A Memory Management System Optimized for BDMPI's Memory and Execution Model

Jeremy Iverson
University of Minnesota
Minneapolis, MN 55455, USA
jiverson@cs.umn.edu

George Karypis
University of Minnesota
Minneapolis, MN 55455, USA
karypis@cs.umn.edu

ABSTRACT

There is a growing need to perform large computations on small systems, as access to large systems is not widely available and cannot keep up with the scaling of data. BDMPI was recently introduced as a way of achieving this for applications written in MPI. BDMPI allows the efficient execution of standard MPI programs on systems whose aggregate amount of memory is smaller than that required by the computations and significantly outperforms other approaches. In this paper we present a virtual memory subsystem which we implemented as part of the BDMPI runtime. Our new virtual memory subsystem, which we call SBMA, bypasses the operating system virtual memory manager to take advantage of BDMPI's node-level cooperative multi-taking. Benchmarking using a synthetic application shows that for the use cases relevant to BDMPI, the overhead incurred by the SBMA system is amortized such that it performs as fast as explicit data movement by the application developer. Furthermore, we tested SBMA with three different classes of applications and our results show that with no modification to the original MPI program, speedups from 2×–12× over a standard BDMPI implementation can be achieved for the included applications.

Keywords

distributed computing, big data, virtual memory, out-of-core, mpi

1. INTRODUCTION

The continuous growth in the amount of data being collected and stored has created a need to perform large computations on small systems, as access to large systems is not widely available and cannot keep up with the scaling of data. Running large problems on smalls systems makes it imperative to have an efficient means to manage the increased memory pressure, a problem which is solved using a technique referred to as out-of-core computation. Many frameworks for out-of-core distributed computing applications (i.e., distributed computing applications that primarily store their data on the disk) have been developed to address this problem [2, 4–7, 10–12, 14, 16, 20–22]. Recently a new framework has been developed called BDMPI [9], which is designed to enable MPI applications to automatically execute in an out-of-core fashion and thus solve very large problems on moderate computational resources.

BDMPI uses co-operative multi-tasking to reduce contention for memory resources and defers the actual management of the exchange of data to the operating system (OS) virtual memory manager (VMM). In [9], the authors of BDMPI showed that even when relying on the OS VMM, BDMPI outperforms other out-of-core distributed frameworks for a variety of applications. The authors also showed that BDMPI can be improved if the application developer explicitly controls the exchange of data between memory and disk before and after blocking MPI operations.

The objective of this work is to enhance BDMPI's runtime with a virtual memory (VM) subsystem that is aware of BDMPI's execution and memory model. The key insight to this approach is that the memory access patterns of most applications can be broadly classified into a small number of categories. Coupled with the memory restrictions for concurrently executing processes imposed by the BDMPI execution and memory models, a VMM can optimize the way it transfers data between disk and physical memory.

Towards this objective, we developed a new VM subsystem for the BDMPI runtime, called *Storage-Backed Memory Allocation* (SBMA), for its use of file backed memory mappings. SBMA can be used with any MPI application, but includes optimizations for specific use cases, such as applications which only access subsets of their memory between communication induced blocking points. Additionally, SBMA is completely transparent to the application developer as its use requires no code modification.

We experimentally evaluated the performance of SBMA on a synthetic micro-benchmark and three real applications: (i) PageRank [13], which uses an iterative algorithm to copmute the pagerank of the vertices in a graph, (ii) ParMetis [15], which computes a k-way partitioning of a graph, and (iii) SPLATT [18], which computes a PARCOMP sparse tensor factorization. Our results show that with no modification of the original MPI program, speedups of 2–12× over a standard BDMPI implementation can be achieved for the included applications.

2. OVERVIEW OF BDMPI

BDMPI is implemented as a layer between an MPI program and any of the existing implementations of MPI. From the application's perspective, BDMPI is just another implementation of a subset of the MPI 3 specification. Programmers familiar with MPI can use it right away and any programs using the subset of MPI functions that have been currently implemented in BDMPI can be linked against it unmodified.

The execution of a BDMPI program creates two sets of processes. The first is the MPI processes associated with the program being executed, which within BDMPI, are referred to as the *slave* processes. The second is a set of processes, one on each compute node, that are referred to as the *master* processes. The master processes are at the heart of BDMPI's execution as they spawn the slaves, coordinate their execution, service communication requests, perform synchronization, and manage communicators.

BDMPI's execution model is based on *node-level cooperative multi-tasking*. BDMPI allows only a subset of the slave processes to be executing concurrently with the rest of the slaves blocking. When a slave process reaches an MPI blocking operation (e.g., point-to-point communication, collective operation, barrier, etc.), BDMPI blocks it and selects a previously blocked and runnable process (i.e., whose blocking condition has been satisfied) to resume execution.

BDMPI's memory model is based on *constrained memory over-subscription*. It allows the aggregate amount of memory required by all the slave processes spawned on a node to be greater than the amount of physical memory on that node. However, it requires that the amount of memory used by any running slave processes be smaller than the amount of physical memory on that node. Within this model, an unmodified MPI program will rely on the OS VMM mechanisms to map in memory the data that each process needs during its execution.

The coupling of constrained memory over-subscription with node-level cooperative multi-tasking is the key that allows BDMPI to efficiently execute an unmodified MPI program whose aggregate memory requirements far exceeds the aggregate amount of physical memory in the system. This is due to the following two reasons. First, it allows the MPI processes to amortize the cost of loading their data from the disk over the longest possible uninterrupted execution that they can perform until they need to block due to MPI's semantics. Second, it prevents memory thrashing (i.e., repeated and frequent page faults), because each node has a sufficient amount of physical memory to accommodate all the processes that are allowed to run. For a more detailed description of BDMPI, please refer to [9].

3. STORAGE BACKED MEMORY ALLOCATION (SBMA)

Even though BDMPI's execution model is designed to maximize the amount of work that can be done with the data that was fetched from disk, the experiments in [9] showed that further performance improvements can be obtained by explicitly controlling how resident memory is transferred to and from disk. These optimizations require an in-depth understanding of the execution and memory models of the BDMPI runtime. Thus, it would be desirable to have a runtime system that can leverage the cooperative multi-tasking execution model of BDMPI to perform many of these optimizations automatically.

Motivated by the observation that optimized resident memory exchange can be implemented without any burden to application developers, we created a new virtual memory (VM) subsystem for the BDMPI runtime which incorporates knowledge of the system's execution and memory models. We call the new VM subsystem *Storage-Backed Memory Allocation* (SBMA), for its use of file backed memory mappings to persist data between exchanges of resident memory.

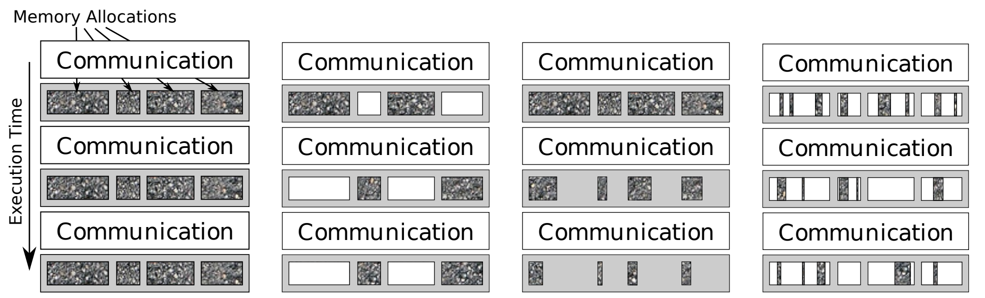
To provide these functionalities, SBMA operates in the following way. Each time that a slave process allocates memory, the request is handled by SBMA, which obtains the memory from the OS on behalf of the slave process. Internally, SBMA keeps track of which parts of the allocation are currently resident in physical memory and which have been modified by the application. This way, just before the physical memory becomes over-subscribed, SBMA exploits the cooperative multi-tasking nature of the BDMPI runtime and evicts the memory of blocked slave processes at an allocation granularity. Like a traditional VMM, SBMA uses a file on disk to persist the contents of evicted memory. In addition, by tracking the parts of each allocation that have been modified by the application, only those parts which have been modified since last being written to disk need to be written upon eviction. Likewise, when an allocation is re-admitted into physical memory, only those parts which exist in the file on disk need to be read. To ensure that memory mappings remain valid from acquisition to release, despite being evicted and re-admitted into physical memory, SBMA relies on a core set of memory-related functionalities provided by the Linux OS, discussed in detail in Section 4.

To reduce unnecessary data transfer between physical memory and disk, SBMA exploits the semantics of certain MPI functions. When a slave process blocks on a MPI operation that requires it to transfer data (e.g., point-to-point communication or collective operations), it is within the semantics of such an operation to *discard* the contents of any output buffers before receiving data. This means that the BDMPI runtime may forgo writing to file any memory regions marked as modified which are present in the output buffers of such MPI routines.

Within this framework, we developed three different strategies for managing the exchange of resident memory, the details of which are provided in the rest of this section.

3.1 Aggressive Read / Aggressive Write (ARAW)

This strategy is designed for applications where the slave processes have a core set of memory allocations which are accessed in their entirety during each execution phase, see Figure 1a. In this approach, when a process enters a blocking state, it writes any modified parts of its allocations to the appropriate file on disk and then releases the associated



(a) All allocations are accessed in entirety.
(b) A subset of allocations are accessed in entirety.
(c) Some allocations change size in successive phases.
(d) Allocations are partly accessed.

Figure 1: Four common memory access patterns. Each subfigure shows a selection of a single slave process' execution phases (communication followed by computation). Within a computation phase, memory allocations are represented by separate blocks and accesses to an allocation are indicated by a filled block or section.

physical memory resources, a procedure which we call *unloading*. Upon exit of the blocking state, the first access to each allocation causes physical memory to be acquired for it and any parts of the allocation which exist in its associated file to be read, a procedure that we refer to as *loading*. Note that only those allocations that a slave process accesses during an execution phase are actually loaded, see Figure 1b. Furthermore, ARAW requires no communication between the master process and the slave processes in order to coordinate the exchange of resident memory.

3.2 Aggressive Read / Lazy Write (ARLW)

A drawback of the ARAW approach is that it assumes that in each execution phase, the physical memory will be oversubscribed. However, there are many applications for which this will not be the case. In general, this is applications in which the aggregate set of accessed memory of the slave processes between synchronization points can fit in memory during successive execution phases. Two such examples are (i) applications which access different memory allocations during successive execution phases (Figure 1b) and (ii) applications which use data structures that change size as execution progresses (Figure 1c). The figures assume that the size of physical memory is roughly equal to the aggregate amount of memory allocations from the first execution phase shown. You can see then that in either figure, memory from other slaves, in addition to the one shown, will fit into the physical memory.

Motivated by this, in the Aggressive Read / Lazy Write (ARLW) approach, memory is allowed to remain resident until memory pressure becomes sufficiently high. By unloading allocations only when required, interaction with disk during these types of execution phases can be significantly reduced and in some cases, completely avoided. To accomplish this, the master process must coordinate the loading and unloading of slave memory. Thus, whenever a slave process loads memory, it first notifies the master process. The master checks to make sure that the pending load can complete *safely*, meaning that it will not over-subscribe the physical memory. If it cannot, the master directs blocked slaves to unload memory until the pending load will complete safely or no blocked slaves have any loaded allocations.

3.3 Lazy Read / Lazy Write (LRLW)

A drawback of both ARAW and ARLW is that upon the first access after being unloaded, all of the previously written parts of the allocation are read from disk. For applications which have execution phases that require access to only parts of each allocation, this behavior is not ideal. Consider as an example, an application which has a brief exchange of point-to-point communications which update only a small part of some number of allocations, an access pattern which might look like that shown in Figure 1d. Under either ARAW or ARLW, each update will cause an entire allocation to be loaded, even if only a small fraction of the allocation is being updated. Instead, only those parts of each allocation actually accessed should be considered for reading.

The lazy read / lazy write (LRLW) strategy addresses this limitation by unloading allocations using the same procedure as ARLW, but loading them at a chunk granularity rather than whole allocations. Here chunk is implementation defined, the details of which are discussed in Section 4. An ancillary advantage of a lazy read approach is that it facilitates asynchronous data transfer from disk. In aggressive reading, loading a memory allocation requires the slave process to block until all relevant parts of the allocation have been read from disk. In contrast, lazy reading performs loads at smaller granularity. This means after loading a chunk, a fast operation compared with loading an allocation, the slave process can resume execution. Meanwhile, instructed by readahead policies, the OS will continue to fetch data from the relevant file in the background. When an allocation is accessed in roughly sequential order, this translates to reduced time spent blocked on data transfer, since most of the data will be prefetched by the OS before it is accessed by the application.

4. SBMA IMPLEMENTATION

4.1 Memory Acquisition and Representation

SBMA uses interposition to intercept calls to libc's standard `malloc` library. That is, a call to `malloc()` in a BDMPI program will be performed by SBMA. Our implementation of SBMA includes a parameter called the *SBMA threshold* which controls the minimum sized allocation to be managed by SBMA. Any allocation less than the SBMA threshold

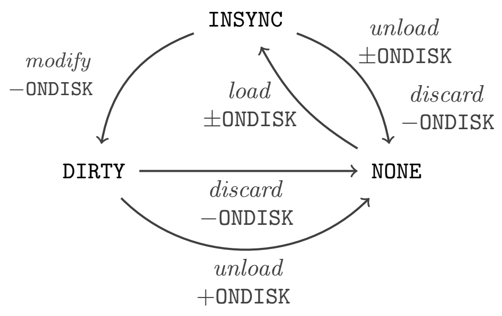


Figure 2: The state transition graph for `sbpages`. +`ONDISK` indicates that the `ONDISK` flag will be given to the `sbpage`, −`ONDISK` that it will be removed, and ±`ONDISK` it will continue to be present or absent, whichever it currently is.

will be passed directly to the appropriate libc function, bypassing SBMA management. SBMA uses the `mmap()` and `munmap()` functions to acquire/release memory to/from the OS. When an application requests memory, SBMA is invoked and calls `mmap()` to obtain a virtual address from the OS pointing to a region of memory that contains the requested amount of memory. Then, a file is created on disk where the memory region will be persisted whenever it is unloaded. The memory region returned by `mmap()` is segmented conceptually into consecutive equal sized blocks, each containing a fixed number of system memory pages, referred to as `sbpages`. Each `sbpage` is associated with an `sbpage` state. Valid `sbpage` states are unmodified (`INSYNC`), modified (`DIRTY`), and not accessible (`NONE`). Furthermore, the `INSYNC` and `NONE` states can be modified with a stored on-disk (`ONDISK`) flag. The precise meanings of these states are described in subsequent sections and Figure 2 provides a summary of the valid state transitions and their effect on the `ONDISK` flag of an `sbpage`. SBMA also provides the functionality to discard an allocation. This operation removes any `ONDISK` flags from the associated `sbpages` and transitions all `sbpages` to the `NONE` state. During this, no `sbpages` which are in the `DIRTY` state are written to the associated file, the allocation is simply treated as if it were newly allocated.

4.2 Access Control and Data Persistence

In order for SBMA to perform the various read/write optimizations, it needs to know which parts of the allocations are being accessed by the application and if these accesses are read and/or write. To achieve that, SBMA uses the functionalities provided by memory protection via the Linux kernel. Each of the three states, `INSYNC`, `DIRTY`, and `NONE`, imply a specific set of access permissions, set using the system call `mprotect()`. Based on these access permissions the OS will generate the signal `SIGSEGV` whenever the application attempts an access that is not permitted according to the state of the corresponding `sbpage`. Trapping these `SIGSEGV` signals is one of the means by which the SBMA system transitions `sbpages` from one state to another; the other being an explicit discard of an allocation. To trap the `SIGSEGV` signal, SBMA installs its own `SIGSEGV` handler at the time that `MPI_Init` is called.

When an `sbpage` is in the `INSYNC` state, it means that its corresponding file is synchronized with its contents and implies that the application has read access to the memory corresponding to the `sbpage`. An `sbpage` in the `DIRTY` state indicates that the application has modified the `sbpage` since the last time that its file was loaded and implies read and write access permissions for the application. Both the `INSYNC` and `DIRTY` signify that the `sbpage` is resident in physical memory. On the other hand, the `NONE` state means that the `sbpage` is not resident in physical memory and as such, implies that there are no access permissions for the `sbpage`.

Initially every `sbpage` of a new allocation is put in the `INSYNC` state. When the application attempts a write access to any of these `sbpages`, a `SIGSEGV` is generated by the OS and trapped by SBMA, which transitions the `sbpage` to the `DIRTY` state and changes the access permissions to read/write. When a slave process unloads an allocation, it scans the states of its `sbpages` and writes to the allocation's associated file only those `sbpages` which are in the `DIRTY` state. All `sbpages` in the allocation are then transitioned to the `NONE` state and any `sbpages` written to disk are given the `ONDISK` flag. Once an `sbpage` is given the `ONDISK` flag, it retains the flag until it is either freed or discarded. As part of unloading an allocation, the slave releases any associated physical memory resources, via the system call `madvise()`, invoked with the flag `MADV_DONTNEED`, which notifies the operating system that the specified memory resources are no longer needed. In addition, SBMA sets the permissions of the allocation to `NONE`, so any future access will be intercepted by SBMA and handled appropriately.

4.3 Aggressive Read / Aggressive Write Details

In the ARAW approach, when a slave process reaches an MPI synchronization point, but before entering the blocked state, it unloads all of its memory allocations. This means that for each of its allocations, it scans for `sbpages` in the `DIRTY` state, writes them to their appropriate location in the allocation's corresponding file and updates their state to include the `ONDISK` flag. Afterwards, all of the allocation's `sbpages` are transitioned to the `NONE` state and the access permissions for the allocation memory are set to none. Lastly, the physical memory resources are released by the process with a call to `madvise()`.

In an aggressive read approach, when a slave process receives a `SIGSEGV`, the allocation to which the offending memory location belongs is identified. Once found, the slave initiates an allocation load by reading into memory the `sbpages` that have the `ONDISK` flag, transitioning all of the `sbpages` to the `INSYNC` state and setting the access permissions to read.

4.4 Lazy Read Details

In a lazy read approach, when a slave process receives a `SIGSEGV` associated with a read access, the allocation to which the offending memory location belongs is identified. Once found, the slave initiates an `sbpage` load on the appropriate `sbpage` by reading it into memory only if it has the `ONDISK` flag, transitioning it to the `INSYNC` state and setting its access permissions to read.

4.5 Lazy Write Memory Tracking Protocol

The lazy write approach requires that the master process maintains an accurate count of the amount occupied physical memory, so that writes can be delayed. To accomplish this, messages are exchanged between the master process and relevant slave processes each time memory is loaded or unloaded. Four different message types are relied on to provide this functionality: load, unload, release, and proceed.

When a slave process wants to load memory, it sends a *load message* to the master containing the size of the memory region to be loaded and then waits to receive a *proceed message* in response. Upon receiving a load message, the master process verifies that the requested amount of memory can be safely loaded. If not, then the master process repeatedly chooses a blocked slave process and commands it to unload all of its memory allocations via a *release message*. This is repeated until no slaves have any loaded allocations or the requested amount of memory can be loaded safely, at which point the master sends a proceed message to the slave who initiated the request. Whenever a slave successfully unloads its memory allocations due to the reception of a release message, it notifies the master with an *unload message* containing the aggregate amount of memory unloaded. It is not sufficient to exchange load messages and unload messages only when memory is loaded and unloaded. It is also necessary to send unload / load messages whenever memory is freed / allocated, since these have the same effect on the total occupied memory as unloading / loading.

When lazy writing is coupled with lazy reading, an extra consideration must be made. Rather than sending a load message each time that an `sbpage` is loaded, which would be prohibitively expensive for applications with large allocations, the LRLW approach sends exactly one load message for each allocation. The message is sent when the first access is made to any of the allocation's `sbpages`.

4.6 Multi-threaded Environements

The previous discussion addresses the needs of many distributed out-of-core applications which rely solely on MPI. However, there are do exist applications which embrace the MPI+X paradigm. This is especially true of MPI+openmp and MPI+pthreads. To thoroughly address the issues of SBMA in a multi-threaded environment is outside the scope of this work. However, the major challenge involved in doing so, is the race condition that occurs when one thread is loading an `sbpage` at the same time that another thread is writing to an `sbpage`. This problem, known as the atomic page update problem, has been studied in other contexts, most notably with relation to distributed shared memory systems [19]. Some of the techniques described in [19] can be directly implemented within the context of SBMA and thus allow it to work for applications which rely on multithreading within a computation node. The performance implications of such a solution with respect to SBMA is left as future research.

5. RELATED WORK

There are a variety of research efforts in the literature related to efficient memory management strategies for data intensive out-of-core applications. Most of the previous work can be broadly categorized as follows: (1) OS memory management policies, (2) problem specific out-of-core solutions, or (3) efficient general purpose virtual memory managers.

Many page replacement policy studies have been performed under various conditions and for various workloads. Some examples are Lee et al. [10], Park et al. [14], and Qureshi et al. [16]. In each of their respective papers, the authors demonstrated different circumstances that application performance can be improved using page replacement policies designed for the characteristics of the application. Other works, like that of Engler et al. [5], focus on the design and implementation of a general purpose virtual memory subsystem at the application level. The authors argue the efficacy of an application level VMM and included microbenchmarks to show its feasibility. SBMA falls into this broad category of application level VMM.

More recently, there has been an emergence of interest in memory management systems catered toward out-of-core applications. A large amount of this work has been focused on problem specific out-of-core solutions. Typically, this means that a memory management strategy is tailored for the specific characteristics of the problem. While this level of customization generally yields the best results, the engineering cost is prohibitively high for most applications. Thus, this type of research has been focused mainly on those problems where performance is critical. For example linear algebra kernels [20] and data management systems [4].

There has also been a number of projects related to optimized general purpose virtual memory management systems. Most relevant to SBMA are three projects. The first is by Brown et al. [2], who looked at an entirely compiler-based solution for prefetching and releasing memory in out-of-core applications. The compiler used analysis of the source code to identify potentials points in the execution where data could be safely loaded and unloaded so as to reduce memory access latency. The complete solution also included a runtime layer which intercepted the compilers prefectch and release instructions and decided if they were still appropriate to execute based on the current state of the system at the time of interception. Using this approach the authors showed that for many large scientific applications, I/O stall time could be reduced by more than 50%. The second is the DI-MMAP system of Van Essen et al. [6, 21], a kernel module which replaces the existing Linux memory map runtime with a custom runtime. The authors showed that for the metagenomics application used in their paper the DI-MMAP system achieved a 4.88× performance improvement over the Linux kernel memory map runtime. Like SBMA, DI-MMAP is also built as a drop in replacement for memory acquisition, however DI-MMAP is built as a kernel module and is seen as an extension to the kernel rather than an application level VMM. In contrast to the Linux kernel VMM, which is optimized for shared libraries, DI-MMAP is optimized specifically for data-intensive out-of-core applications, including optimizations like bulk flushing of the page table and transition-look-aside buffers. Compared with SBMA, the DI-MMAP system is much lower-level and thus, has no notion of the cooperative multi-tasking nature of BDMPI. Lastly is the work of Meswani et al. [12], which addresses this problem for future Exascale machines. The authors argue that a programmer-driven approach to memory management is the quickest way to reach the bandwidth targets of the Exascale machines. Their results show that while programmer-driven methods are an improvement over automated systems, their is still more research to be done before the bandwidth targets of the Exascale machines are reached.

6. EXPERIMENTAL SETUP

6.1 Benchmark Applications

We evaluated the performance of SBMA using a synthetic application and three real-world applications: (i) PageRank on an unweighted undirected graph [13], (ii) graph partioning using ParMetis on an unweighted undirected graph [8], and (iv) sparse tensor factorization using SPLATT on an three dimensional sparse tensor [18].

6.1.1 *Synthetic Application*

The synthetic application was designed to quantify the overhead associated with a memory management subsystem built on top of the Linux kernel signaling mechanisms. The application allocates a single chunk of memory large enough to fill the RAM of the host machine, which in our case was slightly less than 4GB to avoid interference by the OS VMM. The synthetic application is comprised of three micro-benchmarks, each executing a single type of memory access operation from the following: read (`==`), write (`=`), or read/write (`+=`). Each micro-benchmark consists of a `for` loop which iterates over the entire allocation performing the appropriate memory access operation on each byte. To increase the stability of the timings, before each micro-benchmark is executed, the system's page cache is cleared via a call to `posix_fadvise()` and the CPU caches are cleared using unoptimized code to populate a sufficiently large segment of memory which is immediately released.

Each micro-benchmark was executed using the four possible combinations of the following memory placement policies: in-memory (I) and read (R) and memory protection policies: aggressive (A) and lazy (L). The I memory placement policy means that during the entire execution of the micro-benchmark the allocation is stored in memory, this is in contrast to the R policy where memory is stored on disk and must be read before being accessed. For a write (=) access, reading from disk is unnecessary and thus, the data is not loaded before writing. The A memory protection policy dictates that the first time that a read protection fault is raised for the allocation, the memory protection for the entire allocation is updated to read protected. The opposite of this is the L policy where each read protection fault raised causes only the corresponding `sbpage`'s memory protection to be updated to read protected. Note that this is not true for write faults, since SBMA always grants write permission exactly one `sbpage` at time.

The AI and LI combinations allow for the quantification of the inherent overhead of a memory subsystem based on the Linux kernel signaling mechanisms. The AR and LR policy combinations correspond exactly to the SBMA schemes including each of these policies. Thus, they allow us to measure the overhead which can be expected from the SBMA system. A special case, denoted as *no protection (np)* was included, in which the allocation is given read and write permission at the time of request and is not changed during execution, thus incurring no overhead. In this case, the A and L policies will only be applicable to the R memory placement policy where they dictate the resolution at which data is read from the disk. In the case of the I policy, results will be identical for A and L. This special case is representative of a BDMPI runtime without SBMA and acts as a baseline to compare the other schemes against.

For each micro-benchmark we reported the average throughput, namely, the number of system pages operated on per second. This is obtained by timing the entire `for` loop of each micro-benchmark then dividing the total time by the number of system pages in the allocation. Further, each time reported is the average throughput of ten executions of the synthetic application. Due to the precautions taken, the timings results were extremely stable, $< 0.5\%$ error, thus no error statistics are reported.

6.1.2 *Real-world Applications*

The implementation of PageRank was the same as that used in [9] and uses a one-dimensional row-wise decomposition of the sparse adjacency matrix. Each MPI process gets a consecutive set of rows such that the number of non-zeros of the sets of rows assigned to each process is balanced. Each iteration of PageRank is performed in three steps using a *push* algorithm [13]. The ParMetis implementation is version 4.0.3 and was downloaded directly from the author's website [15]. The code was modified by changing a single line to disable the use of its own workspace management and thus allow SBMA to manage its memory allocations. The SPLATT implementation is a version that was obtained directly from the author of [18] and uses a three-dimensional decomposition of the sparse adjacency tensor. Each MPI process gets a consecutive set of rows such that the number of non-zeros of the sets of rows assigned to each process is balanced. For our experiments, we compute 16 factors. Each iteration updates the factorization using the alternating least squares method and checks for convergence.

For the three benchmarks, we gathered results by performing five executions of each application. The times that we report correspond to the average time required to perform each iteration, which was obtained by dividing the total time by the number of iterations. As a result, the reported times include the costs associated with loading and storing the input and output data. The number of running slaves was set to one in all cases, and the total number of slaves was chosen as the smallest number of slaves such that the required memory for each slave fit completely in physical memory.

The performance gains achieved by the SBMA subsystem when the application is executed on a single node are expected to carry over when the program is executed on multiple nodes. For this reason, the focus of our experiments was within a single node. However, in some cases, the SBMA subsystem may change the scheduling within a node which can potentially affect performance in a multi-node system. Thus, to demonstrate the performance of an SBMA enabled BDMPI runtime beyond a single node, we have also included a brief set of experiments which involve execution of applications on multiple nodes.

6.2 Datasets

Table 1: The performance of baseline methods on the different applications.

				#Nodes=1		#Nodes=4	
Application	Dataset	Mem (GB)	Serial	MPI	BDMPI w/o SBMA	MPI	BDMPI w/o SBMA
PageRank	uk-2007-05	35	14.84	> 160.00	19.86	10.25	4.34
ParMetis	nlp-kkt240	13	> 300.00	> 300.00	255.17	N/A	N/A
SPLATT	NELL-large	26	13.52	> 160.00	38.32	5.38	4.01

Each row represents an application and includes the dataset used, the aggregate amount of memory required by all slaves, the runtime for the problem using a single process, standard MPI and BDMPI without SBMA on a single node and on four nodes. All times reported are in minutes. The MPI and BDMPI experiments use the same number of process/slaves as the experiments in Section 7. For example the PageRank problem uses 12 slaves, so on a four nodes, each node would be assigned three process for MPI and three slaves for BDMPI.

For the PageRank experiments the undirected version of the uk-2007-05 [1] web graph was used, with 105 million vertices and 3.3 billion edges. To ensure that the performance of the PageRank algorithm was not affected by a favorable ordering of the vertices, the vertices of the graph were renumbered randomly. For the ParMetis experiments the undirected nlpkkt240 [17] graph was used, with 28 million vertices and 760 million edges. For the SPLATT experiments the NELL-large [3] dataset was used, with 2.9 million rows, 2.14 million columns, 25.5 million fibers, and a total of 143.6 million non-zero elements. The dataset was randomly permuted to ensure the SPLATT algorithms were not affected by favorable tensor ordering.

6.3 System Configuration

The experiments were run on a dedicated cluster consisting of four Dell Optilex 9010s. Each machine is equipped with an Intel Core i7 @ 3.4GHz processor, 4GB of memory, and a Seagate Barracuda 7200RPM 1.0TB hard drive. Because of BDMPI's dependence on the swap-file for data storage when SBMA is disabled, the machines were set up with 300GB swap partitions. The four machines run the Ubuntu 14.04.1 LTS distribution of the GNU/Linux operating system. The C compiler used was GNU GCC 4.8.2 and the MPI implementation was MPICH 3.0.4.

7. RESULTS

7.1 Synthetic Benchmark

Table 2 shows the throughput for the three different memory access operations: read, write, and read/write. The most relevant results with respect to the BDMPI system as a whole are the rows labeled AR and LR. When any of the columns representing `sbpage` sizes are compared against the np column, the result is a quantification of the overhead of the SBMA system. In this respect, Table 2 reveals two important results.

First, by comparing the throughput of the np scheme for a given memory operation to the throughput for the corresponding SBMA operation for each of the `sbpage` sizes, we can derive an upper limit to the overhead associated with SBMA under the set of conditions proposed in this benchmark. Thus, we see that any given memory operation is at most 2.4 times slower under SBMA than in a system which defers memory management to the OS. For a sufficiently large `sbpage` size, the overhead introduced by the implicit signal driven memory handling of SBMA can be greatly reduced, in some cases nearly to the performance of explicit memory handling, see the read (==) operation for `sbpage` size 64 using the AI or LI policy which both have throughput which is less than 3% lower than the np case. Second, by comparing the results for all of the experiments involving disk I/O, we we see that the results for all experiments, including those of the np scheme are identical within a given policy. Thus, we can conclude that the overhead related to the signal handling mechanisms inherent to SBMA are overshadowed by the disk I/O. Furthermore, since all of these results are nearly the same, even when comparing the two policies, it suggests that the disk bandwidth is the limiting factor for operations involving disk I/O.

The read (==) and read/write (+=) experiments of the AI and LI policies can be used to further quantify the overhead associated with the signal handling mechanism. By comparing the results of these experiments between A and L, we see that the L policy introduces a 17.5% overhead on average. The magnitude of the overhead for a given `sbpage` size is proportional to the `sbpage` size itself. This is as we would expect, since the L policy dictates that the memory protection of each `sbpage` must be updated independently versus the A policy which has the memory protection of the entire allocation updated at once. This overhead holds for the write(=) operation as well, since by definition, a write fault in SBMA must first generate a read fault.

7.2 Performance of PageRank

Table 3 shows the performance achieved by the different programs on the PageRank benchmark. The PageRank benchmark does not include any results for variations of the SBMA threshold. The reason for this is that the code requires a fixed set of allocations, all of which are larger than any reasonable SBMA threshold. Thus, for the PageRank benchmark, SBMA will necessarily manage all allocations.

Comparing the performance achieved by the SBMA variations with the single node results in Table 1, we see that all configurations of SBMA lead to a decreased runtime against both MPI and BDMPI without SBMA. BDMPI with SBMA is 1.6×–1.8× and > 12× faster than BDMPI without SBMA and MPI respectively. For the multi-node experiments in the last two columns of Table 1, the performance improvements were similar to the single node experiments. In these cases, BDMPI with SBMA was 1.51× and 3.58× faster on

Table 2: Throughput of Memory Operations on the Micro-Benchmark.

	Read (x == y)					Write (x = y)					Read/Write (x += y)				
	np	1	4	16	64	np	1	4	16	64	np	1	4	16	64
AI	1195	1195	1194	1194	1194	514	288	373	405	414	472	276	352	380	387
LI	1195	537	927	1134	1134	514	208	325	379	395	472	198	310	359	373
AR	28	28	28	28	28	514	288	373	405	414	28	28	28	28	28
LR	30	30	30	30	30	514	208	325	379	395	30	30	30	30	30

Throughput, in system pages/sec, for memory access operations under different memory management strategies using a variety of `sbpage` sizes from 1 to 64. The first column for each memory access operation is the **np** scheme, where memory is obtained via a call to `mmap()` and no memory protection policies are applied to it. **A** is aggressive, **L** is lazy, **I** is in-memory, and **R** is reading. The difference between AI and AR is that in the former, data resides only in memory, while in the latter, the data is read from disk before each read access. The same is true for LI and LR.

Table 3: SBMA performance on PageRank.

	#Nodes=1			#Nodes=4
	4	16	64	64
ARAW	10.96	10.93	10.53	2.82
ARLW	13.75	13.81	12.55	3.37
LRLW	13.91	12.13	10.28	2.52

Runtime results, in minutes, for PageRank run on the uk-2007-05 graph. The application was run on one node with 12 slaves and four nodes with three slaves per node, with one running slave per node for both.

average than BDMPI without SBMA and MPI respectively. Surprisingly, the performance improvement over the serial application, also in Table 1 was more modest. However, as was discussed in [9], the serial version, by its very nature has a few advantages over a parallel implementation. First, since the graph is randomly permuted, distributing it to the 12 slaves is a complex operation. Second, each iteration of the computation requires an all-to-all communication. Both of these computations introduce a non-trivial amount of work into the parallel execution.

For the PageRank experiments, the performance improvement achieved by the SBMA variations can be credited to the cooperative multi-tasking model of the BDMPI runtime. In this type of application, where nearly all of the data is accessed between blocking points, the main advantage of using a system like SBMA is that it automates the bulk load and unload before and after each blocking operation. To prove this, we refer to the results from [9], where the authors performed the same PageRank experiment using BDMPI without SBMA and a version of the PageRank application with explicit read/write to disk before/after each MPI blocking operation. The per iteration runtime for that experiment was 9.98 min for single node and 2.35 min for four node. Thus, the best of the BDMPI with SBMA runs were $< 5\%$ slower on average than an application optimized by hand.

Comparing only the performance of the various memory exchange strategies, we see that ARAW performs the best whereas ARLW and LRLW perform worse, but roughly the same, for all `sbpage` sizes. However, the performance difference between all three variations is within 10%. Since each iteration of the algorithm requires that the processes iterate their memory allocations entirely, there is no obvious advantage of using an LR based approach. Further, since the number of slaves is chosen such that the running slave will occupy a majority of available system memory, whenever a process is in the blocked state, it will be required to unload its memory, nullifying any advantage of an LW based approach. Thus, we would expect that if there was no overhead associated with LR and LW, the runtimes for ARLW and LRLW would be comparable to ARAW. However, as we see, the results are slightly worse, which suggests that there is an non-trivial overhead incurred by the resident memory tracking protocol of LR and LW, which is supported by the results of the micro-benchmark in Section 7.1.

To further explore this overhead, we can compare the performance of ARLW and LRLW, where we see that nothing is gained by the additional selectivity introduced by lazy reading, and the runtime actually increases. This can be explained by referring to the results of the synthetic benchmark, see Table 2, where it was shown that LR based schemes incur an overhead for the write operation when compared with AR based schemes. In fact the discrepancy in runtimes between ARLW and LRLW for corresponding `sbpage` sizes is inversely proportional to the discrepancy in write operation throughput between AR and LR in Table 2 for the same `sbpage` sizes.

Lastly, comparing the results of each of the three multi-node SBMA variations against the corresponding single node experiments, we see that the speedup is super-linear. In fact, the best speedup achieved by any of the three variations was LRLW which achieved a speedup of 8×. The reason for this super-linear speedup is due to the fact that the aggregate amount of memory in the four nodes is higher, which allows the slaves to retain more of their data in memory between successive blocking points.

7.3 Performance of ParMetis

Figure 3 shows the performance achieved by the different SBMA configurations on the ParMetis benchmark. This figure, in addition to the set of experiments in which the `sbpage` size is varied, includes results for variations of the SBMA threshold. These were included because at various points during its execution, ParMetis can allocate chunks of memory whose size is below the SBMA threshold.

The results for the ParMetis benchmark for all SBMA varia-

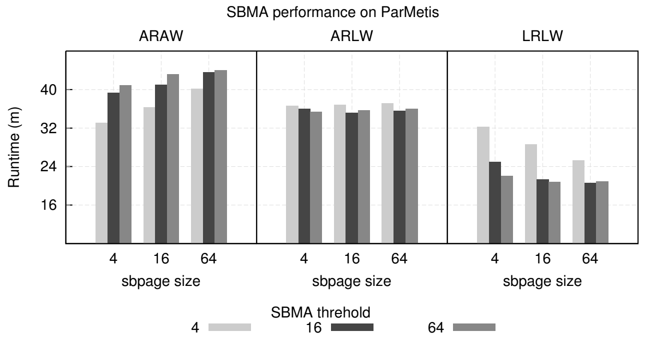


Figure 3: Runtime results, in minutes, for ParMetis run on the nlpkkt240 graph. The application was run on one node with four slaves, with one running slave per node.

Table 4: Amount of disk I/O for ParMetis.

	Read			Write		
	4	16	64	4	16	64
ARAW	116	117	118	52	56	62
ARLW	67	67	68	38	40	45
LRLW	25	28	32	38	40	45

Number of GB transferred to / from disk during the execution of ParMetis on a single node with four slaves for the three SBMA variations and three different `sbpage` sizes.

tions show a much larger performance improvement over the execution of BDMPI without SBMA, found in Table 1, when compared with the results of the PageRank benchmark. In this case, all three of the SBMA approaches outperformed BDMPI run without SBMA with speedup ranging from 7× to 12×. The results compared with the serial execution, also found in Table 1, are even better, where BDMPI with SBMA achieved speedup > 20×. The poor performance of serial ParMetis and BDMPI without SBMA can be attributed to the reliance on the OS VMM swapping mechanism during the first and last few stages of the ParMetis algorithm. The consequence is the same as for the PageRank benchmark, namely, increased competition for disk resources.

Comparing the performance achieved by the various memory exchange strategies, we see that LRLW performs the best whereas ARAW performs the worst, with ARLW in between. For the ParMetis benchmark, the runtimes of ARLW are 10% less than ARAW on average and those of LRLW are 33% less than ARAW on average. This increased gap in performance is expected for an application like ParMetis, which is an example of the multi-level paradigm. In its initial phases of execution, the graph being partitioned is contracted into successively smaller graphs, thus each successive phase requires a lesser amount of memory be loaded than the previous phase. This explains the performance improvement of ARLW over ARAW, since during the later phases of contraction, the memory of all slave processes can fit in the available system memory. This fact makes the loading and unloading of entire allocations, as in ARAW, unnecessary. Then, in the later phases of execution, the graph is un-contracted and refined. In most cases, although the un-

Table 5: SBMA performance on SPLATT.

	#Nodes=1			#Nodes=4
	4	16	64	64
ARAW	12.01	13.04	13.47	2.23
ARLW	11.46	12.28	11.72	2.13
LRLW	11.92	8.41	7.17	2.57

Runtime results, in minutes, for SPLATT run on the NELL-large graph. The application was run on one node with 8 slaves and four nodes with two slaves per node, with one running slave per node for both.

contracted graph will grow from phase to phase, the amount of memory accessed will remain small relative to the size of the un-contracted graph because only the information associated with the interface vertices is accessed. Since only a relatively small amount of memory is being loaded during each of the un-contraction phases, ARAW and ARLW both suffer from excessive loading and unloading of memory, due to the transfer of data to and from disk at the allocation resolution. Table 4 presents the number of GB transferred to/from disk and shows that although LRLW writes roughly the same amount of data as ARAW and ARLW, it reads 75% less data than ARAW and 57% less than ARLW. The characteristics of these two phases, contraction and uncontraction, make LRLW ideally suited for an application like ParMetis, which is supported by the results.

Looking at just the effect of `sbpage` size on the three strategies we see that each is affected differently. ARAW is affected adversely, ARLW neutral, and LRLW positively. To understand this, we have to consider the implications of using different `sbpage` sizes. Under all three strategies, the consequence of a larger `sbpage` size is that the number of bytes included in `DIRTY sbpages` can be unnecessarily increased if the application is not writing to all bytes within an `sbpage`. This is very likely to happen during later contraction and un-contraction phases, when the memory access pattern becomes more sparse. As Table 4 shows, in all cases, as `sbpage` size increase, so does disk I/O. Since this is the only effect of `sbpage` size on ARAW we see that ARAW's runtime increases as the `sbpage` size increases. In ARLW, the increased disk I/O is offset by the nature of lazy writing and its expression in ParMetis. Many of the ParMetis iterations which are likely to cause unnecessary disk I/O, due to sparse memory accesses, are also likely to be candidates to benefit from a lazy write strategy. Like ARLW, LRLW offsets the effects of sparse memory accesses using lazy writing, but has one additional advantage. As `sbpage` size increases, the memory protection overhead related to the lazy reading strategy decreases. This effect of `sbpage` size was confirmed in Table 2. Thus, LRLW realizes a new gain in performance as `sbpage` size increases as can be seen in Figure 3.

7.4 Performance of SPLATT

Table 5 shows the performance achieved by the different program configurations on the SPLATT benchmark. These results do not include any variations of the SBMA threshold for the same reason it was omitted in the PageRank benchmark, namely that the code requires a fixed set of allocations, all of which are larger than any reasonable SBMA

Table 6: Amount of disk I/O for SPLATT.

	Read			Write		
	4	16	64	4	16	64
ARAW	70	70	70	24	24	24
ARLW	51	51	51	20	20	20
LRLW	23	23	23	20	20	20

Number of GB transferred to / from disk during the execution of SPLATT on a single node with 8 slaves for the three SBMA variations and three different `sbpage` sizes.

threshold.

For the single node experiments, the characteristics of the SPLATT experiments were similar to those of the ParMetis experiments. Namely that the addition of SBMA improved performance in all cases when compared with the serial version and BDMPI without SBMA. Also, despite the additional overhead of lazy reading and writing, performance can be improved by enabling these optimizations. Like the ParMetis results, the performance improvement between the three schemes can be explained most easily by referring to Table 6, which shows that the amount of disk I/O for SPLATT. Like ParMetis, ARAW was the slowest and also had the highest amount of disk I/O and LRLW was the fastest and had the lowest amount of disk I/O.

For the multi-node experiments, the results are generally as expected. Each of the three strategies performed had runtimes 3×–6× faster than their single node counterpart. As anticipated, ARLW was faster than ARAW. However, not as expected was the performance of LRLW compared with the other strategies. As it turns out, the aggregate memory on the four nodes was large enough to support a great deal of deferred writes. Thus, the relatively low disk I/O overall, combined with the overhead of lazy reading, actually had a negative impact in the LRLW scheme, causing its runtime to be larger than both ARLW and ARAW. For reference, the amount of data written to disk was 24GB, 14GB, and 20GB for ARAW, ARLW, and ARLW respectively.

8. CONCLUSION

In this paper we presented various methods to automatically manage the virtual memory of parallel applications running on memory constrained systems. By leveraging how BDMPI controls the execution of different processes, we were able to optimize resident memory exchange to reduce data transfer to and from disk. Our results showed that SBMA offers performance gains over applications executed using a BDMPI runtime that relies on the OS VMM to manage resident memory.

9. ACKNOWLEDGMENT

This work was supported in part by NSF (IIS-0905220, OCI-1048018, CNS-1162405, IIS-1247632, IIP-1414153, IIS-1447788), Army Research Office (W911NF-14-1-0316), Intel Software and Services Group, and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

10. REFERENCES

[1] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[2] A. D. Brown. Explicit compiler-based memory management for out-of-core applications. Technical report, Stanford University, 2005.

[3] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka, and T. M. Mitchell. Toward an architecture for never-ending language learning. In *In AAAI*, 2010.

[4] R. Elmasri and S. Navathe. *Fundamentals of database systems*. Springer, 2009.

[5] D. Engler, S. Gupta, and M. Kaashoek. Avm: application-level virtual memory. In *Hot Topics in Operating Systems, 1995. (HotOS-V), Proceedings., Fifth Workshop on*, pages 72–77, May 1995.

[6] B. V. Essen, H. Hsieh, S. Ames, and M. Gokhale. Di-mmap: A high performance memory-map runtime for data-intensive applications. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 731–735, Washington, DC, USA, 2012. IEEE Computer Society.

[7] Apache^{TM}Hadoop®. `http://hadoop.apache.org`.

[8] G. Karypis, K. Schloegel, and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[9] D. LaSalle and G. Karypis. Mpi for big data: New tricks for an old dog. *Parallel Computing*, 40(10):754–767, 2014.

[10] C.-H. Lee, M. C. Chen, and R.-C. Chang. Hipec: High performance external virtual memory caching, 1994.

[11] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[12] M. R. Meswani, G. H. Loh, S. Blagodurov, D. Roberts, J. Slice, and M. Ignatowski. Toward efficient programmer-managed two-level memory hierarchies in exascale computers. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, pages 9–16. IEEE, 2014.

[13] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[14] Y. Park, R. Scott, and S. Sechrest. Virtual memory versus file interface for large, memory-intensive scientific applications. In *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pages 53–53, 1996.

[15] ParMetis. `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.

[16] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. J. Emer. Adaptive insertion policies for high performance caching. In *In Proceedings of the 35th International Symposium on Computer Architecture*, 2007.

[17] O. Schenk, A. Wächter, and M. Weiser. Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. *SIAM Journal on Scientific Computing*, 31(2):939–960, 2008.

[18] S. Smith and G. Karypis. Dms: Distributed sparse tensor factorization with alternating least squares. Technical Report 15-007, Department of Computer Science and Engineering, University of Minnesota, May 2015.

[19] Y. suk Kee, J.-S. Kim, and W.-C. Jeun. Atomic page update methods for openmp-aware software dsm. In *Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on*, pages 144–151, Feb 2004.

[20] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms and Visualization*, 50:161–179, 1999.

[21] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale. Di-mmap:a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18(1):15–28, 2015.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.