Manuscript
Click here to download Manuscript: paper.ps

Runtime and Programming Support for Memory Adaptation in Scientific Applications via Local Disk and Remote Memory

Richard T. Mills (rmills@ornl.gov)
Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN 37831

Chuan Yue (cyue@cs.wm.edu) and Andreas Stathopoulos (andreas@cs.wm.edu)
Department of Computer Science, College of William and Mary, Williamsburg, VA 23187-8795

Dimitrios S. Nikolopoulos (dsn@cs.vt.edu)
Department of Computer Science, Virginia Tech, 660 McBryde Hall, Blacksburg, VA 24061

Abstract. The ever increasing memory demands of many scientific applications and the complexity of today's shared computational resources still require the occasional use of virtual memory, network memory, or even out-of-core implementations, with well known drawbacks in performance and usability. In [24], we introduced a basic framework for a runtime, user-level library, MMLIB, in which DRAM is treated as a dynamic size cache for large memory objects residing on local disk. Application developers can specify and access these objects through MMLIB, enabling their application to execute optimally under variable memory availability, using as much DRAM as fluctuating memory levels will allow. In this paper, we first extend our earlier MMLIB prototype from a proof of concept to a usable, robust, and flexible library. We present a general framework that enables fully customizable, memory malleability in a wide variety of scientific applications. We provide several necessary enhancements to the environment sensing capabilities of MMLIB, and introduce a remote memory capability, based on MPI communication of cached memory blocks between 'compute nodes' and designated memory servers. The increasing speed of interconnection networks makes a remote memory approach attractive, especially at the large granularity present in large scientific applications. We show experimental results from three important scientific applications that require the general MMLIB framework. Their memory-adaptive versions perform nearly optimally under constant memory pressure and execute harmoniously with other applications competing for memory, without thrashing the memory system. Under constant memory pressure, we observe execution time improvements of factors between three and five over relying solely on the virtual memory system. With remote memory employed, these factors are even larger and significantly better than other, system-level remote memory implementations.

Keywords: memory management, shared computational pools, network RAM, scientific libraries, autonomic computing


© 2006 *Kluwer Academic Publishers. Printed in the Netherlands.*

1. Introduction

Commoditization of memory chips has enabled unprecedented increases in the memory available on today's computers and at rapidly decreasing costs. Manufacturing and marketing factors, however, keep the costs disproportionally high for larger memory chips. Therefore, although many shared computational resource pools or even large scale MPPs boast a large aggregate memory, only a small amount (relative to the high demands of many scientific applications) is available on individual processors. Moreover, available memory may vary temporally under multiprogramming. Sharing memory resources across processors is a difficult problem, particularly when applications cannot reserve sufficient memory for sufficiently long times. These realities pose tremendous problems in many memory demanding scientific applications.

To quantify the magnitude of the problem, we use a simple motivating example. We ran a parallel multigrid code to compute a three-dimensional potential field on four SMPs that our department maintains as a resource pool for computationally demanding jobs. Each SMP has 1 GB of memory. Since our code needed 860 MB per processor, we could run it using only one processor per node. While our code was running other users could run small jobs without interference. When a user attempted to launch Matlab to compute the QR decomposition of a large matrix on one of the processors in an SMP, the time per iteration in our multigrid code jumped from 14 to 472 seconds as virtual memory system thrashed. Most virtual memory systems would cause thrashing in this case, because their page replacement policies are not well suited for this type of scientific application.

Memory pressure can also be encountered on dedicated or space-shared COWs and MPPs. An important class of scientific applications runs a large number of sequential or low parallel degree jobs to explore a range of parameters. Typically, larger numbers of processors enable higher throughput computing. However, when memory is insufficient for individual jobs, such applications grind to a halt under virtual memory. On some MPPs, with small or no local disks, virtual memory is not adequate or even possible. Utilizing more processors per job to exploit larger aggregate memory may not be possible either, because the jobs are sequential or of limited parallel scalability. Application scientists often address these issues by implementing specialized out-of-core codes, and utilizing the parallel file systems that many MPPs employ. Besides the double performance penalty (data read from disk *and* propagated through the interconnection network), such codes often

lack flexibility, performing I/O even for data sets that could fit in memory.

Scenarios such as the above are one of the reasons for using batch schedulers, resource matchmakers, process migration and other techniques that warrant that each application will have enough memory to run without thrashing throughout its lifetime. These methods are not without problems. They may incur high waiting times for jobs, or high runtime migration overheads whenever there is resource contention. On certain platforms, such as shared-memory multiprocessors, some of these methods are not even applicable, as users may reserve more memory than their memory-per-CPU share [10]. A similar situation can occur if applications are allowed to use network RAM (NRAM) or other remote memory mechanisms [39]. Moreover, most remote memory research has been conducted at page-level granularity, which may incur unacceptably high latencies for page replacement [22].

The problem is equally hard at the sequential level. Page replacement policies of virtual memory systems are usually generic in nature and are ill-suited to access patterns encountered in many scientific applications. In addition, high seek times during thrashing cannot be amortized by prefetching, as it may be difficult for virtual memory system to predict the locality and pattern of block accesses on disk. On the other hand, compiler or user-provided hints would require modifications to the system.

To tame these problems we have developed a runtime library, MMlib (Memory Malleability library), that controls explicitly the DRAM allocations of specified large objects during the runtime of an application, thus enabling it to execute optimally under variable memory availability. MMlib allows applications to use customized, application-specific memory management policies, running entirely at user-level. To achieve portability and performance, MMlib blocks specified memory objects into *panels* and manages them through memory mapping. This provides programmers with a familiar interface that has no explicit I/O and can exploit the same common abstractions used to optimize code for memory hierarchies. Moreover, the advantages of running applications fully in-core are maintained, when enough memory is available. The library is designed for portability across operating systems and implemented in a non-intrusive manner.

In [24], we gave a proof of concept of MMlib based on a simplified framework, and developed a parameter-free algorithm to accurately ascertain memory shortage and availability strictly at user-level. In this paper, we first provide a general framework that enables memory malleability in a variety of scientific applications, and enhance MMlib's sensing capabilities to require no user input. Second, we introduce re-

mote memory capability into MMlib, based on MPI communication of panels between compute nodes and designated memory servers. Besides performance improvements on clusters with high speed networks, our flexible, user-level design enables a host of options such as multiple memory servers, and dynamic migration of panels between servers.

We see several benefits in this library-based approach for memory malleability. Injecting memory malleability into scientific applications allows them to run under memory pressure with degraded but acceptable efficiency, under a wider variety of execution conditions. Efficient memory adaptive applications can benefit both high-level batch schedulers, by letting them harness cycles from busy machines with idle memory, and operating system schedulers, by avoiding thrashing and thus securing continuous service to jobs. Also, as we show in this paper, the transparent design enables the library to implement remote memory when local disks are small or slower than fetching data from the network. MMlib is an optimization tool for DRAM accesses at the low level of the memory hierarchy (disk/network), but it can co-exist with and complement optimization tools for higher levels (memory/cache), enabling a unified approach to locality optimization in block-structured codes.

We present experimental results from three important scientific applications that we linked with MMlib. Besides their importance for scientific computing, these applications stress different aspects of MMlib and motivate the runtime optimizations presented in this work.

The rest of this paper is organized as follows: Section 2 reviews related work. Section 3 outlines our simplified framework and our parameter-free sensing algorithm from [24]. Section 4 presents the new extensions; general framework, runtime support and performance enhancements, and the remote memory implementation. Section 5 presents the applications along with experimental results. Section 6 concludes the paper.

2. Related work

Interesting memory usage and availability patterns on multiprogrammed clusters of workstations have been pointed out in a quantitative study by Acharya and Setia [3]. They show that, on average, more than half of the memory of the clusters is available for intervals between 5 and 30 minutes, with shorter intervals for larger memory requests. The study did not investigate mechanisms and policies for exploiting idle memory or the impact of fluctuations of idle memory on application performance.

Batch schedulers such as the Maui Scheduler [21], NQE [34], the Portable Batch System [17] and experimental systems [7, 32] as well as schedulers for privately owned networks of workstations and grids, such as Condor-G [16] and the GrADS scheduling framework [12], use admission control schemes which schedule a job only on nodes with enough memory. This avoids thrashing at the cost of reduced utilization of memory and potentially higher job waiting times. Other coarse-grain approaches for avoiding thrashing include checkpointing and migration of jobs. However, such approaches are not generally aware of the performance characteristics or the execution state of the program [38].

Co-scheduling attempts to keep all parallel processes of a job running at the same time, either explicitly [14], implicitly [4] or dynamically [36]. Beyond algorithmic and implementation difficulties, these approaches may compromise fairness and quality-of-service, especially in the not well studied situation of memory contention.

Chang et.al. [9] have presented a user-level mechanism for constraining the resident set sizes of user programs within a fixed range of available memory. They assume that the program has an a-priori knowledge of lower and upper bounds of the required memory range. This work does not consider dynamic changes to memory availability, nor does it address the problem of customizing the memory allocation and replacement policy to the memory access pattern of the application—both central issues in our research.

An approach that addresses a dynamically changing environment for general applications has been developed by Brown and Mowry [8]. This approach integrates compiler analysis, operating system support, and a runtime layer to enable memory-intensive applications to effectively use paged virtual memory. The runtime layer makes the appropriate memory allocation decisions by processing hints on the predicted memory usage that the compiler inserted. Although the approach has shown some good results, it requires modifications to the operating system. In addition, applications with complex memory access patterns can cause significant difficulties in identifying appropriate release points.

Barve and Vitter [6] presented a theoretical framework for estimating the optimal performance that algorithms could achieve if they adapted to varying amounts of available memory. They did not discuss implementation details or how system adaptivity can be achieved. Pang et al. [31] presented a sorting algorithm that dynamically splits and merges the resident buffer to adapt to changes in the memory available to the DBMS. This is a simulation-based study that does not discuss any details of the adaptation interface.

Remote memory servers have been employed in multicomputers [19] for jobs that exceed the available memory per processor. They also en-

abled the implementation of diskless checkpointing [33], a fault-tolerance scheme which exploits the speed of the interconnection network to accelerate saving and restoring of program state.

The advent of high-throughput computing on shared computational resources has motivated the design of NRAM systems for clusters of workstations [22, 20, 5, 15, 30]. Real implementations of NRAM and memory servers [5, 19, 13, 39] extend the operating system paging algorithms and provide support for consistency and fault tolerance at the page level. Though performance improvements have been reported over disk-based virtual memory systems, the page level granularity of memory management still incurs significant overheads, and thrashing can still occur. Moreover, such implementations require substantial changes to the operating system.

At the user-level, Nieplocha et al. [27] have developed the Global Arrays Toolkit that implements a distributed shared memory access to certain, user-specified arrays. Its design philosophy is different from ours, however, as many processes require access to the shared array, and at various levels of granularity. Global Arrays have been used to implement out-of-core computations via shared data structures that spill over onto disk [11], but this approach does not take dynamic memory availability into consideration.

Several researchers have utilized remote memory for implementing co-operative caching and prefetching in clustered web servers. Recently, Narravula et al. [26], exploit remote memory and direct remote DMA operations to improve utilization of aggregate distributed caches in a co-operative caching environment. Our work differs in that it employs remote memory in an application-controlled execution environment.

Koussih et al. [20] describe a user-level, remote memory management system called Dodo. Based on a Condor-like philosophy, Dodo harvests unused memory from idle workstations. It provides allocation and management of fine-grained remote memory objects, as well as user-defined replacement policies, but it does not include adaptation mechanisms for dynamically varying RAM, and cannot apply to local disks, or to remote memory servers that are not idle.

In [29], Nikolopoulos presented an adaptive scheduling scheme for alleviating memory pressure on multiprogrammed COWs, while coordinating the scheduling of the communicating threads of parallel jobs. That scheme required modifications to the operating system. In [28], the same author suggested the use of dynamic memory mapping for controlling the resident set of a program within a range of available physical memory. The algorithm operated at page-level granularity and allowed very little space for customization to the application access pattern. In [25], two of the authors of this paper followed

an application-level approach that avoided thrashing of an eigenvalue solver, by having the node under memory pressure recede its computation during the most computationally intensive phase, hopefully speeding the completion of competing jobs.

MMlib bears similarities with SHMOD (Shared-Memory on Disk) [40], an application-level asynchronous remote I/O library, which enables effective remote disk space usage, continuous application-level checkpointing and out-of-core execution. SHMOD is designed to support specifically a class of hydrodynamics applications and uses coarse-grain panels (called *things* in SHMOD's terminology), allocated and managed transparently across local and remote disks in a cluster of workstations. SHMOD organizes computation as a bag of tasks, with each task retrieving, working on and updating a thing. MMlib differs in that it utilizes remote DRAM instead of remote disks for faster retrieval of panels. On the other hand, unlike MMlib, SHMOD exploits cluster-wide storage and works with a task-parallel programming model.

In [24] we proposed MMlib as an application-level, memory management framework for scientific applications that perform repetitive data accesses. Using the main memory as cache and a user-defined replacement policy the application experiences a graceful degradation of performance as memory becomes scarce. The dynamic adaptation to available memory is performed by a system independent, parameter-free algorithm as described in [24]. In this paper we extend the applicability and functionality of the framework in a few important aspects: supporting multiple memory objects and multiple active panels at the same time; performing automatic accurate estimation of the size of the non-managed memory; and providing application level remote memory capability.

3. User-level adaptation

Application-level approaches are sometimes received with caution because of increased developer involvement. However, to exploit higher memory hierarchies, developers of scientific applications already block the accesses to the data or use libraries where such careful blocking is provided. Blocking for memory access is performed at a much larger granularity and thus complementary to cache access. Based on this, our approach in [24] considered the largest data object partitioned into P blocks, which in out-of-core literature are often called *panels*, and operated as follows:

for i = 1:P
Get panel $p_{pattern(i)}$ from lower level memory
Work with $p_{pattern(i)}$

Most scientific applications consist of code segments that can be described in this familiar to developers format. As long as the "get panel" encapsulates the memory management functionality, no code restructuring is ever needed.

On a dedicated workstation one can easily select between an in-core or an out-of-core algorithm and data structure according to the size of the problem. On a non-dedicated system though, the algorithm should adapt to memory variability, running as fast as an in-core algorithm if there is enough memory to store its entire data set, or utilizing the available memory to cache as many panels as possible. Based on memory mapped I/O, we provided a framework and supporting library for modifying codes for memory adaptivity that are portable to many applications and operating systems. Memory mapping has several advantages over conventional I/O because it avoids write-outs to swap space of read-only panels, integrates data access from memory and disk, allows for fine tuning of the panel size to hide disk latencies and facilitates an implementation of various cache replacement policies.

To get a new panel, the application calls a function from our library that also controls the number of panels kept in-core. At this point, the function has three choices: it can increase or decrease the number of in-core panels if additional, or respectively less, memory is available; or it can sustain the number of in-core panels if no change in memory availability is detected. The policy for selecting panels to evict is user defined as only the application has full knowledge of the access pattern.

Critical to this functionality is that our library be able to detect memory shortage and availability. However, the amount of total available memory is a global system information that few systems provide, and even then, it is expensive and with no consistent semantics. In [24], we developed an algorithm which relies only on measurements of the program's resident set size (RSS), a widely portable and local information. Memory shortage is inferred from a decrease in a program's RSS that occurs without any unmapping on the part of the program. Memory surplus is detected using a "probing" approach in which the availability of a quantity of memory is determined by attempting to use it and seeing if it can be maintained in the resident set. The algorithm is parameter-free, expecting only an estimate of the memory requirements of the program, excluding the managed panels. We call the size of this non-managed memory, *static memory* (sRSS). The algorithm detects memory availability, by probing the system at dynamically selected

time intervals, attempting to increase memory usage one panel at a time.

In [24] we demonstrated the effectiveness of our algorithm and used it to inject memory-malleability into an implementation of a conjugate-gradient linear solver. Our memory adaptation framework was limited, however, to a very specific class of applications with repeated, exhaustive passes through one read-only object. In section 3.1 we describe a more comprehensive framework that captures characteristics from a much larger set of applications, and in section 3.2 we give an abbreviated description of the new MMlib library that provides memory-malleability within this framework. In section 3.3 we explain how we overcome a key technical challenge, that of estimating the size of the *static memory* that is not managed by MMlib. In section 3.4 we describe an optimization in MMlib that allows it to deal with antagonistic page replacement by the operating system by adaptively evicting those panels that the system has already paged out.

3.1. A general framework

Scientific applications often work on many large memory objects at a time, with various access patterns for each object, sometimes working persistently on one panel, while other panels are only partially accessed, or even modified. A framework modeling the memory access needs such applications is shown in Figure 1.

Figure 1 depicts only one computation phase, which is repeated several times during the lifetime of the program. A computation phase denotes a thematic beginning and end of some computation, e.g., one iteration of the CG method or the two innermost of three nested loops. In this phase, a small number of memory objects are accessed (e.g., the matrix and the preconditioner in the CG algorithm), as their sheer size limits their number. In contrast to the previous simplified framework, we do not assume a sequential pass through all the panels of an object, although this is often the case in practice. In this context, a full sweep denotes a completion of the phase.

For each iteration of the computation phase, certain panels from certain memory objects need to be fetched, worked upon, and possibly written back. The iteration space in the current computation phase, the objects needed for the current iteration, and the access patterns for panels depend on the algorithm and can be described by the programmer. Finally, our new framework allows memory objects to fluctuate in size between different computation phases.

```
Identify memory objects M_1, M_2, ..., M_k
needed during this phase
for i = [ Iteration Space for all Objects ]
    for j= [ all Objects needed for iteration i ]
        panelID = accessPattern(M_j, i)
        Get panel or portion of panel (panelID)
    endfor
    Work on required panels or subpanels
    for j= [ all Objects needed for iteration i ]
        panelID = accessPattern(M_j, i)
        if panel panelID was modified
            Write Back(panelID)
        if panel panelID not needed persistently
            Release(panelID)
    endfor
endfor
```

Figure 1. Extended framework modeling the memory access needs of a wide variety of scientific applications. Although write-backs are represented explicitly in the framework, when panels are accessed via a named memory map, write-backs do not occur until a panel is evicted from main memory by the virtual memory system. When panels are accessed via remote memory, the write-back is performed explicitly at panel eviction.

3.2. Core MMlib interface and functionality

Based on the general framework, we have developed an object-based C library, MMlib, to provide memory-malleability while hiding all bookkeeping and adaptation decisions from the user. Here we give an abbreviated presentation of the core library interface and some of the technical issues involved; a more detailed discussion is described in detail in [23].

To be managed by MMlib, a given data-set must be broken into a user-specified number of panels for which a backing store is created on disk, accessed through memory mapping. The size of the panel is usually determined as a large multiple of the block size that is optimal for cache efficiency so that it also amortizes I/O seek times. In case of memory contention, a large number of panels can fine tune more accurately the exact level of available memory but incur higher bookkeeping and I/O overheads. Because of diminishing returns beyond a 5-10% accurate prediction of available memory, and because our goal is to match the performance of unmanaged in-core methods when running without memory contention, we suggest that about 10-40 panels be used per object.

MMS mmlib_new_mmstruct(type, *filename, P)
Each data-set and its panels are associated with an MMS object, which handles all necessary bookkeeping and through which all accesses to the data occurs. The above function constructs an MMS object of a given MMlib type. Type examples include MMLIB_TYPE_MATDENSE for a dense two dimensional array, or MMLIB_TYPE_VECTOR for a one dimensional array. The filename specifies the name of the backing store, and P is the number of the panels into which the data is broken.

void mmlib_set_update_queue(void (*func) (MMReg, MMS, int))
An MMS object is associated with a distinct priority queue. This queue orders the panels according to the eviction policy chosen by the user for that object. When more than one objects are active simultaneously, the choice of panel eviction must consider not only the intra- but also the inter-object priorities. For this reason, MMlib maintains a global registry of all MMS objects (MMReg), using this instead to make its adaptation decisions. 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 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.

void *mmlib_get_panel(MMS mms, p)
This function is the basic building block of the library. It returns a pointer to the beginning of panel p, hiding the rest of the bookkeeping. If the panel is already mapped, it returns its address and updates the global and corresponding local queues. If the panel is not mapped, it checks for memory shortage or surplus, consults the eviction policy and adjusts the number of panels in the queues accordingly.

void mmlib_release_panel(MMS mms, p)
Some applications work on many memory objects simultaneously, but not all objects have the same lifetime. In particular, certain panels may persist throughout the mapping and unmapping of other panels of the same or different objects. For example, assume we need to compute the interaction of a panel X_m with panels $X_i, i = 1, m-1$. It would be a performance disaster if, based on the MRU policy, we decided to

unmap this panel because it was recently accessed. In this case, the user needs to "lock" this panel as persistent, until all relevant computation is completed. In MMLIB, the pointer returned by `mmlib_get_panel` remains valid until the `mmlib_release_panel` is called. The release does not evict the panel; it merely unlocks it so that it can be evicted if deemed necessary.

3.3. Estimating static memory size

Our memory adaptation algorithm in [24] assumes that the program has an accurate estimate of the size of its *static memory*, i.e., memory not associated with managed objects. This is needed for calculating how much of the RSS belongs to the mapped objects. However, this size may not be easily computed or even available to the program if large enough static memory is allocated within linked, third party libraries. Moreover, for some programs the static memory may fluctuate between sweeps of the computation phase. A more serious problem arises when the static memory is not accessed during the computation phase. Under memory pressure, most operating systems consider the static memory least recently used and slowly swap it out of DRAM. This causes a false detection of memory shortage, and the unmapping of as many panels as the size of the swapped static memory.

An elegant solution to this problem relies on a system call named `mincore()` for most Unix systems and `VirtualQuery()` for Windows. The call allows a process to obtain information about which pages from a certain memory segment are resident in core. Because MMLIB can only ascertain residency of its managed memory objects, it uses `mincore()` to compute the actual resident size of the managed panels, which is exactly what our algorithm needs. Obviously, the use of this technique at every `get_panel` is prohibitive because of the overhead associated with checking the pages of all mapped panels. We follow a more feasible, yet equally effective strategy. We measure the residency of all panels (mRSS) in the beginning of a new computational phase, and derive an accurate estimate of the static memory: sRSS = RSS − mRSS, with RSS obtained from the system. As long as no memory shortage is detected, we use sRSS during the computation phase. Otherwise, we recompute mRSS and sRSS to make sure we do not unnecessarily unmap panels. Since unmapping is not a frequent occurrence the overall `mincore` overhead is tiny, especially compared to the slow down the code experiences when unmapping is required.

3.4. A most-missing eviction policy

One of the design goals of MMlib is to preempt the virtual memory system paging policy by holding the RSS of the application below the level at which the system will begin to swap out the pages of the application. Under increasing memory pressure, the paging algorithm of the system could be antagonistic by paging out data that MMlib tries to keep resident, thus causing unnecessary additional memory shortage. 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 our policy, say MRU. The rationale is that if the OS has evicted LRU pages, these will have to be reloaded either way, so we might as well evict the corresponding panels. Evicting MRU panels may make things worse because we will have to load the swapped out LRU pages as well as the MRU panels that we evicted.

The `mincore` functionality we described above facilitates the implementation of this "most missing" policy. This policy is not at odds with the user specified policy because it is only applied when memory shortage is detected, which is when the antagonism with the system policy can occur. Under constant or increasing memory availability the user policy is in effect. Preliminary results in section 5 show clear advantages with this policy.

4. Remote memory extension

Despite a dramatic increase in disk storage capacities, improvements in disk latencies have lagged significantly behind those of interconnection networks. Hence, remote virtual memory has often been suggested [22]. The argument is strengthened by work in [3, 2, 20] showing that there is significant benefit to harvesting the ample idle memory in computing clusters for data-intensive applications. The argument is imposing on MPPs, where parallel I/O must pass also through the interconnection network.

The general MMlib framework in Figure 1 lends itself naturally to a remote memory extension. The key modification is that instead of memory mapping a new panel from the corresponding file on disk, we allocate space for it and request it from a remote memory server. This server stores and keeps track of unmapped panels in its memory, while handling the mapping requests. In implementing this extension we had to address several design issues.

First, we chose MPI for the communication between processors, because it is a widely portable interface that users are also familiar with.

Also, because MMlib works entirely at user-level, we need the user to be able to designate which processors will play the role of remote memory servers. MMlib is not concerned with locating memory servers, and the long experience of some of the authors in scientific programming suggests that users are empowered, not burdened by exercising this control. We note, however, that nothing precludes the use of a system such as Condor or Dodo to suggest appropriate machines to use. The downside of using MPI is that the user must compile and run sequential programs with MPI. All other MPI set up and communications are handled internally in MMlib. For parallel programs there is no additional burden to the user. In the future, and if experience deems it necessary, a more transparent communication mechanism can be implemented with minimal change to MMlib. Nevertheless, our implementation is practically transparent, and it has provided a proof of concept for this functionality.

Second, because each panel is associated with a particular remote server, the executing process knows where to request it from and there is no need for consistency maintenance between replicas of panels. This allows the panels of one memory object to be kept on a number of servers. At the same time each server may be storing and handling panels from many objects, and possibly from many processors. Because of the large granularity, there is only a small number of panels, so the additional bookkeeping is trivial. This flexible design, which is reminiscent of home-based shared virtual memory research [18], enables a load balancing act between servers, that can migrate panels completely independently from the execution nodes. As long as the server pointer of each unmapped panel is updated in the corresponding managed memory object, execution nodes know where to direct their next request. An exploration of the many possibilities that arise from this design as well as more fine-grain consistency models is beyond the scope of this paper.

Third, the memory that will hold a remotely fetched panel does not have to be allocated with named memory mapping. Named memory mapping was important in the original MMlib as it was used to read a panel implicitly from a disk file, and because of that file it could avoid writing to the swap device under memory pressure. With remote memory, the existence of a file image for each panel is not required. We have explored the question of which allocation mechanism among `malloc()`, named `mmap()`, and anonymous `mmap()` provides the most benefits in performance and flexibility. Our experimental testbed and results, shown in the following section, yield anonymous `mmap()` as the best choice. In principle, memory mapping should be preferred because it permits the use of `mincore()` by MMlib to compute the

static memory size, and thus provide accurate sensing measurements for adaptation. On some operating systems `malloc()` is not implemented on top of `mmap()`, and thus does not permit the use of `mincore()`.

An outline of the remote memory algorithm follows. Initially, the memory server(s) load all the (initially unmapped) panels of all objects of the application into their memory. When a working process issues a `get_panel(mms,p)`, and the panel is not in memory (mapped), MMLIB sends a request to the appropriate server holding panel p of the object `mms`. If no panel is to be unmapped, MMLIB allocates the appropriate memory space and issues an `MPI_Recv`. If a panel, q, is to be unmapped, MMLIB figures out the server to send it to, and initiates an `MPI_Send`. When the send returns, this space of q can be reused to store the incoming panel p, so MMLIB simply issues an `MPI_Recv`. We should point out that if an object is designated as read-only, its panels need not be sent to the memory server when unmapped, provided that the server keeps all panels in its memory. Finally, the MMLIB framework retains all of its adaptivity to external memory pressure when our remote memory implementation is used in place of disk I/O.

5. Experiments

First we describe three applications that we modified to use MMLIB; their special characteristics require our extended framework and optimizations and cannot be implemented using the simple framework of [24]. Second, we present experiments with these applications under constant and variable memory pressure. Third, we explore experimentally the question posed in the previous section about the most appropriate allocation mechanism for remote memory. Finally we demonstrate the power of remote memory in MMLIB using the CG application.

5.1. The applications

The first application is the conjugate gradient (CG) linear system solver provided in SPARSKIT [35]. Each iteration of CG requires one sparse matrix-vector multiplication and a few inner products and vector updates. Our only managed object is the coefficient matrix, as it poses the bulk of the memory demands of the program, and is broken into 40 panels. CG also has a sizable amount of static memory for six work vectors. For MMLIB to work, this size must be known. In [24], we hard coded the size of this static memory. Here, we let MMLIB detect it dynamically. Our test code, `CG`, does not construct a matrix, but loads from disk a pre-generated sparse matrix in diagonal format. Figure

```
Algorithm: 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 2. 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.

2 depicts the algorithm for matrix-vector multiplication of a sparse matrix in this format; this is the algorithm in which MMlib is 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.

The second application is a modified Gram-Schmidt (MGS) orthogonalization procedure. A memory demanding application of MGS stems from materials science, where Krylov eigensolvers are used to find about 500–2000 eigenvectors for an eigenvalue problem of dimension on the order of one million [37]. Figure 3 depicts the algorithm executed by the `MGS` code. Our code simulates a Krylov solver (such as GMRES) except that it generates the recurrence randomly, not through 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,

Algorithm: MGS test code

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

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

Figure 3. 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.

and then appended to them. Only these vectors need to be managed by MMlib. In our experiment, we use one panel per vector for a total of 30 vectors, each of 3,500,000 doubles (80 MB total). 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; our improved MMlib is needed because the size of the managed object varies at runtime, and multiple panels are active simultaneously.

The third application is an implementation of the Ising model, which is used to model magnetism in ferromagnetic material, liquid-gas transition, and other similar physical processes. Considering a two-dimensional lattice of atoms, each of which has a spin of either up or down, the code runs a Monte-Carlo simulation to generate a series of configura-

tions that represent thermal equilibrium. The memory accesses follow a simple 5-point stencil pattern, common to many scientific applications. Figure 4 presents a pseudocode summary of the operation of our `ISING` code. For each iteration the code sweeps the lattice and tests whether to flip the spin of each lattice site. The flip is accepted, if it causes a negative potential energy change, ΔE. Otherwise, the spin flips with a probability equal to $\exp \frac{-\Delta E}{kT}$, where k is the Boltzman constant and T the ambient temperature. The higher the T, the more spins are flipped (equivalent to a "melting" of magnetic order or evaporation of liquids). In computational terms, T determines the frequency of writes to the lattice sites at every iteration. The memory-adaptive version partitions the lattice row-wise into 40 panels. 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. The improved MMlib framework is needed for panel write backs with variable frequency, and multiple active panels: 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` requires more than two panels to be active simultaneously, so that interactions across panel boundaries can be computed.

In all three applications the panel replacement policy is MRU, but there is also use of persistent (MGS) and neighboring (Ising) panels.

5.2. Adaptation via local disk

5.2.1. *Graceful degradation of performance*

Figure 5 includes one graph per application, each showing the performance of three versions of that application under constant memory pressure. Each point in the charts shows the execution time of one version of the application when run against a dummy job that occupies a fixed amount of physical memory, using the `mlock()` system call to pin its pages in-core. For each application we test a conventional in-core version (blue top curve), a memory-adaptive version using MMlib (red lower curve), and an ideal version (green lowest curve) in which the application fixes the number of panels cached at an optimal value provided by an oracle. The charts show the performance degradation of the applications under increasing levels of memory pressure. In all three applications, the memory-adaptive implementation performs consistently and significantly better than the conventional in-core implementation. Additionally, the performance of the adaptive code is very close to the ideal-case performance, without any advance knowledge of memory

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]
        ΔE = 2 · spin[i, j] · (left + right + up + down)
        if random() ≤ w[ΔE + 8] then
            spin[i, j] = -spin[i, j]
            E = E + ΔE
            M = M + 2 · spin[i,j]
        endif
    enddo
enddo
```

Figure 4. 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.

availability and static memory size, and regardless of the number of active panels (whether read-only or read/write).

5.2.2. *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 tested MMLIB-enabled `ISING` under constant memory pressure for different values of T. Figure 6 displays performance curves generated under static memory pressure on Linux 2.4 for temperatures $T = 0$, $T = 2$, and $T = 50$. At $T = 0$,

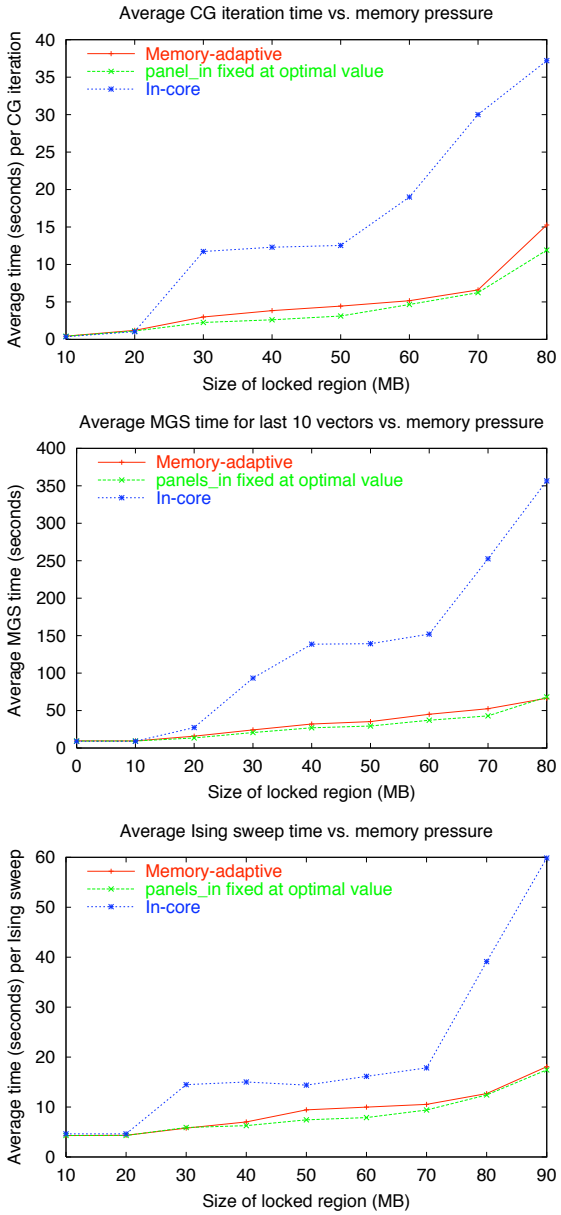

Figure 5. Performance under constant memory pressure. The top chart shows the average time per iteration of CG with a 70 MB matrix, which requires a total of 81 MB of RAM including memory for the work vectors. The middle chart shows the time to orthogonalize via modified Gram-Schmidt the last 10 vectors of a 30 vector set. Approximately 80 MB are required to store all 30 vectors. The bottom chart shows the time required for an Ising code to sweep through a 70 MB lattice. All experiments were conducted on a Linux 2.4.22-xfs system with 128 MB of RAM, some of which is shared with the video subsystem.

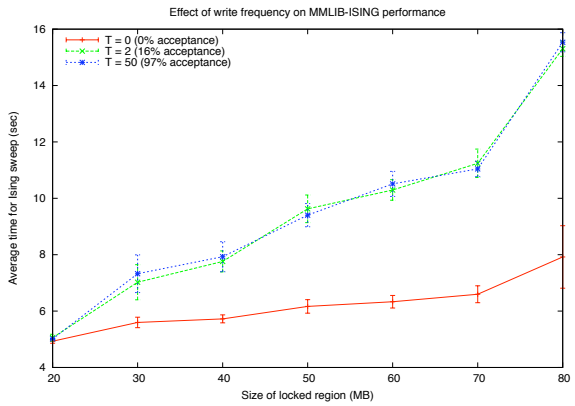

Figure 6. Effects of write frequency on MMLIB performance in `ISING`. `ISING` runs with a 70 MB spin lattice against static memory pressure applied via `memlock` on a Linux 2.4 system with 128 MB RAM. $P = 20$, $R_{pen\_max} = 10$, and low frequency probing is used. Performance curves are shown for temperatures $T = 0$, $T = 2$, 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.

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 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 5, 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 5 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.)

5.2.3. *Quick response to transient memory pressure*

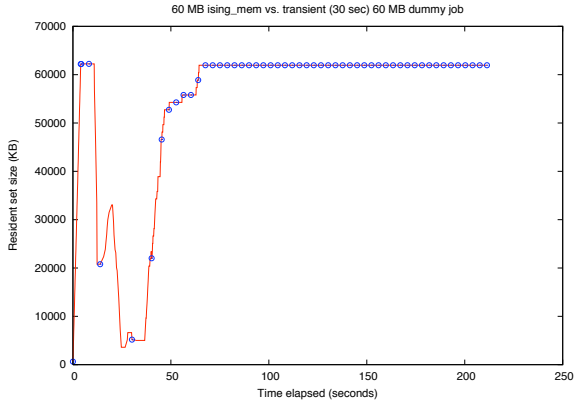

Figure 7. Adaptation to transient memory pressure. A memory-adaptive Ising job with a 60 MB lattice begins running on a Linux 2.4.22-xfs system with 128 MB of RAM. Circles in the figure represent sweeps through the lattice. 12 seconds later a competing jobs starts and writes randomly to a 60 MB region of RAM for 30 seconds.

Our goal is for MMLIB-enabled applications to not only exhibit graceful performance degradation under memory pressure, but also to respond quickly to changes in memory availability. To verify that this is the case, we performed a test in which we started a memory-adaptive Ising model computation, allowed it to complete a few sweeps through its lattice, and then applied transient memory pressure in the form of a competing memory-intensive job that ran for 30 seconds. The results or this experiment are depicted in Figure 7 and show that the job quickly adapts its resident set size to a safe level at the onset of

memory pressure and, furthermore, readily adjusts its memory usage back to normal when the competing job finishes.

5.2.4. *Adaptive versus adaptive jobs*
The litmus test for MMlib is when multiple instances of applications employing the library are able to coexist on a machine without thrashing the memory system. Figure 8 shows the resident set size (RSS) over time for two instances of the memory adaptive Ising code running simultaneously on a Sun Ultra 5 node. After the job that starts first has completed at least one sweep through the lattice, the second job starts. Both jobs have 150 MB requirements, but memory pressure varies temporally. The circles in the curves denote the beginning of lattice sweeps. Distances between consecutive circles along a curve indicate the time of each sweep.

The results show that the two adaptive codes run together harmoniously without constantly evicting each other from memory and the jobs reach an equilibrium where system memory utilization is high and throughput is sustained without thrashing. The system does not allow more than about 170 MB for all jobs, and it tends to favor the application that starts first. A similar phenomenon was observed in Linux. We emphasize that the intricacies of the memory allocation policy of the OS are orthogonal to the policies of MMlib. MMlib allows jobs to utilize as efficiently as possible the available memory, not to claim more memory than what is given to each application by the OS.

5.2.5. *Performance under a most-missing eviction policy*
Figure 9 shows the benefits of our proposed "most-missing" eviction policy. After external memory pressure starts, the job that uses strict MRU eviction exhibits very slow performance initially, 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 operating 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.

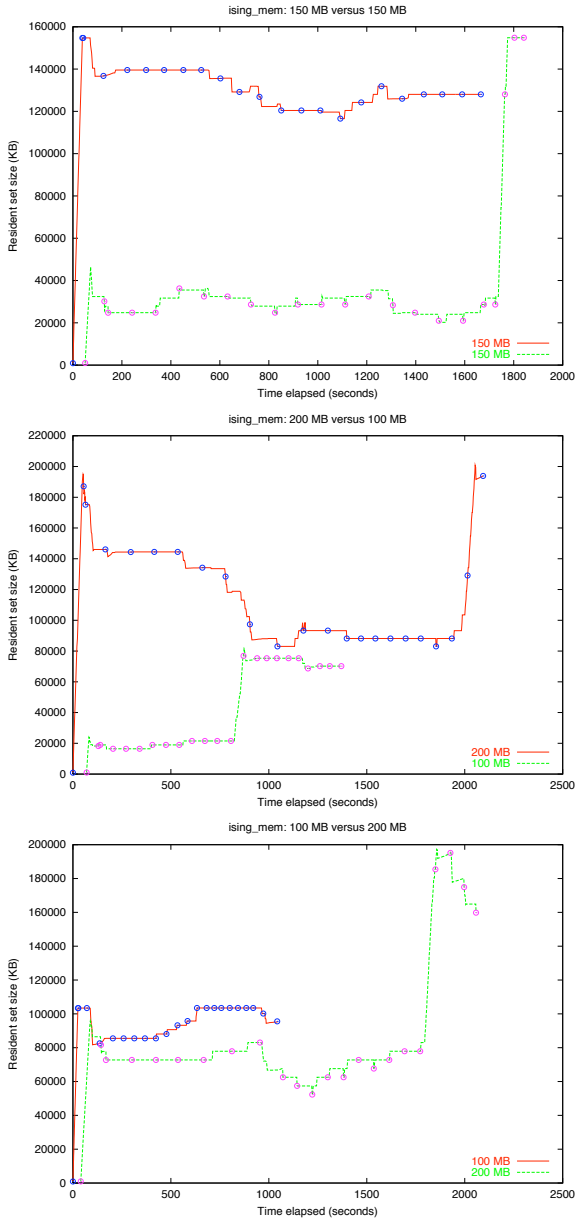

Figure 8. Running adaptive versus adaptive jobs. The top chart plots RSS vs. time for two equal-sized (150 MB) memory-adaptive Ising jobs. The second job is started 30 seconds after the first job. The chart in the middle plots the performance of a small (100 MB) and a large (200 MB) Ising job. The small job starts 70 seconds after the large job. The bottom chart plots the performance of a small and a large job (100 MB and 200 MB respectively), where the large job starts 40 seconds after the small job.

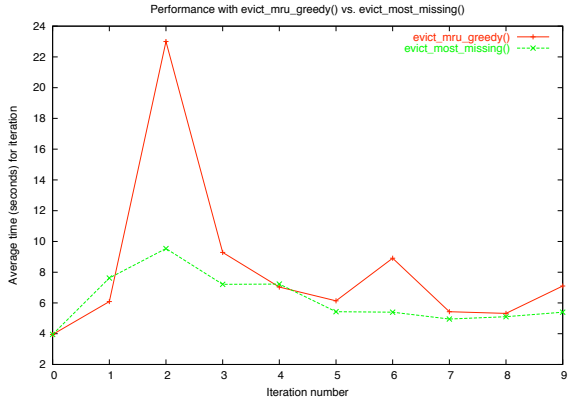

Figure 9. Benefits of evicting partially swapped out panels. The chart shows the times for the first 10 sweeps of a memory-adaptive Ising job running with a 60 MB lattice on a Linux machine with 128 MB of RAM. After 12 seconds, a dummy job that uses 60 MB of memory begins. The job that evicts panels with the largest number of missing pages, labeled `evict_most_missing`, has lower and less variable times than the original MRU replacement, labeled `evict_mru_greedy`.

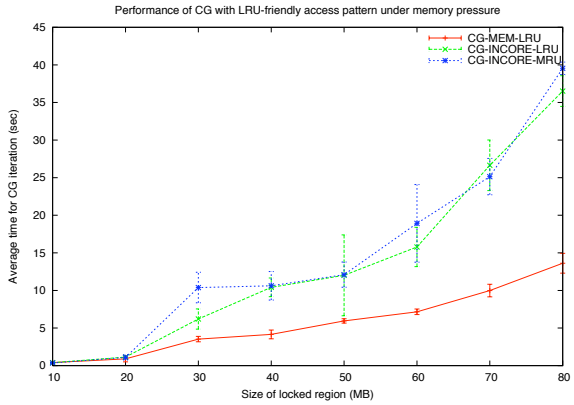

Figure 10. Performance of in-core and memory adaptive versions of `CG-LRU`, which uses an LRU-friendly access pattern. Jobs run under static memory pressure provided by `memlock` on a Linux 2.4 system with 128 MB of RAM. The performance profile for in-core, MRU-friendly `CG` is also depicted for comparison.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

5.2.6. *Dependence of performance gains on replacement policy*

Because all three of the applications we have tested employ memory access patterns for which MRU replacement is appropriate (and are thus very poorly served by the LRU-like algorithms employed by virtual memory systems), one might wonder if all of the performance gains

provided by MMLIB are attributable to the MRU replacement that it enables. To test this notion, we devised two experiments.

In the first experiment, we modified the `CG` code to utilize an LRU-friendly access pattern. Normally, `CG` performs matrix-vector multiplications starting with the first row of the matrix and proceeding down the rows until the last row is processed. Similarly for the memory-adaptive case, at each `CG` iteration, a sweep from the first (top) to the last (bottom) panel of the matrix 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 10 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`, implying that the improved access pattern of `CG-LRU` is immaterial. Tracing the reasons for this is challenging because the system activity that occurs during thrashing can be quite complicated. Although 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, it is likely that its poor performance also stems from problems in the page replacement policy of the operating system. The LRU-like page replacement policies employed by many operating systems are prone to thrashing given certain small disruptions in the LRU access pattern. Although `CG-LRU` uses an LRU-friendly pattern to access the matrix, this modification does not extend to the work vectors (static memory), which cannot be protected from page reclamation in the same way they are by MMLIB-enabled `CG` codes; this may explain at least part of the poor performance observed.

In our second experiment, we tested what performance gains MMLIB could still provide when used with an inappropriate replacement policy: We ran the standard (MRU-friendly) MMLIB-enabled `CG` under memory pressure and instructed MMLIB to use LRU replacement. This introduces a serious performance bug in the replacement policy — in the presence of memory pressure, *all* panel fetches will result

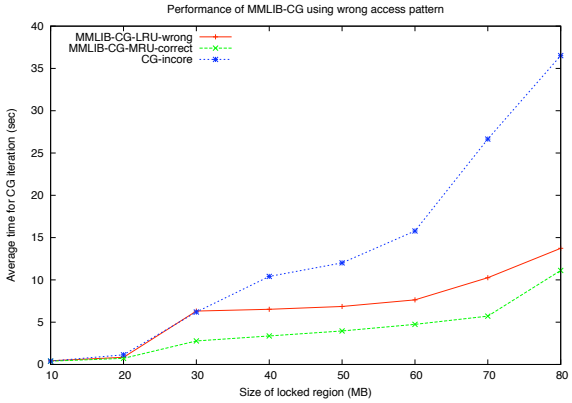

Figure 11. Performance of memory-adaptive CG versus memory pressure when using the wrong panel replacement policy, which is depicted by the curve labeled MMLIB-CG-LRU-wrong. The appropriate replacement policy is MRU, but LRU is used instead; any observed performance gains are not due to the ability of MMLIB to allow application-specific replacement. For comparison, the performance of memory-adaptive correctly employing MRU is depicted by the curve labeled MMLIB-CG-MRU-correct. Jobs run under static memory pressure provided by locking a region of memory in-core on a Linux 2.4 system with 128 MB of RAM.

in a miss! Consequently, any performance benefits observed will be unrelated to the ability of MMLIB to enable the use of application-specific replacement policies. Figure 11 compares the performance of `CG` using the wrong (LRU) replacement policy with the `CG` correctly employing MRU. The version using LRU replacement performs markedly 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 code using the wrong replacement policy 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. The performance benefits in this case come strictly from the large granularity of panel access, as opposed to the page-level granularity of the virtual memory system, and from avoiding thrashing. Our experiments also suggest that for a good replacement policy to offer additional benefits, the code must have structured, controlled memory accesses.

5.3. Adaptation via remote memory

The remote memory experiments are conducted on the SciClone cluster [1] at William & Mary. All programs are linked with the MPICH_GM package and the communications are routed via a Myrinet 1280 switch.

We use dual-cpu Sun Ultra 60 workstations at 360MHz with 512MB memory, of which about 80MB are reserved by the Solaris 9 system.

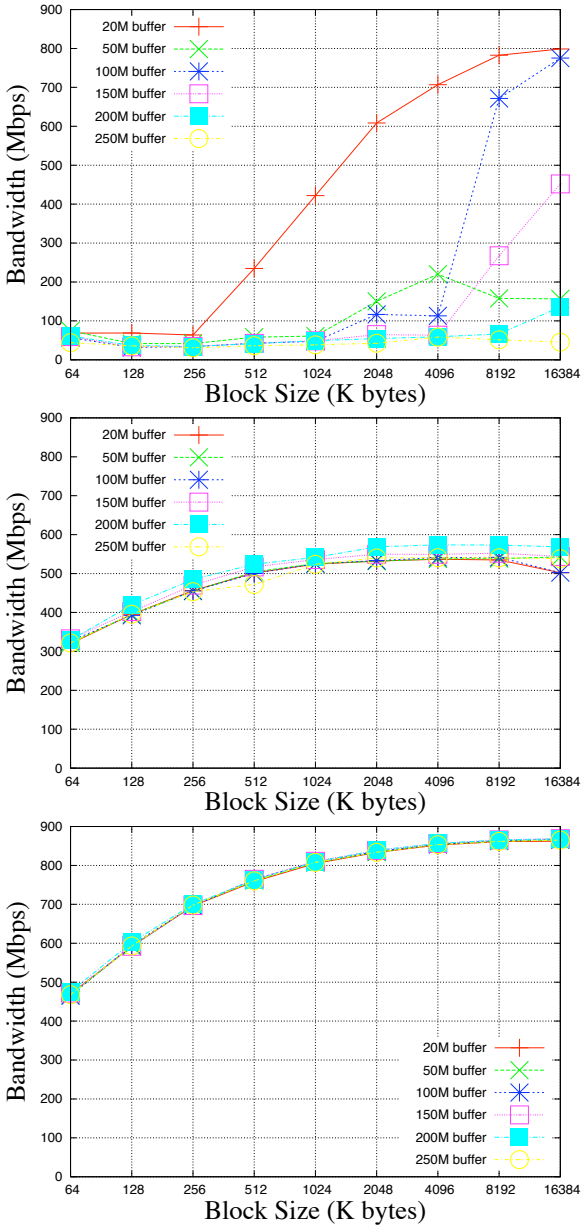

Figure 12. MPI_Recv perceived bandwidth microbenchmark results. The charts show the perceived MPI_Recv bandwidth vs. receiving block size for six different total buffer sizes. The receiving block is allocated using named `mmap()` (top chart), anonymous `mmap()` (middle chart), or `malloc()` (bottom chart).

5.3.1. *Microbenchmark results*
These experiments help us understand the effect of various allocation schemes (`malloc`, named `mmap`, anonymous `mmap`) on the MPI_Recv performance, under various levels of memory pressure. Figure 12 shows three graphs corresponding to the three methods for allocating the receiving block of the MPI_Recv call. Each graph contains six curves plotting the perceived MPI_Recv bandwidth for six different total 'buffers' that the MPI_Recv tries to fill by receiving 'Block Size' bytes at a time. This microbenchmark simulates an MMlib process that uses remote memory to bring each one of the panels (of 'Block Size' each) of a memory object (of 'buffer' size). On the same node with the receiving process, there is a competing process reading a 300MB file from the local disk. The larger the total 'buffer' size, the more severe the memory pressure on the node. The remote memory server process is always ready to send the requested data.

The top chart suggests that named mapping is the worst choice for allocating MPI_Recv buffers, especially under heavy memory pressure, which is exactly when MMlib is needed. In fact when the receiving block is small, performance is bad even without memory pressure (e.g., all curves for block size of 256 KB). With large block sizes bandwidth increases but only when the total 'buffer' does not cause memory pressure (e.g., the 20MB 'buffer' curve). A reason for this is that receiving a remote panel causes a write-out to its backing store on the local disk, even though the two may be identical.

For both the middle and bottom charts, all six curves are very similar, suggesting that allocation using anonymous `mmap()` or `malloc()` is not very sensitive to memory pressure. The MPI_Recv bandwidth performance of using `malloc()` for memory allocation is better than that of using anonymous `mmap()`. As no other processes were using the network during the experiments, the perceived differences in bandwidth must be due to different mechanisms of Solaris 9 for copying memory from system to user space.

Although these microbenchmarks suggest `malloc()` as the allocator of choice for implementing remote memory, experiments with CG favor anonymous mapping, with `malloc()` demonstrating unpredictable behavior.

5.3.2. *CG application results*
To demonstrate the remote memory capability of MMlib, we performed two sets of experiments with the CG application on the same computing platform as in microbenchmark experiments.

In the first set, the MMlib enabled CG application runs on one local node. It works on a 200MB matrix, which is equally partitioned into 20

Table I. This table shows the wall-clock time for MMlib CG running against in-core CG, in six modes corresponding to the six rows in the table. MMlib CG works on local node on a 200MB matrix, which is the managed object, requiring a total of 263MB. Memory pressure on the local node is created by in-core CG running on three different matrix sizes: 300MB, 150MB, 100MB, with total memory requirements: 385MB, 194MB, and 128MB respectively. Without Pressure means in-core CG is not running.

Wall-clock time for MMlib CG against in-core CG

mode \ memory pressure	300MB	150MB	100MB	No Pressure
named `mmap()` local	388.097	326.642	309.365	285.135
named `mmap()` remote	484.34	472.831	371.277	289.617
anonymous `mmap()` local	541.005	367.357	317.726	294.406
anonymous `mmap()` remote	379.114	360.516	317.756	293.213
`malloc()` local	1325.485	1023.782	1059.640	1050.507
`malloc()` remote	534.229	504.043	475.662	462.117

panels. We create various levels of memory pressure on the local node by running in addition the in-core CG application (without MMlib) with matrices requiring 300MB, 150MB, and 100MB memory sizes. The experimental results are shown in Table I. There are two rows of data for each of the three memory allocation methods. In the first row, we run the MMlib CG without remote memory; in the second row, we run MMlib CG with remote memory capability.

First, we see that under named `mmap()`, performance for remote mode is inferior to local mode, while under anonymous `mmap()` mode and `malloc()`, remote mode is obviously superior to local mode, especially when memory pressure is severe. The results also confirm the microbenchmark observations that named `mmap()` is the wrong choice for remote memory.

In contrast to the microbenchmark results, however, remote mode performance is better under anonymous `mmap()` than under `malloc()`. There are two reasons for this. The most important reason is that on Solaris 9 `malloc()` extends the data segment by calling `brk()`, rather than by calling `mmap()`. Because MMlib issues a series of `malloc()` and `free()` calls, especially when there is heavy memory pressure, the unmapped panels may not be readily available for use by the system. This causes the runtime scheduler to think that there is not enough memory and thus to allocate less resources to the executing process. In our experiments on the dual cpu Suns, we noticed that the Solaris

Table II. This table shows the wall-clock time for MMLIB CG running against MMLIB CG. Both MMLIB CG applications use anonymous `mmap()` to allocate memory. The first three rows show the results when local disk is used by both MMLIB CG applications. The following three rows show the results when remote memory is used by both MMLIB CG applications. The last row shows the total wall-clock time reduction for remote over local mode. Actual memory requirements for all codes are about 28% more than the matrix size.

Wall-clock time for MMLIB CG against MMLIB CG

processes \ matrix size	300MB	250MB	200MB	150MB	100MB
*cg*1 local	1496.700	1116.355	413.500	265.929	181.252
*cg*2 local	1015.495	755.859	638.448	266.697	185.796
Total local	2512.195	1872.214	1051.948	532.626	367.048
*cg*1 remote	1139.011	815.240	415.056	266.049	158.954
*cg*2 remote	809.446	519.031	603.588	252.822	157.289
Total remote	1948.457	1334.271	1018.644	518.871	316.243
Time reduction of remote over local	22.4%	28.7%	3.16%	2.58%	13.8%

scheduler gave less than 25% cpu time to the application in `malloc()` mode, while it gave close to 50% cpu time to the one in anonymous `mmap()` mode. Interestingly, the local MMLIB implementation without remote memory demonstrates even a worse behavior, suggesting that, on Solaris, the use of malloced segments should be avoided for highly dynamic I/O cases. The second reason is that the `mincore()` system call does not work on memory segments allocated by `malloc()`. Therefore, MMLIB cannot obtain accurate estimates of the static memory to adapt to memory variabilities.

In the second set of the experiments, we let two MMLIB CG applications run against each other to measure the performance advantages of using remote memory over local disk in a completely dynamic setting. Both MMLIB CG applications use anonymous `mmap()` to allocate memory and work on different matrices of equal size. Each matrix is equally partitioned into 10MB panels. The experimental results are shown in Table II. When remote memory instead of local disk is used, the total wall-clock time of the two MMLIB CG applications is always reduced. Especially when the overall memory pressure is severe such as the 300MB matrix and 250MB matrix cases (the overall memory requirements for these matrices are 750MB and 640MB respectively), the wall-clock time reduction can be 22.4% and 28.7% respectively.

We emphasize that these improvements are *on top* of the improvements provided by the local disk MMlib over the simple use of virtual memory. Considering also the improvements from Figure 5, our remote memory library improves local virtual memory performance by a factor of between four and seven. This compares favorably with factors of two or three reported in other remote memory research [20].

6. Conclusions

We presented a general framework and supporting library that allows scientific applications to automatically manage their memory requirements at runtime, thus executing optimally under variable memory availability. The library is highly transparent, requiring minimal code modifications and only at a large granularity level.

This paper extends our previous simplified framework and adaptation algorithm for memory malleability with the following key functionalities: (a) multiple and simultaneous read/write memory objects, active panels, and access patterns, (b) automatic and accurate estimation of the size of the non-managed memory, and (c) application level remote memory capability.

We showed how each of the new functionalities (a) and (b) were necessary in implementing three common scientific applications, and how (c) has significant performance advantages over system-level remote memory. Moreover, the remote memory functionality has opened a host of new possibilities for load and memory balancing in COWs and MPPs, that will be explored in future research. Our experimental results with MMlib, have confirmed its adaptivity and near optimality in performance. Integration of MMlib with high-level parallel programming models will also be explored in future work.

7. Acknowledgments

This work is supported by the National Science Foundation (ITR/ACS-0082094, ITR/AP-0112727, ITR/ACI-0312980, CAREER/CCF-0346867) the Department of Energy (DOE-FG-02-05ER2568), a DOE computational sciences graduate fellowship, the College of William and Mary and the College of Engineering at Virginia Tech. Part of this work was performed using the SciClone clusters at the College of William and Mary which were enabled by grants from Sun Microsystems, the NSF, and Virginia's Commonwealth Technology Research Fund.

References

1. SciClone cluster project at the College of William and Mary. 2005.
2. A. Acharya, G. Edjlali, and J. Saltz. The utility of exploiting idle workstations for parallel computation. volume 25, pages 225–234, 1997.
3. A. Acharya and S. Setia. Availability and Utility of Idle Memory in Workstation Clusters. In *Proc. of the 1999 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'99)*, pages 35–46, Atlanta, Georgia, May 1999.
4. A. Arpaci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. *ACM Transactions on Computer Systems*, 19(3):283–331, August 2001.
5. A. Barak and A. Braverman. Memory Ushering in a Scalable Computing Cluster. *Journal of Microprocessors and Microsystems*, 22(3–4):175–182, August 1998.
6. 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.
7. A. Batat and D. Feitelson. Gang Scheduling with Memory Considerations. In *Proc. of the 14th IEEE International Parallel and Distributed Processing Symposium (IPDPS'2000)*, pages 109–114, Cancun, Mexico, May 2000.
8. Angela Demke Brown and Todd C. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 31–44, 2000.
9. F. Chang, A. Itzkovitz, and V. Karamcheti. User-Level Resource Constrained Sandboxing. In *Proc. of the 4th USENIX Windows Systems Symposium*, pages 25–36, Seattle, WA, August 2000.
10. S. Chiang and M. Vernon. Characteristics of a Large Shared Memory Production Workload. In *Proc. 7th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'2001), Lecture Notes in Computer Science, Vol. 2221*, pages 159–187, Cambridge, MA, June.
11. Holger Dachsel, Jarek Nieplocha, and Robert Harrison. An out-of-core implementation of the COLUMBUS massively-parallel multireference configuration interaction program. In *Proceedings of Supercomputing '98*, 1998.
12. H. Dail, H. Casanova, and F. Berman. A Decoupled Scheduling Approach for the GrADS Program Development Environment. In *Proc. of the IEEE/ACM Supercomputing'02: High Performance Networking and Computing Conference (SC'02)*, Baltimore, MD, November 2002.
13. M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, , and C. Thekkat. Implementing global memory management in a workstation cluster. In *15th ACM Symposium on Operating Systems Principles(SOSP-15)*, pages 201–212, 1995.
14. D. Feitelson and L. Rudolph. Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control. *Journal of Parallel and Distributed Computing*, 35(1):18–34, May 1996.
15. M. Flouris and E. Markatos. Network RAM. In *High Performance Cluster Computing*, pages 383–408. Prentice Hall, 1999.
16. J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proc. of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10)*, pages 55–63, San Francisco, California, August 2001.

17. R. Henderson. Job Scheduling Under the Portable Batch System. In *Proc. of the First Workshop on Job Scheduling Strategies for Parallel Processingolph, Lecture Notes in Computer Science Vol. 949*, pages 279–294, Santa Barbara, CA, April 1995.
18. L. Iftode. *Home-Based Shared Virtual Memory*. PhD thesis, Princeton University, June 1998.
19. L. Iftode, K. Petersen, and K. Li. Memory Servers for Multicomputers. In *Proc. of the IEEE 1993 Spring Conference on Computers and Communications (COMPCON'93)*, pages 538–547, February 1993.
20. S. Koussih, A. Acharya, and S. Setia. Dodo: A user-level system for exploiting idle memory in workstation clusters. In *HPDC*, 1999.
21. M. Lewis and L. Gerner. Maui Scheduler, an Advanced System Software Tool. In *Proc. of the ACM/IEEE Supercomputing'97: High Performance Networking and Computing Conference (SC'97)*, San Jose, CA, November 1997.
22. Evangelos P. Markatos and George Dramitinos. Implementation of a reliable remote memory pager. In *USENIX Annual Technical Conference*, pages 177–190, 1996.
23. R. T. Mills. *Dynamic adaptation to CPU and memory load in scientific applications*. PhD thesis, Department of Computer Science, College of William and Mary, Fall, 2004.
24. R. T. Mills, A. Stathopoulos, and D. S. Nikolopoulos. Adapting to memory pressure from within scientific applications on multiprogrammed COWs. In *International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santq Fe, NM, USA, 2004.
25. 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.
26. S. Narravula, H. Jin, K. Vaidyanathan, and D. Panda. Designing Efficient Cooperative Caching Schemes for Multi-Tier Data Centers over RDMA-Enabled Networks. In *Proc. of the 6th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 401–408, Singapore, May 2006.
27. Jarek Nieplocha, Manojkumar Krishnan, Bruce Palmer, Vinod Tipparaju, and Yeliang Zhang. Exploiting processor groups to extend scalability of the GA shared memory programming model. In *ACM Computing Frontiers, Italy, 2005*, 2005.
28. 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 Symposin (IPDPS'2003)*, Nice, France, April 2003.
29. D. Nikolopoulos and C. Polychronopoulos. Adaptive Scheduling under Memory Pressure on Multiprogrammed Clusters. In *Proc. of the 2nd IEEE/ACM International Conference on Cluster Computing and the Grid (ccGrid'02)*, pages 22–29, Berlin, Germany, May 2002.
30. J. Oleszkiewicz, L. Xiao, and Y. Liu. Parallel network RAM: Effectively utilizing global cluster memory for large data-intensive parallel programs. In *2004 International Conference on Parallel Processing (ICPP'2004)*, pages 353–360, 2004.
31. HweeHwa Pang, Michael J. Carey, and Miron Livny. Memory-adaptive external sorting. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors,

19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings, pages 618–629. Morgan Kaufmann, 1993.

32. F. Petrini and W. Feng. Time-Sharing Parallel Jobs in the Presence of Multiple Resource Requirements. In *Proc. of the 6th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'2000), in conjunction with IEEE IPDPS'2000, LNCS Vol. 1911*, pages 113–136, Cancun, Mexico, May 2000.
33. J. Plank, K. Li, and M. Puening. Diskless Checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.
34. R. Daugherty and D. Ferber. Network Queuing Environment. In *Proceedings of the Spring Cray Users Group Conference (CUG'94)*, pages 203–205, San Diego, CA, March 1994.
35. 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/>.
36. P. Sobalvarro, S. Pakin, W. Weihl, and A. Chien. Dynamic Coscheduling on Workstation Clusters. In *Proc. of the 4th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'98), Lecture Notes in Computer Science Vol. 1459*, pages 231–256, Orlando, Florida, April 1998.
37. 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.
38. S. Vadhiyar and J. Dongarra. A Performance Oriented Migration Framework for the Grid. Technical Report, Innovative Computing Laboratory, University of Tennessee, Knoxville, 2002.
39. G. Voelker, E. Anderson, T. Kimbrel, M. Feeley, J. Chase, A. Karlin, and H. Levy. Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System. pages 33–43, Madison, Wisconsin, June 1999.
40. P. Woodward, S. Anderson, D. Porter, and A. Iyer. Distributed Computing in the SHMOD Framework on the NSF TeraGrid. Technical report, Laboratory for Computational Science and Engineering, University of Minnesota, February 2004.