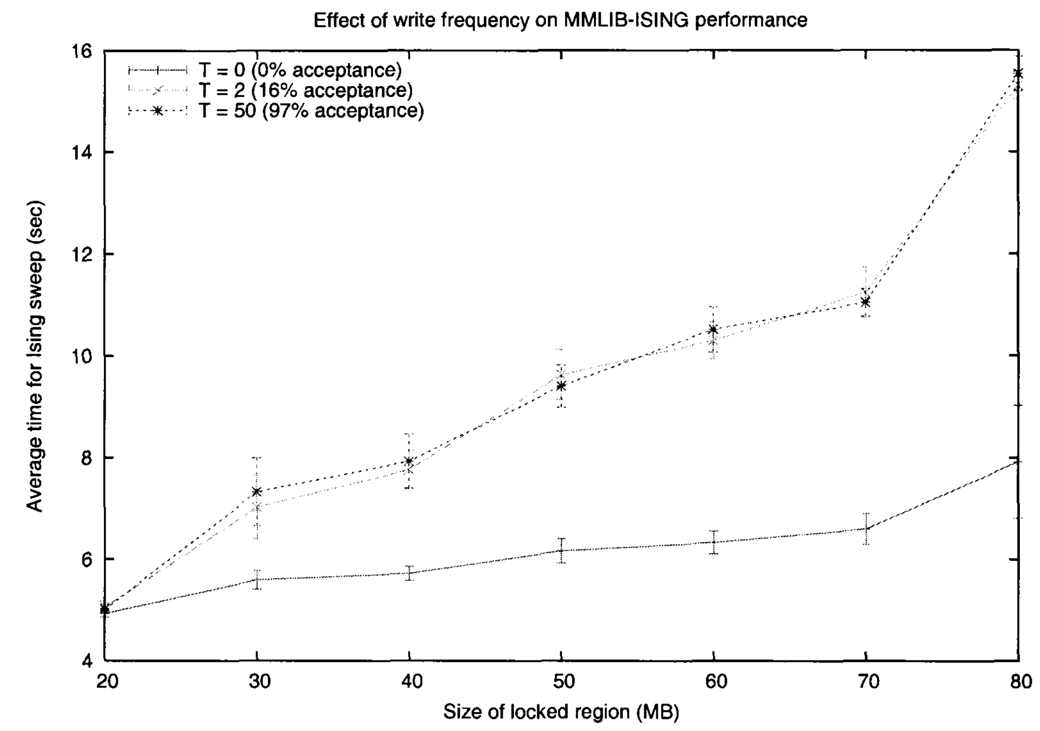


Figure 7.12: Effects of write frequency on MMLIB performance in ISING. ISING runs with a 70 MB spin lattice against static memory pressure applied via `memlock` on a Linux 2.4 system with 128 MB RAM. $P = 20$, $R_{pen\_max} = 10$, and low frequency probing is used. Performance curves are shown for temperatures $T = 0$, $T = 2$, and $T = 50$, which correspond to acceptance probabilities of 0, 16.2, and 97.0 percent. Error bars represent 95% confidence intervals. Performance is markedly better in the $T = 0$ case because no time is spent flushing panels to disk.

memory begins and runs for 30 seconds. As soon as the dummy job introduces memory pressure, MMLIB responds by quickly dropping the ISING memory usage to a safe level; such quick response is crucial to avoid thrashing the system. Soon after the dummy job exits, MMLIB begins to adjust the memory usage back upwards, decreasing the frequency of probes for memory as more memory is obtained. Although it takes considerable time to finally reach the full memory usage, RSS does grow significantly soon after memory pressure disappears. We note that in practical terms the long time to reach full memory usage has little effect, as the buffer cache of the system caches many of the panels that MMLIB does not—hence the fast iteration times observed before full memory utilization is resumed.

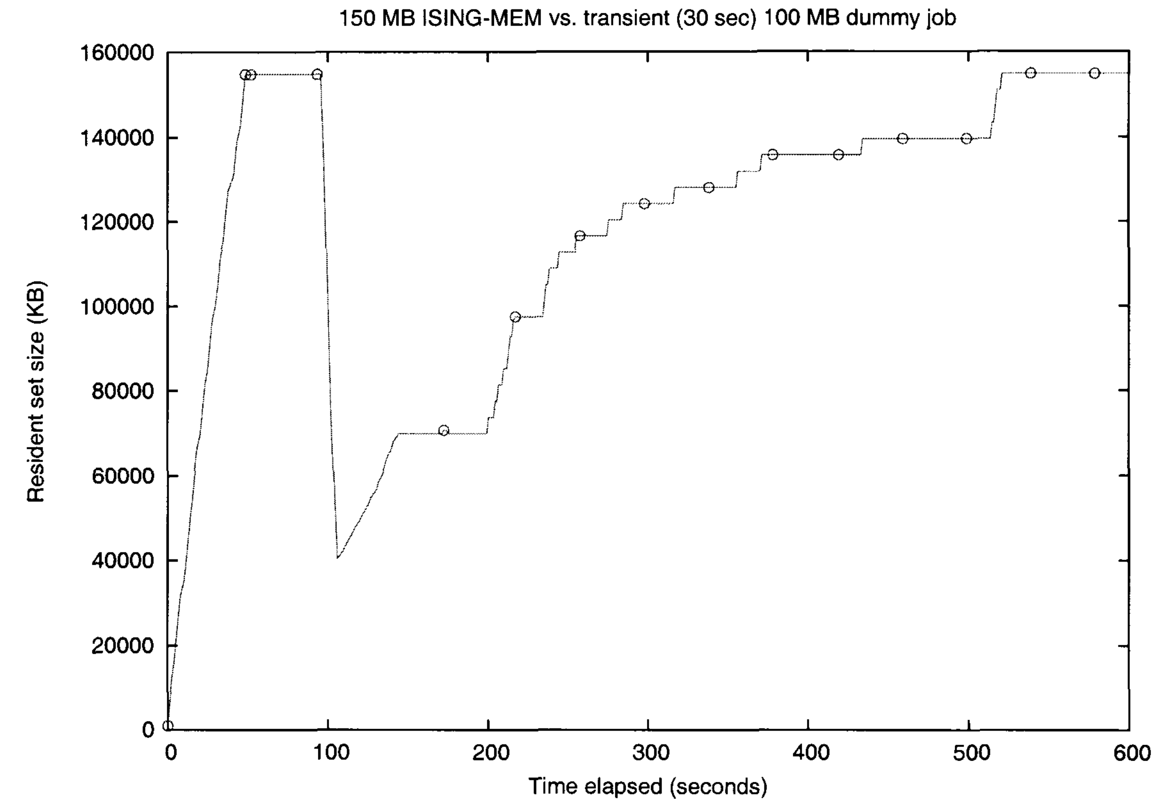


Figure 7.13: Quick response by MMLIB to transient memory pressure. Memory-adaptive ISING runs at $T = 2$ with a 150 MB lattice on a Solaris 9 system with 256 MB of RAM. After 12 seconds, a dummy job that randomly writes to a 100 MB region of RAM begins and runs for 30 seconds. MMLIB quickly drops memory usage when the dummy job begins, and then increases memory usage in a controlled manner after the job exits. $P = 40$, $R_{pen\_max} = 10$ with low frequency probing. Circles denote the beginning of a lattice sweep. (One might notice that the second sweep completes very quickly: only a simple calculation of the initial energy of the spin system is computed during that sweep.)

7.6 Adaptive versus adaptive jobs

For our memory adaptation technique to be actually useful, multiple instances of jobs employing it must be able to coexist on a machine without causing thrashing on, monopolizing, or otherwise disrupting the system. Our research has shown that memory-adaptive jobs run harmoniously with each other, reaching a dynamic equilibrium where system memory utilization is high and good CPU throughput is sustained.

Figure 7.14 depicts the resident set size (RSS) over time for various configurations of two memory-adaptive ISING jobs running simultaneously on a Solaris 9 machine with 256

MB of RAM. We show results from three configurations, all of which have a total memory requirement of 300 MB: a 150 MB versus a 150 MB job, a 200 MB versus a 100 MB job, and a 100 MB versus a 200 MB job. In each case, the second job begins 100 seconds after the first job has started, in order to allow the first job time to touch all of its panels first.

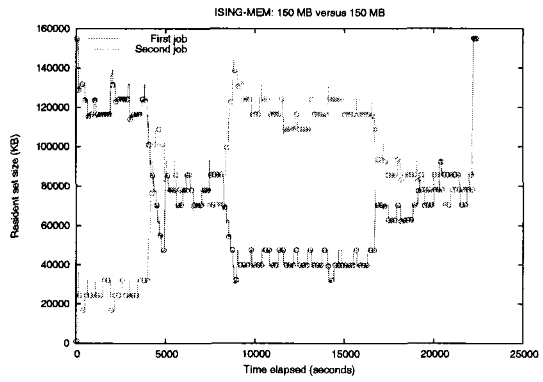
In all cases, the competing jobs quickly reach a sort of dynamic equilibrium in which the total memory usage of the two jobs stays in the region of 150–200 MB. As one job probes for more memory, the other job drops its memory usage by a roughly corresponding amount. In all cases, CPU utilization is high (80% or better is typical) and most of the available memory is used: although there are 256 MB available on a node, the parameter settings for the Solaris 9 operating system are somewhat conservative and limit the total amount of file cache memory to about 200 MB. Because any named memory mapped regions occupy space in the file cache, when the total utilization of our two jobs is at 200 MB, they are using all memory available to them on the system. (Note that our experiments on the Solaris nodes also indicate that they will only allow about 205 MB of anonymous memory to be utilized by user-level processes.)

The results in Figure 7.14 were obtained using low frequency probing. We have also run the adaptive-versus-adaptive experiments using high frequency probing, to see if they corroborate our results from section 7.3.3. Memory pressure arising from competing memory-adaptive jobs is very different from pressure created by our dummy jobs or by conventional, in-core codes, and it seems worthwhile to investigate the effects of the more aggressive probing type in this situation. Figure 7.15 depicts `RSS` versus time using high frequency probing for the three test configurations used in Figure 7.14. Comparing the figures for the two probing types, one can see that the more aggressive probing in the high frequency case results in much more frequent fluctuations in the RSS of the jobs, as expected. In all cases, using high frequency probing results in somewhat higher overall memory utilization on the node. This is most noticeable in the 100 MB versus 200 MB case, where using high frequency probing results in the second, 200 MB job receiving a fairer share of memory compared to the low frequency case. However, the higher memory usage when using high frequency probing seems to be offset by the increased give-and-take between the competing jobs: In the 100 versus 200 MB experiments, while the 100 MB job runs, the 200 MB job averages 131.79 seconds per sweep (standard deviation 10.78 seconds) with low frequency probing, compared to 134.53 seconds per sweep (standard deviation 10.63 seconds) using high frequency probing. This is despite the fact that the 200 MB job consistently gets 20–30 MB more memory in the high frequency case. In the 200 versus 100 MB experiments, for both types of probing the 200 MB job completes 99 iterations before the 100 MB job ends. Looking at the time that this occurs shows that clearly the high frequency jobs fare worse. Similarly, in the 150 MB versus 150 MB experiments, the jobs employing low frequency probing finish roughly 2500 seconds before the high frequency jobs, even though all of the jobs run for 240 iterations. Clearly, in our adaptive-versus-adaptive experiments, low frequency probing results in better performance, just as in the experiments in section 7.3.3. High frequency probing does provide much quicker response to memory surplus, but this is offset by the increased give-and-take between the competing jobs.

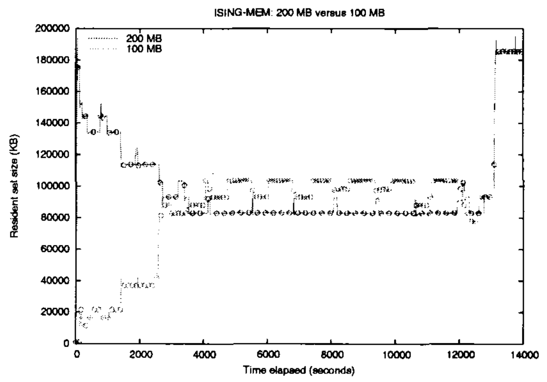
7.7 Performance with LRU-friendly access patterns

All three of the applications we have tested employ memory access patterns for which MRU-replacement is most appropriate, and are therefore at extreme odds with the LRU-like algorithms employed by the operating systems. The results we have presented so far show that the MMLIB-enabled codes yield superior performance compared to conventional in-core versions that rely on the virtual memory system to handle memory pressure. The question remains, however, whether these performance gains are due solely to MMLIB enabling MRU replacement to be used, or are attributable to other characteristics of the adaptation strategy, such as avoidance of writes to the swap device or pre-fetching of panels.

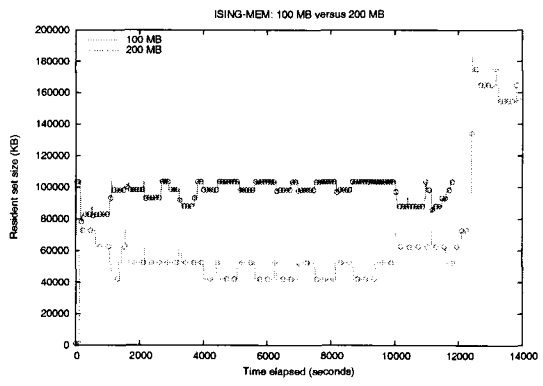
To answer this question, we have developed a version of the `CG` code that utilizes an LRU-friendly access pattern. Normally, `CG` performs matrix-vector multiplications starting



(a) 150 MB vs. 150 MB, R_{pen} type I

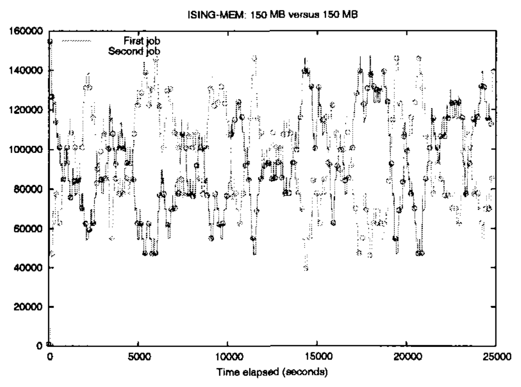


(b) 200 MB vs. 100 MB, R_{pen} type I

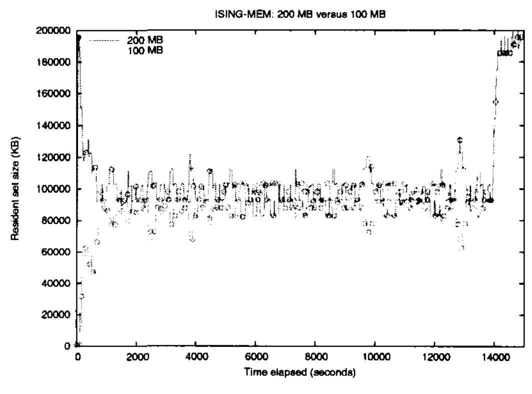


(c) 100 MB vs. 200 MB, R_{pen} type I

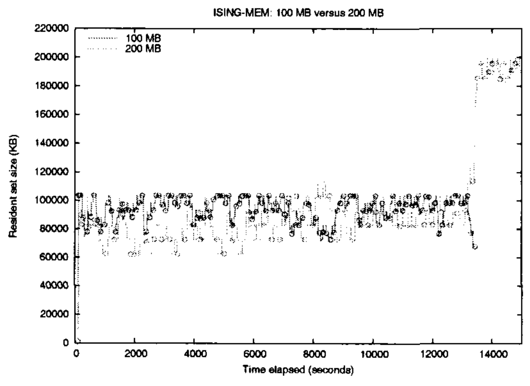
Figure 7.14: Profiles of RSS vs. time for two instances of memory adaptive ISING jobs running simultaneously, using low frequency probing. In each experiment, the second job starts 100 seconds after the first job. Chart (a) depicts the 150 MB vs. 150 MB case, chart (b) 200MB vs. 100MB, and chart (c) 100 MB vs. 200 MB. Circles denote the beginning of lattice sweeps: horizontal distance between consecutive circles along a curve indicate the time required to complete the sweep. In all cases, the simulated temperature is $T = 2$, $P = 20$, and $R_{pen\_max} = 10$.



(a) 150 MB vs. 150 MB, R_{pen} type II



(b) 200 MB vs. 100 MB, R_{pen} type II



(c) 100 MB vs. 200 MB, R_{pen} type II

Figure 7.15: Profiles of RSS vs. time for two instances of memory adaptive ISING jobs running simultaneously, using high frequency probing. The charts depict the same experimental cases as shown in Figure 7.14, but using high frequency probing instead of low frequency probing. Chart (a) depicts the 150 MB vs. 150 MB case, chart (b) 200MB vs. 100MB, and chart (c) 100 MB vs. 200 MB. Circles denote the beginning of lattice sweeps: horizontal distance between consecutive circles along a curve indicate the time required to complete the sweep. In all cases, the simulated temperature is $T = 2$, $P = 20$, and $R_{pen\_max} = 10$.

with the first row of the matrix and proceeding down the rows until the last row is processed. That is, at each CG iteration, a sweep from the top to the bottom of the matrix (and, for the memory-adaptive case, from the first to the last panel) is performed. We can make CG LRU-friendly, however, by employing an alternating access pattern: during one iteration, sweep through the matrix from top to bottom, and then during the next iteration, sweep from bottom to top. In our LRU-friendly implementations, CG-LRU, the conventional in-core code performs the backsweeps row by row, while the MMLIB-enabled version sweeps backwards through its set of panels, but processes the rows of each panel in the usual top to bottom fashion. In this manner the MMLIB version utilizes an LRU-friendly access pattern while still taking advantage of the pre-fetching that the large units (panels) of data access enable.

Figure 7.16 compares the in-core and MMLIB-enabled versions of CG-LRU with the in-core CG that uses the conventional, MRU-friendly access pattern. The MMLIB-enabled CG-LRU performs well (as expected) under memory pressure. But somewhat surprisingly, the in-core CG-LRU performs only marginally better than conventional in-core CG, despite the improved access pattern used by CG-LRU. Because the activity that occurs during thrashing can be quite complicated, it is difficult to say why CG-LRU fails to exhibit significant performance improvements. Of course, in-core CG-LRU does not avoid VM-system overheads and write-backs to the swap device, and it cannot take advantage of pre-fetching from backing store like MMLIB-enabled codes can, but it seems unlikely that these factors account entirely for the poor performance we see. It may be that CG-LRU loses significant time due to the work vectors being paged out: While CG-LRU uses an LRU-friendly pattern to access the matrix, this modification does not extend to the work vectors. The MMLIB-enabled CG codes can protect the work vectors from page reclamation by controlling the number of panels cached, while the in-core CG-LRU cannot.

To further isolate the effects of using optimal replacement from the other improvements that MMLIB affords, we conducted experiments with memory-adaptive CG in which we used

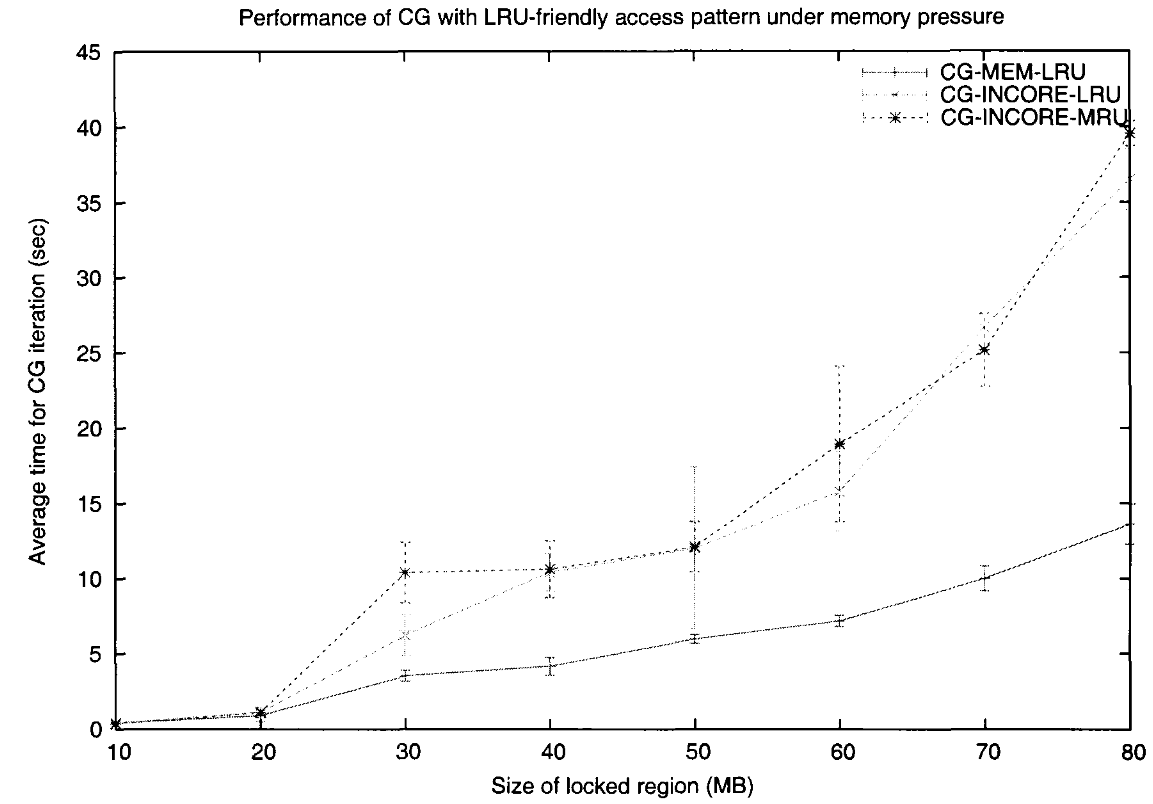


Figure 7.16: Performance of in-core and memory adaptive versions of CG-LRU, which uses an LRU-friendly access pattern. Jobs run under static memory pressure provided by `memlock` on a Linux 2.4 system with 128 MB of RAM. The performance profile for in-core, MRU-friendly CG is also depicted for comparison. $P = 20$ and low frequency probing is used with $R_{pen\_max} = 10$. Error bars represent 95% confidence intervals.

the normal, MRU-appropriate access pattern, but we instructed MMLIB to use LRU replacement. This ensures that under any significant memory pressure, all mmlib_get_panel() calls will result in a panel miss, so any benefits conferred by MMLIB will not be due to the replacement strategy. Figure 7.17 compares the CG code using the wrong replacement policy (CG-LRU-WRONG) with the CG correctly employing MRU. The version using LRU replacement performs significantly worse, requiring roughly twice the amount of time required by the MRU version to perform one iteration. However, when compared to the performance of in-core CG under memory pressure, the CG-LRU-WRONG code still performs iterations in roughly half the time of in-core CG at lower levels of memory pressure, and at higher levels performs even better.

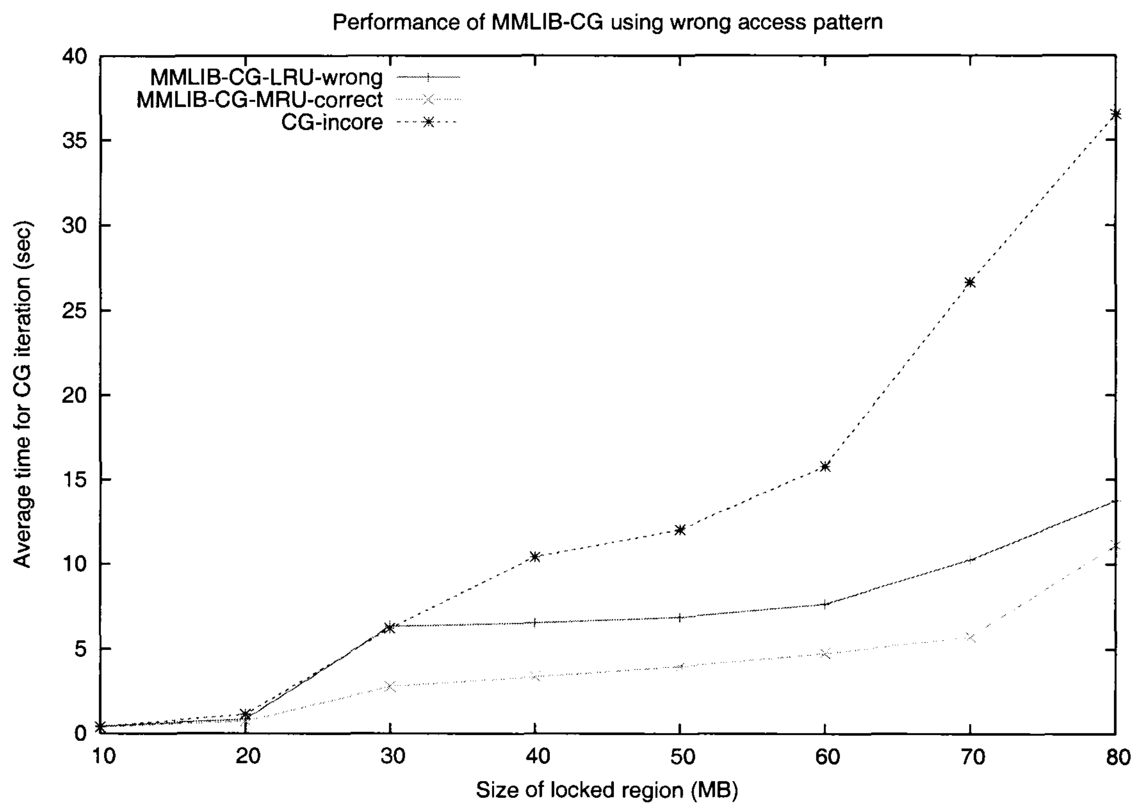


Figure 7.17: Performance of memory-adaptive CG versus memory pressure when using the wrong panel replacement policy. The appropriate replacement policy is MRU, but LRU is used instead; any observed performance gains are not due to the ability of MMLIB to allow application-specific replacement. Jobs run under static memory pressure provided by `memlock` on a Linux 2.4 system with 128 MB of RAM. $P = 20$, and low frequency probing is used with $R_{pen\_max} = 10$.

All of the experiments with LRU replacement demonstrate that the value of MMLIB does not lie solely in its ability to allow use of application-specific replacement policies. The larger granularity of memory access and preemption of the virtual memory system and its associated overheads and possible antagonism also play a vital role.

Chapter 8

Conclusions and future work

We contend that in order to effectively utilize modern, shared computing environments, applications should monitor and adapt to changing system conditions. This research has pursued two distinct but complementary mechanisms to enable some classes of scientific applications to adapt to variations in CPU and memory availability.

8.1 Flexible-phase load balancing

The first mechanism we studied is a strategy for dynamic load balancing of a class of iterative algorithms that possess what we term a *flexible phase*, during which the amount of local work completed can vary without detriment to the end result. The technique can dynamically correct load imbalance from both internal and external sources and requires minimal overhead. Although limited to a narrow class of algorithms, this class includes some popular numerical algorithms that form the computational kernels of many scientific applications. We have experimentally demonstrated the utility of our method in two Krylov-like methods: a multi-grain formulation of a block Jacobi-Davidson eigensolver and an FGMRES linear solver employing additive-Schwarz preconditioning. Our method has proven able to cope with severe load imbalance in the Jacobi-Davidson eigensolver, and it makes practical the full-multigrain parallel formulation that can hide network latencies but may accentuate load imbalance. In the additive-Schwarz preconditioned FGMRES solver, performance gains are significant but more modest due to some intrinsic properties of the numerical method. We also note that the method can smooth load imbalances due to differences in conditioning of the subdomains, which cannot be fixed by repartitioning.

Our load balancing strategy is able to rectify large load imbalances in the multi-grain, block eigensolver we tested, but is more limited in the domain-decomposition linear system solver. In future work, it may be worthwhile to see if we can achieve better results with a linear solver by working with a multi-grain block Krylov-type solver, such as block FGMRES. Additionally, our strategy may show better performance in a domain-decomposition context involving coarser subproblems, such as an inexact Newton method employing nonlinear additive Schwarz preconditioning [17, 16]. In such a method, a nonlinear problem is solved on each subdomain, involving much more computation than the simple, linear subdomain solves of the additive-Schwarz preconditioner for FGMRES.

8.2 A dynamic memory-adaptation framework

The second idea we have explored is a general framework that allows scientific and other data-intensive applications to dynamically adapt their memory requirements, facilitating graceful degradation of performance as memory pressure arises. It does so with minimal reliance on operating system provided information, and can be embedded in many scientific kernels with little coding effort using the supporting library, MMLIB, we have developed. The key technical hurdle in realizing our framework is the lack of information about memory availability provided by operating systems. We have developed a relatively simple algorithm that judges memory availability effectively using a single metric, the resident set size of the application. The memory-adaptation framework is especially suited to non-centrally administered, open systems, and supports the following functionality:

- management of multiple memory objects with multiple active panels at the same time
- memory objects that can be modified or vary in size
- access to memory objects at user-defined granularities

- automatic estimation of the size of memory not managed within the framework
- user-defined panel replacement policies, which can optionally be coordinated with the system policy through a "most-missing" alternative.

We have used our MMLIB library to embed memory adaptivity into three scientific applications: a conjugate gradient linear system solver, a modified Gram-Schmidt orthogonalization routine, and a Monte-Carlo Ising model simulation. These applications or kernels, though simple, are representative of elements of a wide range of scientific and engineering codes. Experimental evaluation of the applications in Linux and Solaris environments demonstrate that the memory-adaptation framework yields considerable performance advantages over reliance on the virtual memory system to handle memory pressure. These advantages stem not only from enabling the use of application-specific replacement policies, but from avoidance of virtual memory system overheads and antagonism. Importantly, multiple jobs employing the memory-adaptation framework can coexist on a system without thrashing, monopolizing, or otherwise disrupting the system.

We believe that our results have demonstrated the utility of our memory-adaptation framework, but many topics remain for future work. We have briefly investigated the use of a "most-missing" replacement policy that is coordinated with the page reclamation of the operating system. Our results using this policy to evict panels that have been mostly evicted by the operating system show promise, but there are some questions that must be answered: We suspect that we should always circumvent the application specific policy during memory shortage, but we must verify that this is the case or whether it is sometimes appropriate to strike a balance between the two policies. If a balance should be struck, we must determine the thresholds that dictate when the MRU policy should be followed for a page that is targeted by the most-missing policy.

We have designed MMLIB to require as little operating system provided information as possible, in part to facilitate portability. We have tested MMLIB extensively under Linux 2.4 and Solaris 9, and to a lesser extent under Linux 2.2 and Solaris 7 and 8. We have also tested it cursorily under MacOS X 10.1. It is important to evaluate MMLIB under other operating systems, whose memory-management strategies may differ. For instance, we have not evaluated MMLIB under systems that employ a page-fault frequency (PFF) strategy for thrashing prevention.

We have demonstrated with MMLIB that it is possible to hide memory-adaptation decisions and bookkeeping behind a simple library interface. The current version of the library is still somewhat rudimentary, however, and work should be done to further refine it. For example, more functions implementing common panel replacement policies should be added, and interfaces to facilitate mapping more complex data structures to panels should be provided. Developing a more "production ready" version of MMLIB will make easier what is perhaps the most important work to be pursued: using MMLIB in more types of and more complex applications. Our eventual goal is to use MMLIB to implement memory adaptivity in a widely used scientific toolkit such as PETSc. By providing features such as a memory-adaptive sparse matrix class that supports the standard PETSc matrix methods, we can enable scientists to embed, with minimal effort, memory-adaptivity into their existing application codes. Such ease of use is vital if we wish to see adoption of our memory-adaptivity techniques in real-world applications.

Appendix A

LBlib: A flexible-phase load balancing library

To facilitate general use of our load balancing strategy, we have written an object-based C library, LBLIB, that hides much of the required bookkeeping from the application programmer. To simplify data management and provide information hiding, data required for resource balancing are stored in a variable of the defined type `LBS`. An application programmer can access the data within an `LBS` structure through LBLIB functions. Data encapsulation is ensured by appropriate use of `void *` pointers in the internal implementation. A unique `LBS` structure is explicitly associated with a group of processors, and implicitly with a particular flexible section that these processors must load balance. Below, a subset of the basic functionality of the library is described:

`LBS lb_new_lbstruct(MPI_Comm communicator)`

This is the `LBS` constructor function, where `communicator` is the MPI communicator with which the `LBS` is to be associated. Note that a node of an MPI job could participate in the load balancing of nested, but different, flexible sections.

`void lb_section_start(LBS lbs)`

`double lb_section_end(LBS lbs, double ops_completed)`

These functions designate the beginning and end of the flexible section associated with the `lbs` structure, which is to be load balanced. `lb_section_start()` begins timing the execution of the section and, if memory balancing is used, it also begins tracking the amount of page swapping and CPU idling. Once all operations within the flexible section are completed, `lb_section_end()` stops the timing and calculates the rate at which the local processor performed operations since `lb_section_start()`. This rate is returned by the function for convenience, though it is also stored in the `lbs`. `ops_completed` is the number of operations that were completed during the flexible section. It is up to the application programmer to decide on a suitable way of counting the number of operations. In our eigensolver application, we use the number of iterations performed by the linear system solver during the correction phase.

`double lb_decide(LBS lbs, double ops, int method, [function])`

Before entering a section we must determine the time T that all processors will spend in it. This is accomplished by calling `lb_decide()`, where `ops` is the number of operations that we ideally want each processor to complete. `lb_decide()` is the only synchronous LBLIB call that causes all processors in the communicator associated with `lbs` to gather the computation rates observed by each processor during the most recent execution of the flexible section. The rates are then sorted, and are used to determine T using the method specified by the `method` argument. In the current version of LBLIB, the possible values of `method` and the corresponding procedure for determining T are as follows:

- `LB_USE_FASTEST`: T is the predicted time for the fastest processor to compute `ops` operations.
- `LB_USE_SLOWEST`: T is the predicted time for the slowest processor to compute `ops` operations.
- `LB_USER_DEFINED`: A pointer to a user-defined function for calculating T is passed as a fourth argument. `lb_decide()` calls this function, passing `ops` and the array of sorted

rates to it.

`lb_decide()` returns the time T, but it also stores it in `lbs` so that the application programmer does not need to know it. Note that before the first call to `lb_decide()`, one execution of the flexible section should have already been timed through calls to `lb_section_start()` and `lb_section_end()`. Otherwise, no computation rates are available for `lb_decide()` to determine the time T.

`int lb_continue(LBS lbs, double ops_completed, double ops_needed)`

This function allows the programmer to check whether the allotted time T is about to be exceeded, in which case the program must exit the flexible section. `ops_completed` is the number of operations completed so far during the current flexible section and `ops_needed` is the number of operations that will be completed before another call to `lb_continue()` can be made. `lb_continue()` calculates the execution rate of the local processor during the current flexible section, and then uses this rate to predict the time required to do `ops_needed` more operations. If this time falls within the allotted time, then `lb_continue()` returns 1, indicating that execution of the flexible section should continue. Otherwise it returns 0, indicating that we should exit the flexible section to avoid load imbalance. In our eigensolver application, the value of `ops_needed` is one, because we call `lb_continue()` after each iteration of the linear system solver.

`double *lb_get_rates(LBS lbs)`

`int *lb_get_index(LBS lbs)`

It is often necessary for the application programmer to know the ordered rates of the processors, so that the critical tasks can be assigned to the most appropriate processors. `lb_get_rates()` returns a pointer to an array containing the rates of the processors, in ascending order. The processor indices corresponding to each of these rates can be retrieved through a pointer to an array returned by `lb_get_rates()`.

The above functions significantly simplify the load balancing of codes that abide by the general model of section 3.1. Figure A.1 demonstrates the use of LBLIB to balance CPU load in our coarse-grain JDcg eigensolver. Note that, the first time through the flexible section, the for-loop should execute a constant number of iterations in order to collect the first set of performance data. The semantics of the flexible section and its associated data dictate certain rules when calling some functions of the LBLIB library. Following the `lb_decide()` semantics, `lb_continue()` must be placed between an `lb_section_start()` and an `lb_section_end()`. Also, `lb_get_rates()` must be called only after a call to `lb_decide()` has been made, so that the rates have been exchanged between processors. Finally, between a call to `lb_section_start(lbs1)` and a call to `lb_section_end(lbs1)`, we cannot issue another call to `lb_section_start(lbs1)`. However, a call to `lb_section_start(lbs2)`, for a different `LBS` structure, is permissible.

In order to enforce such rules, an `LBS` keeps track of two fields: `scope` and `context`. `scope` keeps track of whether program execution is currently within the flexible section, while `context` tracks the availability of data for functions such as `lb_decide()`. In case of inappropriate scope or context, LBLIB functions set an error code in the `LBS`, which the programmer can check through `void lb_chkerr(LBS lbs)`.

```
Algorithm: Load balanced JDcg
lbs = lb_new_lbstruct(MPI_COMM_WORLD);
Until convergence do:
        // Control phase
        Perform projection phase, steps 1–7 of JDcg
        Determine optimal number of iterations optits to perform.
        lb_decide(lbs,optits,LB_USE_FASTEST);
        ordering = lb_get_index(lbs);
        Perform all-to-all: faster processors receive more critical residuals

        // Flexible Phase
        lb_section_start(lbs);
            for (ops = 0; lb_continue(lbs,ops,1) ; ops++)
                Perform one BCGSTAB step on eq. (3.1)
        lb_section_end(lbs,ops);
end do
```

Figure A.1: Pseudocode depicting how the LBLIB library can be used to load balance the JDcg solver.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, 1986.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Du Croz, A. Greenbaum, S. Hammarling, A McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide, Third Edition.* SIAM, 1999.

[3] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[4] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, 2001.

[5] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[6] Satish Balay, Victor Eijkhout, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, pages 163–202. Birkhäuser Press, 1997.

[7] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali. A load balancing framework for adaptive and asynchronos applications. *IEEE Transasctions on Parallel and Distributed Systems*, 14(12):183–192, 2004.

[8] K. J. Barker and N. Chrisochoides. An evaluation of a framework for the dynamic load balancing of highy adaptive and irregular applications. In *Proceedings of the IEEE/ACM Supercomputing Conference*, 2003.

[9] Rakesh D. Barve and Jeffrey Scott Vitter. A theoretical framework for memory-adaptive algorithms. In *IEEE Symposium on Foundations of Computer Science*, pages 273–284, 1999.

[10] M. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580, 1987.

[11] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, 1989.

[12] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[13] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Supercomputing 1996*, Fall 1996.

[14] Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, Gary Shao, Shava Smallen, Neil Spring, Alan Su, and Dmitrii Zagorodnov. Adaptive computing on the grid using apples. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, 2003.

[15] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 297–308, 1996.

[16] X.-C. Cai, D. E. Keyes, and L. Marcinkowski. Nonlinear additive Schwarz preconditioners and applications in computational fluid dynamics. *International Journal of Numerical Methods in Fluids*, 40:1463–1470, 2002.

[17] Xiao-Chuan Cai and David Keyes. Nonlinearly preconditioned inexact Newton algorithms. *SIAM J. Sci. Comput.*, 24:183–200, 2002.

[18] P. Cao, E. Felten, A. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[19] F. Chang and V. Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *Proceedings of HPDC '00*, pages 11–20. IEEE CS Press, 2000.

[20] Y.-J. Chiang. Out-of-core isosurface extraction of time-varying fields over irregular grids. In *Proceedings of IEEE Visualization '03*, pages 217–224, 2003.

[21] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, 1989.

[22] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan. Zoltan: A dynamic load-balancing library for parallel applications; user's guide. Technical Report Tech. Rep. SAND99-1377, Sandia National Laboratories, Albuquerque, NM, 1999.

[23] J. DONGARRA, J. DUCROUZ, I. DUFF, AND S. HAMMARLING. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.

[24] J. DONGARRA, J. DUCROUZ, S. HAMMARLING, AND R. HANSON. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14:1–32, 1988.

[25] J. DONGARRA, S. HAMMARLING, AND D. WALKER. Key concepts for parallel out-of-core LU factorization. *Parallel Computing*, 23(1–2):49–70, April 1997.

[26] D. R. FOKKEMA, G. L. G. SLEIJPEN, AND H. A. VAN DER VORST. Jacobi-Davidson style QR and QZ algorithms for the partial reduction of matrix pencils. *SIAM J. Sci. Comput.*, 20(1), 1998.

[27] I. FOSTER AND C. KESSELMAN. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[28] GEOFFREY C. FOX, ROY D. WILLIAMS, AND PAUL C. MESSINA. *Parallel Computing Works.* Morgan Kaufmann Publishers, 1994.

[29] MATTEO FRIGO AND STEVEN G. JOHNSON. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[30] MATTEO FRIGO, CHARLES E. LEISERSON, HARALD PROKOP, AND SRIDHAR RAMACHANDRAN. Cache-oblivious algorithms. In *Proc. 40th Ann. Symp. on Foundations of Comp. Sci. (FOCS)*, pages 285–297. IEEE Comput. Soc., 1999.

[31] A. GEORGE AND J. LIU. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, 1981.

[32] JOHN R. GILBERT, GARY L. MILLER, AND SHANG-HUA TENG. Geometric mesh partitioning: Implementation and experiments. *SIAM J. Sci. Comput.*, 19(6):2091–2110, 1998.

[33] HARVEY GOULD AND JAN TOBOCHNIK. *An introduction to computer simulation methods: applications to physical systems.* Addison-Wesley Publishing Company, 1996.

[34] K. GREMBAN, GARY L. MILLER, AND SHENG-HUA TENG. Moments of inertia and graph separators. In *Proceedings of the Fifth SIAM Symposium on Discrete Algorithms*, 1994.

[35] KIERAN HARTY AND DAVID R. CHERITON. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27:9, pages 187–197, New York, NY, 1992. ACM Press.

[36] B. HENDRICKSON AND R. LELAND. The Chaco user's guide — version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, 1994.

[37] BRUCE HENDRICKSON AND ROBERT W. LELAND. A multi-level algorithm for partitioning graphs. In *Proceedings of Supercomputing '95*, 1995.

[38] Y. F. HU, R. J. BLAKE, AND D.R. EMERSON. An optimal migration algorithm for dynamic load balancing. *Concurrency: practice and experience*, 10:467–483, 1998.

[39] S. F. HUMMEL, E. SCHONBERG, AND L. E. FLYNN. Factoring: a practical and robust method for scheduling parallel loops. In *Proceedings of Supercomputing 91*, pages 610–619. IEEE CS Press, 1991.

[40] S. JIANG AND X. ZHANG. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 31–42, 2002.

[41] SONG JIANG. *Efficient caching algorithms for memory management in computer systems.* PhD thesis, Department of Computer Science, College of William and Mary, 2004.

[42] M. T. JONES AND P.E. PLASSMANN. Computational results for parallel unstructured mesh computations. *Computing Systems in Engineering*, 5:297–309, 1994.

[43] GEORGE KARYPIS AND VIPIN KUMAR. *METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, 1995.

[44] GEORGE KARYPIS AND VIPIN KUMAR. A coarse-grain parallel formulation of a multilevel k-way graph-partitioning algorithm. In *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[45] GEORGE KARYPIS AND VIPIN KUMAR. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[46] P. J. KELEHER, J. K. HOLLINGSWORTH, AND D. PERKOVIC. Exposing application alternatives. In *Proceedings of ICDCS '99*, pages 384–392. IEEE CS Press, 1999.

[47] C. T. KELLEY. *Iterative methods for linear and nonlinear equations.* SIAM, 1995.

[48] B. KERNIGHAN AND S. LIN. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 29:291–307, 1970.

[49] KEITH KRUEGER, DAVID LOFTESNESS, AMIN VAHDAT, AND THOMAS ANDERSON. Tools for the development of application-specific virtual memory management. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 48–64, 1993.

[50] CLYDE P. KRUSKAL AND ALAN WEISS. Allocating independent subtasks on parallel processors. *IEEE Trans. Softw. Eng.*, 11(10):1001–1016, 1985.

[51] V. KUMAR, A. Y. GRAMA, AND NAGESHWARA RAO VEMPATY. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.

[52] Vipin Kumar and V. Nageshwara Rao. Parallel depth-first search on multiprocessors part I: Analysis. *International Journal of Parallel Processing*, 16(6):501–519, 1987.

[53] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Soft.*, 5:308–325, 1979.

[54] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1360, 2001.

[55] Benoit B. Mandelbrot. *The Fractal Geometry of Nature.* W. H. Freeman, 1982.

[56] Jim Mauro and Richard McDougall. *SOLARIS Internals, Core Kernel Architecture.* Prentice Hall PTR, 2001.

[57] J. R. McCombs, R. T. Mills, and A. Stathopoulos. Dynamic load balancing of an iterative eigensolver on networks of heterogeneous clusters. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003)*, 2003.

[58] J. R. McCombs and A. Stathopoulos. Multigrain parallelism for eigenvalue computations on networks of clusters. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC 02)*, pages 143–149, 2002.

[59] J. R. McCombs and A. Stathopoulos. Parallel, multigrain iterative solvers for hiding network latencies on MPPs and networks of clusters. *Parallel Computing*, 29(9):1237–1259, 2003.

[60] D. McNamee and K. Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. Technical Report TR-90-09-05, Carnegie Mellon University School of Computer Science, 1990.

[61] R. T. Mills, A. Stathopoulos, and D. S. Nikolopoulos. Adapting to memory pressure from within scientific applications on multiprogrammed COWs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2004)*, 2004.

[62] R. T. Mills, A. Stathopoulos, and D. S. Nikolopoulos. Runtime support for memory adaptation of scientific applications on shared computational resources. 2004. To be submitted.

[63] R. T. Mills, A. Stathopoulos, and E. Smirni. Algorithmic modifications to the Jacobi-Davidson parallel eigensolver to dynamically balance external CPU and memory load. In *2001 International Conference on Supercomputing*, pages 454–463. ACM Press, 2001.

[64] D. Nikolopoulos. Malleable memory mapping: User-level control of memory bounds for effective program adaptation. In *Proc. of the 17th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice, France, April 2003.

[65] Dimitrios S. Nikolopoulos and Constantine D. Polychronopoulos. Adaptive scheduling under memory constraints on non-dedicated computational farms. *Future Gener. Comput. Syst.*, 19(4):505–519, 2003.

[66] Leonid Oliker and Rupak Biswas. PLUM: parallel load balancing for adaptive unstructured meshes. *Journal of Parallel Distributed Computing*, 52(2):150–177, 1998.

[67] HweeHwa Pang, Michael J. Carey, and Miron Livny. Memory-adaptive external sorting. In *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, Rakesh Agrawal, Seán Baker, and David A. Bell, editors, pages 618–629. Morgan Kaufmann, 1993.

[68] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, 1987.

[69] A. Pothen, H. Simon, , and K-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis*, 11(3):430–452, 1990.

[70] V. Nageshwara Rao and Vipin Kumar. Parallel depth-first search on multiprocessors part I: Implementation. *International Journal of Parallel Processing*, 16(6):479–499, 1987.

[71] E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, January 2000.

[72] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[73] Y. Saad and M. Sosonkina. Non-standard parallel solution strategies for distributed sparse linear systems. In *Parallel Computation: Proc. of ACPC'99*, A. Uhl P. Zinterhof, M. Vajtersic, editor, Lecture Notes in Computer Science, Berlin, 1999. Springer-Verlag.

[74] Yousef Saad. SPARSKIT: A basic toolkit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990. Software currently available at <ftp://ftp.cs.umn.edu/dept/sparse/>.

[75] Yousef Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, March 1993.

[76] Yousef Saad. ILUT: a dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994. Technical Report 92-38, Minnesota Supercomputer Institute, University of Minnesota, 1992.

[77] Yousef Saad. *Iterative methods for sparse linear systems.* PWS Publishing Company, 1996.

[78] Horst D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1994.

[79] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: The Complete Reference.* MIT Press, 1995.

[80] A. Stathopoulos and J. R. McCombs. A parallel, block, Jacobi-Davidson implementation for solving large eigenproblems on coarse grain environments. In *1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 2920–2926. CSREA Press, 1999.

[81] A. Stathopoulos and J. R. McCombs. Parallel, multi-grain eigensolvers with applications to materials science. In *First SIAM Conference on Computational Science and Engineering*, 2000.

[82] A. Stathopoulos, Serdar Öğüt, Y. Saad, J. R. Chelikowsky, and Hanchul Kim. Parallel methods and tools for predicting material properties. *Computing in Science and Engineering*, 2(4):19–32, 2000.

[83] V. E. Taylor and B. Nour-Omid. A study of the factorization fill-in for a parallel implementation of the finite element method. *International Journal of Numerical Methods in Engineering*, 37:3809–3823, 1994.

[84] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms and Visualization*, James Abello and Jeffrey Scott Vitter, editors, pages 161–180. American Mathematical Society Press, Providence, RI, 1999.

[85] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: a practical scheduling scheme for parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–89, 1993.

[86] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys (CSUR)*, 33(2):209–271, 2001.

[87] C. Walshaw. Jostle: Graph partitioning and load balancing software. `http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/`.

[88] Michael S. Warren and John K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of Supercomputing '93*, pages 12–21, 1993.

[89] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proceedings of the IEEE/ACM Supercomputing Conference (SC'98)*, Orlando,FL, November 1998.

[90] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, pages 3–25, January 2001.

[91] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, 1999.

[92] WEIYE ZHANG AND PER-AKE LARSON. Dynamic memory adjustment for external mergesort. In *The VLDB Journal*, pages 376–385, 1997.

VITA

Richard Tran Mills

Richard Tran Mills was born on August 16, 1978 in Cookeville, Tennessee. His interest in computing began at the age of six when he was presented with a Commodore-64 personal computer for Christmas. After dropping out of Cookeville High School in 1995, he attended the University of Tennessee, Knoxville, as a Chancellor's Scholar, where he studied geology and physics and conducted research in computational geophysics. After graduating *summa cum laude* with the B.A. degree in May of 1999, he enrolled in the graduate program in computer science at the College of William & Mary, specializing in computational science. He received the M.S. degree in May of 2001. In addition to his research at William & Mary, he completed his DOE Computational Science Graduate Fellowship practicum experience at Los Alamos National Laboratory, where he worked with Dr. Peter Lichtner on developing codes for modeling multi-phase, subsurface fluid flow on parallel computers.