Adapting to memory pressure from within scientific applications on multiprogrammed COWs [*]

Richard T. Mills, Andreas Stathopoulos, and Dimitrios S. Nikolopoulos
Department of Computer Science
College of William and Mary
Williamsburg, Virginia 23187-8795
rtm/andreas/dsn@cs.wm.edu

Abstract

Dismal performance often results when the memory requirements of a process exceed the physical memory available to it. Moreover, significant throughput reduction is experienced when this process is part of a synchronous parallel job on a non-dedicated computational cluster. A possible solution is to develop programs that can dynamically adapt their memory usage according to the current availability of physical memory. We explore this idea on scientific computations that perform repetitive data accesses. Part of the program's data set is cached in resident memory, while the remainder that cannot fit is accessed in an "out-of-core" fashion from disk. The replacement policy can be user defined. This allows for a graceful degradation of performance as memory becomes scarce. To dynamically adjust its memory usage, the program must reliably answer whether there is a memory shortage or surplus in the system. Because operating systems typically export limited memory information, we develop a parameter-free algorithm that uses no system information beyond the resident set size (RSS) of the program. Our resulting library can be called by scientific codes with little change to their structure or with no change at all, if computations are already "blocked" for reasons of locality.

Experimental results with both sequential and parallel versions of a memory-adaptive conjugate-gradient linear system solver show substantial performance gains over the original version that relies on the virtual memory system. Furthermore, multiple instances of the adaptive code can coexist on the same node with little interference with one another.

[*]Work supported by the National Science Foundation (ITR/ACS-0082094 and ITR/AP-0112727), a DOE Computational Science Graduate Fellowship, and the College of William and Mary.

1 Introduction

Powerful, yet cost efficient, clusters of workstations (COWs) bear the brunt of the scientific computing workload at many institutions. These can be dedicated, space-shared COWs, or, quite often, networks of desktops used as a shared computational resource for parallel and sequential jobs. Besides increased computational power, these environments also address the ever increasing memory demands of scientific applications. However, COWs are often shared by one or more research groups, and networks of workstations by the pool of local users. In periods of high demand, (approaching deadlines, end of semester, etc.) time sharing the limited memory resources on these environments can have particularly adverse effects on the effectiveness of the system.

An example of such adverse effects is our experience with running a large, parallel multigrid code to compute a three-dimensional potential field on four SMPs that our department maintains to support computationally demanding jobs. Our code required 860MB per processor. Because each SMP node had 1GB available, we used only one processor per node. Other users were running smaller jobs at the same time without interference. However, when a user attempted to use Matlab to compute the QR decomposition of a large matrix on one of the processors, the time for one iteration of our code jumped from 14 seconds to 472 seconds! Such thrashing is a familiar scenario to many researchers that rely on similar shared environments.

In the presence of memory pressure on a node, the local operating system usually chooses one of the following two strategies. It may swap out some of the competing processes to enable the remaining processes to fully utilize the resources and finish earlier, thus improving the throughput of the node. However, if the swapped out process happens to be part of a parallel job that requires frequent synchronizations, the job experiences extreme increase in response

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

time. Alternatively, the system may choose to time share all jobs, leading to thrashing, low CPU utilization, high response time for all jobs and thus low throughput. The problem is equally severe on SMPs where memory pressure may not coincide necessarily with additional CPU load, and even for sequential jobs on time shared compute servers.

A typical solution to these problems is not to multiprogram the computational nodes but to enforce admission control, usually through some centrally administered batch queue [14, 8, 9]. This solution, adopted in most supercomputing centers, assumes full availability of resources (including CPUs and memory) for a job to commence execution. The most common problem of these schedulers is that jobs may suffer high slowdown compared to standalone parallel execution, due to long waiting times in the queues. It may also be problematic in systems that use SMPs: Although the processors of an SMP can be partitioned between jobs, each job has access to the entire physical memory available on the machine. In fact, many users prefer to send their jobs to large-scale SMPs, such as the NCSA Origin2000 [5], precisely for their ability to oversubscribe memory, i.e. use more memory than the memory-per-CPU share of the CPUs on which their jobs are running. This may force the admission control scheme to allocate processors with unit equal to the size of an SMP, or limit the amount of memory given to each job to a small fraction of the memory available on the SMP.

Migrating processes to unloaded nodes may provide a solution to memory pressure in COWs but it does not work within SMPs, as the shared memory is equally accessible regardless of the processor of execution. Migration of parallel jobs in clusters and distributed systems is difficult in theory and in practice. It incurs high overhead (typically on the order of minutes, even for programs with small problem sizes) and its effectiveness depends on the granularity, execution time, and communication patterns of the application, of which the system has limited or no knowledge [18].

In this paper, we propose a general framework for application-level dynamic memory adaptation in a certain class of applications with repetitive access patterns. Using this framework, an application can run very efficiently in-core when enough memory is available, and, when memory becomes scarce, can gracefully degrade its performance by shifting some of its work out-of-core in a controlled way. Available main memory is used as cache, while uncached pages are explicitly brought to and from the disk. Because the application has exact knowledge of the access pattern, optimal caching and prefetching policies can be used, vastly improving on what the virtual memory system can do. Throughout the paper we maintain the generality of our approach, but we draw our examples from scientific computing, where repetitive access of large amounts of data is typical.

In section 2, we summarize other related work in the context of the problem we study. In section 3, we provide a framework for minimally modifying applications to add memory adaptivity and we discuss some implementation details. In section 4, we develop a parameter-free algorithm that dynamically ascertains the amount of available memory at any point of execution using only runtime measurements of the resident set size. Using this algorithm, our supporting library enables a graceful degradation of performance as memory becomes scarce. In section 5, we modify a Conjugate Gradient linear system solver for memory adaptation and we provide timings that show the benefits of our method. We conclude with some discussion on the future directions of our approach.

2 Related work

Chang et.al. [4], have presented a user-level mechanism for constraining the resident set sizes of user programs within a fixed range of available memory. This mechanism assumes that the program knows a-priori the lower and upper limit of the band of available memory on which it can run, and customizes its resident set accordingly at startup. This work does not consider dynamic changes to the memory available to a program at runtime (either increases or decreases), nor does it address the problem of customizing the memory allocation and replacement policy to the memory access pattern of the application.

Application-specific algorithms for physical memory management [7], and caching, prefetching and disk scheduling [3] have been proposed to remedy the problems of generic operating system policies for memory management, such as approximations of LRU replacement. These algorithms have been proposed for stand-alone applications with specific access patterns, rather than for multiprogrammed systems. Furthermore, they generally assume a fixed amount of physical memory that is available to a program at runtime. Out-of-core methods for sequential and parallel numerical programs [6, 15, 19, 17] assume that the program runs in a fixed memory space which is not enough to cache the working set of a program throughout execution, and use restructuring optimizations to minimize I/O latency and improve disk utilization. These methods do not react to variations in the memory available to the program at runtime.

Brown and Mowry [2] developed an approach in which a compiler inserts hints where it believes that pages should be prefetched or released. A run-time layer follows the hints when it deems them appropriate to current system conditions. The approach has shown some good results, although applications with complex memory-access patterns can cause significant difficulties in identifying appropriate release points.

Barve and Vitter [1] presented a theoretical framework for estimating the optimal performance that algorithms could achieve if they adapted to varying amounts of available memory. However, they did not discuss how such adaptivity could be implemented. Pang et.al. [13] presented an adaptive version of a sorting algorithm, which dynamically splits and merges the size of the resident buffer to adapt to changes in the memory available to the sort by the database management system (DBMS). This study was conducted with a simulator and no implementation details of the adaptation interface between the algorithm and the DBMS were discussed.

In [12], one of the authors presented an adaptive scheduling scheme for alleviating memory pressure on multiprogrammed COWs, while co-ordinating the scheduling of the communicating threads of parallel jobs. That scheme required modifications to the operating system. In [11], the same author suggested the use of dynamic memory mapping for controlling the resident set size of a program, so that it stays within a band of available physical memory at any point of execution. The proposed mechanism was application-independent and used generic, but suboptimal algorithms for eviction of memory blocks. The algorithm operated at page-level granularity. However, better optimizations are possible with application-defined units of data transfer.

In [10], two of the authors followed an application-level approach for memory balancing. The idea was to avoid thrashing during the most computationally intensive phase of an iterative eigenvalue solver, the so-called correction phase. If the program detected memory pressure on a node, it receded its correction phase from that node, hopefully speeding the completion of competing jobs. A load balancing scheme guaranteed that other nodes would pick up the correction work of the receded process. Outside the correction phase the code executed with memory pressure but for a very short period of time.

3 A portable framework for memory adaptivity

Many scientific applications, such as sparse iterative methods, dense matrix methods, and Monte Carlo techniques, use blocked algorithms to exploit memory hierarchies. Applications with data sets that do not fit in the DRAM available in a workstation typically use out-of-core algorithms, which are also blocked to effectively use DRAM as a cache for disk data. For out-of-core methods, the blocks are often referred to as panels to distinguish from disk blocks. Blocked algorithms have a common processing pattern. Normally, data is partitioned into P panels and the algorithm operates on them in a loop as shown below:

```
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
```

On a dedicated workstation with a fixed amount of DRAM on board, one can easily select between an in-core or an out-of-core algorithm, according to the size of the problem that needs to be solved. On a non-dedicated system though, the choice between in-core and out-of-core algorithms is not obvious. Multiple applications may contend for physical memory. If the amount of DRAM available to a specific application fluctuates at runtime, the data set of the application may or may not fit in memory at different points of execution. It is desirable to use an adaptive algorithm, which dynamically adjusts the resident set size of the application based on memory availability.

In theory, virtual memory mechanisms can transparently adjust the resident sets of applications according to memory load. The operating system pages in non-memory-resident data on demand, and reclaims pages from programs when it detects memory shortage. Virtual memory is entirely transparent to the application, but has several shortcomings. The most important is that the page replacement algorithms used by the virtual memory system do not necessarily match the data access patterns and the memory demands of applications. As a consequence, the operating system often pages out data when they are actually needed by the application. In the worst case, poor replacement decisions have a cascading effect and lead to thrashing. Eventually, the system spends more time paging data than executing useful computations.

Adaptation to memory load can be achieved by switching dynamically from an in-core to an out-of-core version of the algorithm, whenever the application detects memory pressure. This solution is attractive from many points of view. Optimized out-of-core algorithms are readily available for many applications. In an out-of-core algorithm, controlling the size of the resident set can be done naturally by controlling the number and the size of panels kept in core. Out-of-core algorithms optimize the data transfers by taking advantage of the physical placement of panels on the disk and exploit filesystem optimizations such as prefetching and data aggregation to minimize latency and maximize disk throughput.

Nevertheless, several additional mechanisms need to be introduced in out-of-core algorithms to make them work in an adaptive manner. The out-of-core algorithm should run as fast as an in-core algorithm if the program has enough memory to cache its entire data set. Besides that, the algorithm needs a mechanism to detect if the operating system changes the amount of physical memory that the program can use at runtime. The algorithm must react to both memory shortage and memory availability. At the application level, this is a non-trivial task, since most operating systems reveal little information on available physical memory.

In this section we provide a framework for memory adaptivity which is portable to many block-structured applications and operating systems. In the following sections we provide algorithms for realizing this memory adaptivity and we confirm their optimal performance given a certain amount of memory.

A key element of our implementation is the management of panels with memory mapped I/O. This is a highly portable mechanism, available in all modern desktop and server operating systems. Memory-mapped I/O unifies computation and I/O and simplifies the code to a great extent. With it, we can derive an adaptive implementation of an algorithm which is identical to an in-core version, with one minimal extension to adjust the number of panels kept in-core, whenever a new panel is fetched. Additionally, using named mappings to files has some striking advantages under memory pressure: Clean pages from a named memory mapping can be evicted by the virtual memory system without being written to the swap device. Finally, via the `madvise()` call we can provide hints to the operating system about how a mapped region of memory will be used.

We control the number of panels that the application keeps in-core, whenever the algorithm attempts to bring in a new panel to work on. At this point, the algorithm has three choices: it can increase the number of in-core panels if additional memory is available; it can decrease the number of in-core panels if 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 and panels to bring in is application-specific. Given full knowledge of the data access pattern, the application can use the optimal policy for panel replacement. For instance, the test programs in this paper have repetitive, sequential access patterns, so MRU is the optimal policy, whereas the LRU approximations used in most virtual memory systems would completely fail.

The next section describes machine-independent algorithms to detect memory shortage or availability from within the applications, using solely local information. Coupled with memory mapped I/O, these algorithms inject adaptivity to memory shortage and availability with minimal implementation cost and maximum portability.

4 Adapting to memory availability

Having addressed how the library decides on the total number of panels and on their replacement policy, the main question is that of memory adaptivity. We would like to be able to reduce the number of panels when memory shortage is detected but still cache as many panels as possible. Moreover, when memory becomes available we should be able to utilize it promptly by mapping more panels.

Detecting memory shortage is relatively straightforward. During execution, a decrease in the program's resident set size (RSS), without any program unmapping action, is an indication of memory pressure. Detecting the level of pressure can be determined by the disparity between RSS and the amount of memory the program thinks it should have.

Detecting memory availability is more involved. Ideally, the system would provide an estimate of the amount of available memory, and the program would use this to determine the number of additional panels to map. Unlike RSS, however, this is global system information and most operating systems do not provide it accurately. The amount of free memory that is reported by many systems can be a huge underestimate of the amount of memory actually available. For instance, in many systems, the amount of free memory is usually close to zero, because any memory not associated with running processes is used by the file cache. In this case, the system might still service a large memory request from a program by reducing the size of the file cache. Thus, the most reliable way to determine if a quantity of memory is available is to use it and see if it can be maintained in RSS.

We emphasize that memory shortage and availability are concepts that are local to the program. For example, high system CPU utilization may still mean memory shortage if our program is swapped out, and memory availability may be the result of memory pressure on other processes.

4.1 Detecting memory shortage

Consider an application that is memory managed by our library. We denote by Panels_in the number of panels that are cached in memory. Because the application has knowledge of the rest of its memory requirements, it can compute what its current RSS should be. We call this desired RSS and denote as dRSS:

$$\text{dRSS} = (\text{Other Program Memory}) + \text{Panels\_in} * \text{Panel\_size}.$$

By definition, the application is under (additional) memory pressure when it cannot maintain this desired RSS. If the application detects a decrease in RSS, a number of cached panels should be unmapped so that the new desired RSS reflects the reduced RSS. However, the panels to be unmapped may not coincide with the memory paged out by the system (the cause of RSS reduction), so the following straightforward scheme

```
    if ( $\text{RSS} < \text{dRSS}$ ) then
```

```
diff = (dRSS-RSS) / Panel_size
unmap diff panels
Panels_in = Panels_in $-$ diff
dRSS = dRSS $-$ diff * Panel_size
```

may lead to a cascade of unmappings until Panels_in = 1. Consider an example where 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 replacing 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 but one panels are unmapped.

This cascade of unnecessary unmappings of cached panels reduces the performance of the code significantly. To avoid this problem, we introduce lastRSS, a variable that tracks the value that RSS had immediately before the access of the last panel. If memory pressure increased during that panel access, then additional panels may have to be unmapped in the following iteration. We initialize lastRSS to dRSS and we update it by executing the first five lines and the last line of the algorithm in figure (1) before accessing each new panel. We assume that within the execution of the if-statement no additional page out activity occurs, so that RSS can be reduced only by the unmap call. This is because the number of page faults that can occur in the system during the execution of three statements is limited and far smaller than the panel size. Finally, in practice, we can only unmap down to a minimum of one panel, so that the program can still perform work.

Assuming that eviction of pages with no action by the program is an indicator of memory shortage, a program experiences memory shortage if and only if our algorithm detects it. Using variable subscripts to denote the iteration number, observe that lastRSS$_i$ is the RSS at iteration $i-1$ after the end of the algorithm and before the panel access. If the condition of the algorithm holds, RSS$_i$ $<$ lastRSS$_i$, RSS decreased during the access of the last panel, so there must be memory shortage. Conversely, if there is memory shortage it will manifest itself by reducing RSS during the access of the last panel. Since lastRSS$_i$ records the last value of RSS at $i-1$ iteration, the condition will hold.

We note that operating systems employing a page-fault frequency (PFF) strategy for preventing thrashing will 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

```
Algorithm: Adapting to memory variability
RSS = Get current RSS
if ( RSS < lastRSS ) & ( Panels_in > 1 )
    diff = (lastRSS-RSS) / Panel_size
    unmap diff panels,  Panels_in −= diff,
    dRSS −= diff * Panel_size
else if ( dRSS == RSS ) & ( Panels_in < $P$ )
    peakRSS = max( peakRSS, RSS )
    delay = Time to access the last $P$ panels
    if (Time since last unmap >
        delay * min(10, peakRSS/(peakRSS−RSS)) )
        Panels_in ++
        dRSS += Panel_size
        peakRSS = RSS
    endif
endif
lastRSS = Get current RSS
```

Figure 1. The complete algorithm for adapting to memory variability

the process then increases its page fault rate, the OS will give frames back. Under a PFF strategy, our algorithm may interpret this 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 a PFF system.

4.2 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.

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.

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 2a 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.

We can reduce the aggressiveness of our policy by delaying growth of Panels_in for a time after Panels_in 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. This model is very simplistic, but 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 (that is, a 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 2a 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 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:

$$\text{delay} = (\text{base delay}) * \text{maxRSS}/(\text{maxRSS - RSS})$$

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 therefore is a natural value for the base delay.

The complete memory adaptation algorithm is shown in figure 1. Our code maintains a queue of timestamps for the the last P panel accesses to determine the base delay. Because we do not know maxRSS, we must approximate it somehow. We use peakRSS, which is the maximum RSS that has been achieved by the program since the last probe for more memory. Figure 2 demonstrates how the introduction of our delay parameter improves performance.

4.3 Graceful degradation of performance

Given the algorithm in figure (1), which closely tracks the available memory in the system, and a user defined replacement policy, our approach can achieve a nearly optimal caching scheme. The remaining question is whether the system can exploit this caching efficiently. Ideally, we would expect a linear increase in execution time as the available memory decreases. Figure (3) provides evidence that performance does degrade gracefully and adaptively. The left figure shows that a static version of our method benefits almost linearly from more cached panels up to the point of memory contention. On the other hand, a traditional out-of-core implementation is mainly insensitive to the panel size, favoring rather small panels. The right figure shows that our dynamic scheme achieves the same linear degradation of performance under increasing load but without any foreknowledge of available memory.

5 Conjugate Gradient experiments

To test our adaptive strategy and supporting library in the context of a scientific application, we implemented it in a conjugate gradient (CG) linear system solver using the CG routine provided in SPARSKIT [16]. The computational and storage requirements of CG are typical of many other scientific algorithms. Each iteration requires a sparse matrix-vector multiplication and a few dot products. The program requires storage for only four vectors while the

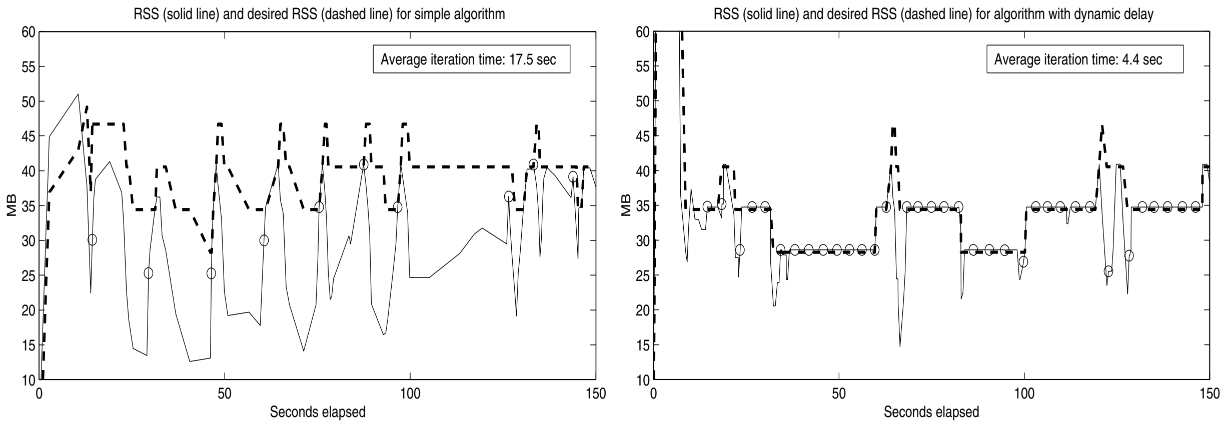


Figure 2. RSS (solid line) and desired RSS (dashed line) versus time for two versions of a memory-adaptive program that performs dense matrix-vector multiplications with a 70 MB matrix broken into 10 panels. It runs against a 70 MB dummy job on the Linux machines described in section 5. Circles denote the beginning of a matrix-vector multiplication. The left graph (a) uses the original 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 right 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.

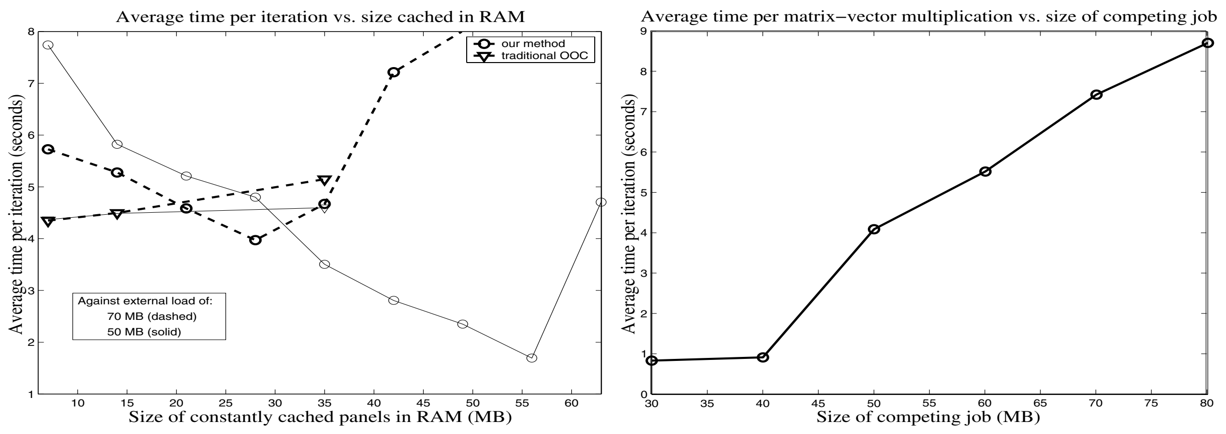


Figure 3. Graceful degradation of performance. The left graph shows the execution times (designated by circles) for a static version of our method that caches a constant number of panels. Also, it shows the times (designated by triangles) for a traditional out-of-core (OOC) implementation for various sizes of its single panel. We report times running with a 70 MB matrix against two cases: a 50 MB (solid lines) and a 70 MB (dashed lines) external load running on the Linux machines described in section 5. Increasing the number of panels cached improves performance almost linearly as long as the amount of available available memory is not exceeded; times increase towards the right of the graph as the amount of panels cached exceeds available memory. In the right graph, a similar graceful degradation of performance is observed for our dynamic method under external loads of increasing size. However, the number of panels cached is chosen dynamically.

bulk of the memory demand comes from the coefficient matrix. In our experiments, the driver program breaks the matrix into $P = 10$ panels and stores them on disk. Work vectors are kept in-core. We note that the CG routine is not modified, and we use an off-the-shelf blocked sparse matrix vector multiplication with only a single call to our `get_next_panel()` function. We use both sequential and parallel versions of the solver. In the parallel version, whenever a matrix-vector or inner product is called for, a collective communication operation must occur.

We conducted a series of experiments on four identically configured 1 GHz Pentium III machines with 128MB of DRAM. All machines run Linux 2.4.18-19.7 and are connected to the same fast Ethernet switch. We use `gcc` and `g77` compilers and the LAM MPI communication library for the parallel version.

5.1 Sequential experiments

Some results have already been described in section 4. Here, we compare four different ways to implement a CG code. Besides a standard in-core implementation and our memory adaptive one, we also use a conventional out-of-core code, as well as an in-core code that stores the matrix on disk and accesses it via a read-only memory map to avoid inefficient write-outs to the swap device. Table 1 summarizes what happens when any possible combination of two sequential CG solvers are run against each other. Each solver runs on a 70MB matrix with a total of 81MB storage requirements; this causes considerable memory pressure, as only about 105 MB total are available to the programs. The performance of the in-core code under memory pressure is very poor, as expected. The memory-mapped code performs well if it is started first, but it is starved when jobs other than out-of-core are already running. The out-of-core code performs consistently against all other codes, but its lack of adaptivity does not justify general purpose use. The memory adaptive code works well in all cases, even when run against itself, demonstrating its truly dynamic nature.

In table 2 we present results like those from table 1 but obtained under Solaris 8. Detailed discussion of the performance of our adaptation scheme under Solaris is beyond the scope of this paper, but we wish to show that the scheme works under both Linux and Solaris, though the systems use different memory-management strategies.

5.2 Parallel experiments

In our parallel CG experiments on four processors, we used the in-core and memory-adaptive versions of the solver to solve a problem with a 280 MB coefficient matrix arising from an eighth-order finite-difference discretization of a three dimensional Laplacian problem. To create mem-

Time for method X running against Y

X \ Y	incore	ooc	mmap	mema
incore	204.00	0.82	0.66 20.50	27.00 22.50
ooc	5.00	8.82 9.60	4.90	5.10
mmap	0.70 35.00	0.84	0.67 35.00	0.79 35.00
mema	0.76	0.90	4.50 0.72	0.89 5.34

Table 1. Average time per iteration for method X when running against method Y. “incore” denotes a standard in-core algorithm, “ooc” a conventional out-of-core one, “mmap” an in-core algorithm that uses memory-mapped I/O to read the matrix, and “mema” our memory adaptive code. Both jobs execute CG on a 70 MB matrix, reading it from different files where applicable. The time is measured after both methods have stabilized sharing the CPU. One of the jobs is started 9 seconds after the other. If one time is reported, it is independent of starting order. If two times are reported, the top is the time for method X when X is started first, while the bottom is the time when X is started second.

Time for method X running against Y

X \ Y	incore	ooc	mmap	mema
incore	493.0	11.74	239.8	203.4
ooc	20.25	92.10	19.76	19.87
mmap	7.59	12.18	99.72	66.34
mema	5.32	9.89	33.33 42.30	38.66

Table 2. A table like table 1, but showing experimental results obtained on a SunBlade 100 workstation with 384MB running Solaris 8. Both jobs execute conjugate gradient on a 192 MB matrix. Note that the in-core code is “starved” by the OS when running against the memory-adaptive or memory-mapped I/O codes.

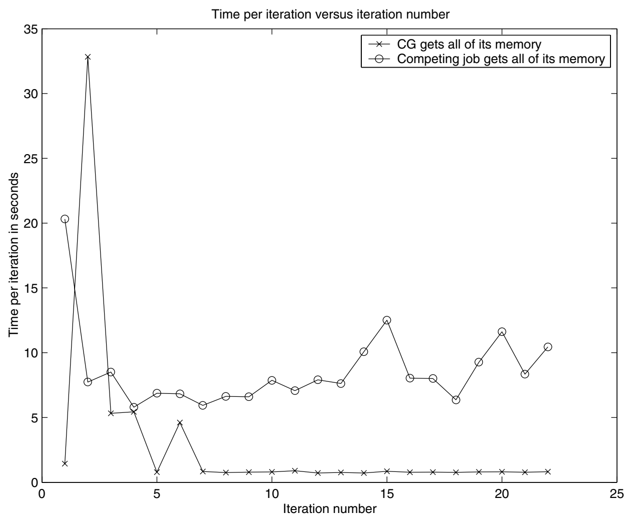


Figure 4. Execution times for each iteration for the first 25 iterations of the parallel CG code running against a 70MB external job. The two sets of data correspond to the two possibilities shown in figure 5. Even the slower set of times are far lower than those obtained by relying on the virtual memory system.

ory pressure, the root node executes a memory-intensive 70 MB dummy job. We have used a job that allocates memory and continuously writes random numbers to it as well as a sequential in-core CG code. Results were identical using either dummy job, and are consistent with those observed in the sequential tests. Without competition, the in-core code averages 0.72 seconds per iteration, and the memory-adaptive code 0.73 seconds. When running against the dummy job, the in-core code performs very poorly, taking anywhere from 32 to 80 seconds per iteration. Under constant memory pressure, the memory adaptive code averages between 8 and 9 seconds, consistent with the slowdown experienced by the sequential adaptive code under the same memory pressure. However, the slowdown in the parallel case affects all nodes.

In some cases (observed in the sequential code as well), the memory adaptive code would thrash (system mode CPU utilization $\geq 95\%$) for 40–50 seconds, after which it would obtain enough memory resources to keep its entire local portion of the matrix. Figure 4 illustrates for both cases how the time per iteration changes as the solver adapts to memory pressure, while figure 5 shows the actual memory adaptation. Even in the slower case, the timings are far lower than those obtained by relying on the virtual memory system to handle memory pressure.

6 Conclusions

We have presented a framework for dynamic adaptation to memory pressure in scientific applications. This framework enables an application running on a non-dedicated workstation to gracefully degrade its performance when it cannot obtain the resources required to fit its data set in main memory. It is particularly suited for non-centrally administered, open systems, such as clusters of privately owned desktops, where loads can fluctuate unpredictably.

We have made the following key contributions: We presented a novel, system-independent algorithm that ascertains the availability of main memory using a single metric, i.e. the resident set size of the application. In addition, we presented an algorithm that enables an application to dynamically and optimally adjust its resident set size in response to memory shortage or availability. The algorithms are portable to almost any modern operating system and hardware platform.

In addition to easy portability, our framework has a modular design. Its use requires minimal extensions to block-structured application kernels. Because it can be embedded in a computational kernel, it can be immediately deployed in any application that uses that kernel. For example, embedding our framework in low-level libraries such as BLAS or SPARSKIT makes it immediately available to higher level libraries, such as LAPACK and scaLAPACK, that depend upon them. In turn, applications that rely on these higher level libraries can immediately benefit from the framework.

References

[1] R. D. Barve and J. S. Vitter. A theoretical framework for memory-adaptive algorithms. In *Proc. of the 40th Symposium on Foundations of Computer Science*, pages 273–284. IEEE Press, 1999.

[2] A. D. Brown and T. C. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 31–44, 2000.

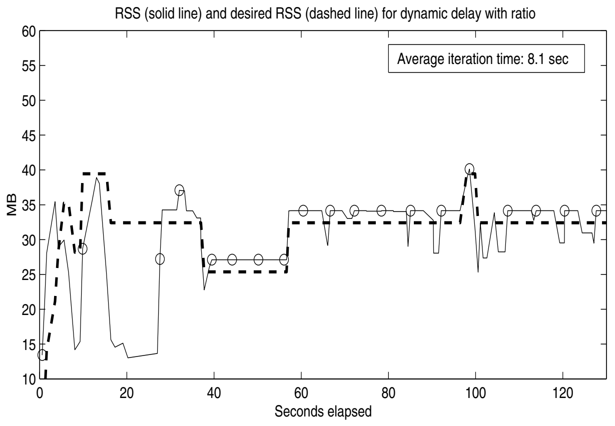
[3] P. Cao, E. Felten, A. Karlin, and K. Li. Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, Nov. 1996.

[4] 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, Aug. 2000.

[5] S. Chiang and M. Vernon. Characteristics of a Large Shared Memory Production Workload. In *Proc. of the 7th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'2001), Lecture Notes in Computer Science, Vol. 2221*, pages 159–187, Cambridge, MA, June 2001.

IEEE COMPUTER SOCIETY

0-7695-2132-0/04/$17.00 (C) 2004 IEEE



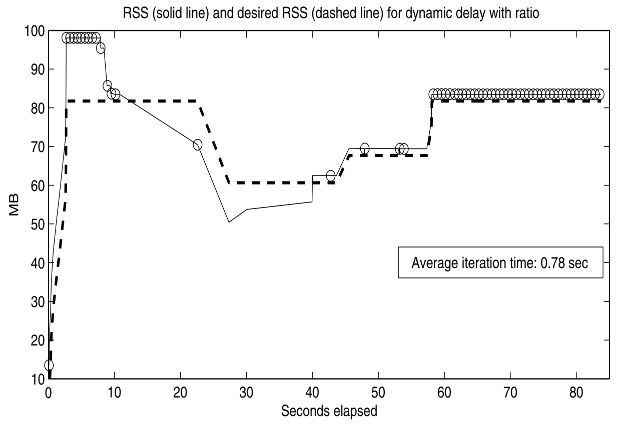


Figure 5. A memory profile similar to Figure (2) but for the root node of a parallel job. The external load is 70 MB. We observe two possibilities. In the left graph, the external load keeps its entire working set while our method utilizes the remaining memory. The scenario reverses in the right figure. In both cases, resource utilization is high for all participating nodes.

[6] J. Dongarra, S. Hammarling, and D. Walker. Key concepts for parallel out-of-core LU factorization. *Parallel Computing*, 23(1–2):49–70, Apr. 1997.

[7] K. Harty and D. Cheriton. Application-controlled Physical Memory Using External Page-Cache Management. In *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'V)*, pages 187–197, Boston, Massachusetts, Oct. 1993.

[8] R. Henderson. Job Scheduling Under the Portable Batch System. In *Proc. of the First Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science Vol. 949*, pages 279–294, Santa Barbara, CA, Apr. 1995.

[9] 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, Nov. 1997.

[10] 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 *Proc. of the 15th ACM International Conference on Supercomputing*, pages 454–463. ACM Press, 2001.

[11] 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, Apr. 2003.

[12] 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.

[13] H. Pang, M. J. Carey, and M. Livny. Memory-adaptive external sorting. In *Proc. of the 19th International Conference on Very Large Data Bases*, pages 618–629, Dublin, Ireland, Aug. 1993. Morgan Kaufmann.

[14] R. Daugherty and D. Ferber. Network Queuing Environment. In *Proc. of the Spring Cray Users Group Conference (CUG'94)*, pages 203–205, San Diego, CA, Mar. 1994.

[15] E. Rothberg and R. Schreiber. Efficient Methods for Out-of-Core Sparse Cholesky Factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, Jan. 2000.

[16] Y. 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/>.

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

[18] S. Vadhiyar and J. Dongarra. A Performance Oriented Migration Framework for the Grid. In *Proc. of the 3rd International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, May 2003.

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

IEEE COMPUTER SOCIETY