Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

MPI for Big Data: New tricks for an old dog

Dominique LaSalle *, George Karypis

Department of Computer Science & Engineering, University of Minnesota, Minneapolis, MN 55455, USA

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
Out-of-core
Distributed
High performance
Big Data

ABSTRACT

The processing of massive amounts of data on clusters with finite amount of memory has become an important problem facing the parallel/distributed computing community. While *MapReduce*-style technologies provide an effective means for addressing various problems that fit within the MapReduce paradigm, there are many classes of problems for which this paradigm is ill-suited. In this paper we present a runtime system for traditional MPI programs that enables the efficient and transparent out-of-core execution of distributed-memory parallel programs. This system, called BDMPI,[1] leverages the semantics of MPI's API to orchestrate the execution of a large number of MPI processes on much fewer compute nodes, so that the running processes maximize the amount of computation that they perform with the data fetched from the disk. BDMPI enables the development of efficient out-of-core parallel distributed memory codes without the high engineering and algorithmic complexities associated with multiple levels of blocking. BDMPI achieves significantly better performance than existing technologies on a single node as well as on a small cluster, and performs within 30% of optimized out-of-core implementations.

1. Introduction

The dramatic increase in the size of the data being collected and stored has generated a lot of interest in applying data-driven analysis approaches, commonly referred to as *Big Data*, in order to gain scientific insights, increase situational awareness, improve services, and generate economic value. The amount of the data coupled with the complexity of the analysis that often needs to be performed, necessitates the use of analysis algorithms that do not load all the data in memory and the use of distributed/parallel computing platforms. The first requirement stems from the fact that the amount of DRAM in most modern computers and moderate-size clusters is no longer sufficient to store and analyze these large datasets, whereas the second requirement is designed to address the computational requirements of the analysis.

In recent years, there has been a burst of research activity in developing frameworks for out-of-core distributed computing applications (i.e., distributed computing applications that primarily store their data on the disk). Some of the most significant outcomes of this research have been the functional programming model used by the MapReduce [1] and associated frameworks (e.g., Hadoop [2], Spark [3]) and the vertex-centric model used by various graph-based distributed processing frameworks (e.g., Pregel [4], Hama [5], GraphLab [6], Giraph [7]). These frameworks use specific computational models that enable the efficient expression and execution of applications whose underlying computational structure fit these models well. However, there is a large number of applications whose computational structure does not fit these existing frameworks well, which makes it difficult to express the computations and/or efficiently utilize the underlying computational resources.

* Corresponding author.
E-mail address: lasalle@cs.umn.edu (D. LaSalle).

[1] The source code is available at http://glaros.dtc.umn.edu/gkhome/bdmpi/download.

http://dx.doi.org/10.1016/j.parco.2014.07.003
0167-8191/© 2014 Published by Elsevier B.V.

The objective of this work is to provide a message-passing out-of-core distributed computing framework that can achieve high performance while simultaneously is flexible to allow the expression of computations required by a wide-range of applications. The key insight underlying our approach is the observation that scalable distributed-memory parallel applications (e.g., those written in MPI [8]) tend to exhibit two characteristics: (i) they are *memory scalable* in the sense that the memory required by each process decreases as the number of processes used to solve a given problem instance increases, and (ii) they exploit *coarse grain* parallelism in the sense that they structure their computations into a sequence of local computation followed by communication phases in which the local computations take a non-trivial amount of time and often involve a non-trivial subset of the process' memory.

Relying on these observations, we developed a framework for out-of-core distributed computing that couples scalable distributed memory parallel programs written in MPI with a runtime system that facilitates out-of-core execution. In this framework, which we implemented in the form of an MPI library and its associated runtime system, collectively referred to as *Big Data MPI* (BDMPI), the programmer needs to only develop a memory-scalable parallel MPI program by assuming that the underlying computational system has enough computational nodes to allow for the in-memory execution of the computations. This program is then executed using a sufficiently large number of processes so that the per-process memory fits within the physical memory available on the underlying computational node(s). BDMPI maps one or more of these processes to the computational nodes by relying on the OS's virtual memory management to accommodate the aggregate amount of memory required by them. BDMPI prevents memory thrashing by coordinating the execution of these processes using node-level co-operative multi-tasking that limits the number of processes that can be running at any given time. This ensures that the currently running process(es) can establish and retain memory residency and thus achieve efficient execution. BDMPI exploits the natural blocking points that exist in MPI programs to transparently schedule the co-operative execution of the different processes. In addition, BDMPI's implementation of MPI's communication operations is done so that to maximize the time over which a process can execute between successive blocking points. This allows it to amortize the cost of loading data from disk over the maximal amount of computations that can be performed.

We experimentally evaluated the performance of BDMPI on three problems (*K*-means, PageRank, and stochastic gradient descent). Our experiments show that BDMPI programs often perform within 30% of optimized out-of-core codes, two to five times faster than GraphChi, and up to a 100 times faster than Hadoop. Furthermore, whereas as other attempts to solve this problem propose new programming paradigms, we use the well established MPI semantics and API, which over the past twenty years have been shown to allow the efficient expression of a wide variety of algorithms.

This paper is organized as follows. In Section 2 we review related approaches. We examine the insights that led us to develop BDMPI in Section 3. In Section 4 we present an overview of BDMPI and its interface. In Section 5 we describe the implementation of BDMPI. In Section 6 we discuss the conditions of our experimental evaluation of BDMPI, and in Section 7 we discuss the results of this evaluation. Finally in Section 8 we give an overview of our findings.

2. Related work

The problem of processing datasets that do not fit within a system's memory has received significant attention. The two major issues are the engineering effort required to develop out-of-core codes and the challenge of extracting performance from such codes. The solutions proposed so far range from low-level strategies for engineering high-performance solutions, to high-level and often domain-specific frameworks that attempt to minimize the engineering effort.

The best performance is achieved by directly engineering problem specific out-of-core codes and taking advantage of any available shortcuts the individual problems present. Vitter provides a survey of explicit out-of-core algorithms in [9]. In [10], Bordawekar and Choudhary present strategies for out-of-core and distributed computing. A survey specifically for linear algebra out-of-core algorithms is presented in [11]. Despite these general strategies, the engineering cost is still extremely high.

In an attempt to greatly reduce the effort needed for large scale data processing, MapReduce [1] has come to represent a class of software for processing Big Data. The MapReduce model is comprised of two phases: *map* and *reduce*. In the map phase, a list of key-value pairs are generated/gathered for a specific computational task. Then in the reduce phase, a computation is performed on all of the key-value pairs from the map phase, and the result is saved. This allows for task-parallel execution, where compute nodes can pick up map and reduce tasks to perform. A theoretical foundation for the MapReduce model is provided in [12]. A popular and publicly available implementation of MapReduce is that of Hadoop [2]. This popularity has also led to the development of domain specific libraries. The Mahout [13] library provides a set of machine learning algorithms. The Pegasus framework [14] provides several Hadoop versions of different graph mining algorithms targeting massive graphs. The MapReduce paradigm's ability to efficiently express and execute iterative computations is limited, as it results in unnecessary data movement.

To address this short-coming, a modification to the MapReduce model was proposed by Bu et al., called Haloop [15]. Based on Hadoop, Haloop is specialized for handling iterative problems with a modified task scheduler and the ability to cache frequently used data. However, there are still several classes of algorithms which do not fit the MapReduce paradigm or its extensions.

Originally developed by Valiant [16], the *Bulk Synchronous Parallel* model (BSP), aimed to provide a theoretical foundation for parallel algorithms that accounts for communication and synchronization costs. Algorithms following the BSP model contain three steps per iteration: *computation*, *communication*, and *synchronization*. The Hama [5] project provides a BSP framework that runs on top of the Hadoop Distributed File System. Hama attempts to improve data locality for matrix and graph based algorithms, as well as provide a framework capable of expressing a wider range of problems.

Inspired by BSP style computations, Pregel [4] and GraphLab [6] provide graph-specific frameworks for distributed graph processing. They work on a vertex-centric model, where each computation takes the form of an operation on a vertex and its connected edges. Both of these technologies currently can only run *in-memory*, requiring massive amounts of DRAM to tackle large problems. Apache's open source framework Graph [7] extends these by adding out-of-core computation capabilities for processing extremely large graphs. GraphChi [17], based on GraphLab's computational model, provides an efficient platform for out-of-core graph processing but does not support distributed execution.

Charm++ [18] is a parallel programming system based on the migratable-objects programming model. The programmer decomposes the computations into a large number of work objects, referred to as *chares*, that interact with each other via method invocations. Charm++'s runtime system maintains a work-pool of chares that can be executed and schedules their execution based on the available resources. Charm++'s runtime system provides support for disk-based processing via disk-based storing and prefetching of the data associated with each chare object [19].

3. Motivation of the approach

The general approach used by algorithms that are designed to operate on problems whose memory requirements exceed the amount of available physical memory is to structure their computations into a sequence of *steps* such that the working set of each step can fit within the available physical memory and the data associated with each step can be loaded/stored from/to the disk in a disk-friendly fashion (e.g., via sequential accesses or via a small number of bulk accesses) [9].

Scalable distributed memory parallel algorithms share a common structure as well [20]. In these algorithms, the computations are decomposed into different tasks and each task along with its associated data is mapped on the available compute nodes. This decomposition is optimized so that it maximizes the computations that can be done with the local data (i.e., maximize locality) and reduce the frequency as well as the volume of the data that needs to be communicated across the nodes (i.e., minimize communication overheads). In addition, most of these algorithms structure their computations into a sequence of *phases* involving a local computation step followed by inter-process communication step. Moreover, if M is the amount of memory required by a serial algorithm for solving a given problem instance, the amount of memory required by each process is $O(M/p) + f(p)$, where p is the number of processes involved and $f()$ is often a sub-linear function on p. An example of this is the memory required for communication structures in finite element computations where each processor communicates updates to each of its neighbors.

The key observation motivating our work is that a scalable distributed memory parallel algorithm can be transformed into an algorithm whose structure is similar to that used by out-of-core algorithms. In particular, if p is the number of processes required to ensure that the per-process memory fits within the compute node's available physical memory, then the computations performed by each process in a single phase will correspond to a distinct step of the out-of-core algorithm. That is, one parallel phase will be executed as p sequential steps. Since the working set of each of these steps fits within the physical memory of a node, the computations can be performed efficiently. Moreover, if the underlying computational infrastructure has n available nodes, each node will perform p/n of these steps in sequence, leading to a distributed out-of-core execution.

BDMPI performs this transformation in a way that is entirely transparent to the programmer. It uses the OS's virtual memory management (VMM) mechanisms to provide the programmer with the illusion that the parallel program is operating as if all the data could fit in memory and, when appropriate, uses disk-based message buffering to ensure the correct and efficient execution of the communication operations. Note that even though our discussion so far has focused on BSP-style parallel programs, as the subsequent sections will illustrate, BDMPI works for non-synchronous programs as well.

4. 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 with its own job execution command (`bdmpiexec`). 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.

A BDMPI program is a standard MPI-based distributed memory parallel program that is executed using the `bdmpiexec` command as follows:

```
bdmpiexec -nn nnodes
          -ns nslaves
          -nr nrunning
          progname [arg1] [arg2] ...
```

The `nnodes` parameter specifies the number of compute nodes (e.g., machines in a cluster) to use for execution, the `nslaves` parameter specifies the number of processes to spawn on each of the `nnodes` nodes, and the `nrunning` parameter specifies the maximum number of slave processes that can be running at any given time on a node. The name of the BDMPI program to be executed is progname, which can have any number of optional command-line arguments. This command will create an MPI execution environment consisting of nnodes × nslaves processes, each process executing progname. In this environment, these processes will make up the MPI_COMM_WORLD communicator.

BDMPI uses two key elements in order to enable efficient out-of-core execution. The first relates to how the MPI processes are executed on each node and the second relates to the memory requirements of the different MPI processes. We will refer to the first as BDMPI's *execution model* and to the second as its *memory model*.

While the nrunning parameter can be used to allow more than one slave process to be run on each node concurrently and exploit the processing power of multiple compute cores using pure MPI codes, the focus of BDMPI is inter-node parallelism. Hybrid approaches combining MPI with a threading technology have been shown for some problems to lead to increased performance and reduced memory footprints compared to pure MPI approaches [21]. BDMPI allows for both models of intra-node parallelism and achieving performance with such parallelism is left up to the programmer.

BDMPI's execution model is based on *node-level co-operative multi-tasking*. BDMPI allows only up to nrunning processes to be executing concurrently with the rest of the processes blocking. When a running 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 MPI processes spawned on a node to be greater than the amount of physical memory on that node. However, it requires that the sum of the memory required by the nrunning processes to be smaller than the amount of physical memory on that node. Within this model, an unmodified MPI program will rely on the OS's VMM mechanisms to map in memory the data that each process needs during its execution. Alternatively, the program can be explicitly optimized for BDMPI's memory and execution model. Two ways of achieving this is for the program, at possible blocking/resumption points, to (i) use memory locking/unlocking to prefetch from the swap file and subsequently release the parts of the address space that it needs, or (ii) use file I/O to explicitly load/store the data that it needs from the disk and thus bypass most of the VMM mechanisms.

The coupling of constrained memory over-subscription with node-level co-operative 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 sufficient amount of physical memory to accommodate all the processes that are allowed to run.

The importance of the last part can be better understood by considering what will happen if the nslaves processes were allowed to execute in the standard pre-emptive multi-tasking fashion. In such a scenario, each of the nslaves processes will execute for a period of time corresponding to the OS's time-slice and then relinquish the core that they were mapped on so that another process can be scheduled. Due to memory over-subscription, such an approach will provide no guarantees that any of the process' memory that was mapped in physical memory in one time-slice will be there for the next time-slice the process is scheduled, potentially resulting in severe memory thrashing.

Given BDMPI's execution and memory model, we can see that the optimal number for the nrunning parameter is determined by the number of physical cores on the nodes, the ability of its disk subsystem to service concurrent requests, and the amount of memory required by each MPI process. Among these parameters, the disk subsystem is often the rate limiting component and its ability to allow for more running processes depends on the number of spinning disks and/or the use of SSDs.

BDMPI dramatically lowers the burden of developing out-of-core distributed programs by allowing programmers to focus on developing scalable parallel MPI programs and leave the aspects related to out-of-core execution to BDMPI. This also increases the portability of programs, because when the memory in the system is sufficient, the program can be executed as a regular MPI program.

5. Implementation of BDMPI

From the developer's view, BDMPI consists of two components. The first is the bdmpiexec program used to execute a BDMPI (MPI) program on either a single or a cluster of workstations, and the second is the bdmpilib library that provides the subset of the MPI 3 that BDMPI implements, which needs to be linked with the application code. The subset of the MPI that is currently implemented in BDMPI is shown in Table 1. This contains a reasonable set of MPI functions for developing a wide-range of message passing programs. Note that since BDMPI is built on top of MPI and itself uses MPI for parallel execution, we have prefixed the MPI functions that BDMPI supports with "BD" in order to make the description of BDMPI's implementation that follows unambiguous.

In the rest of this section we provide information on how BDMPI's node-level co-operative multi-tasking execution is implemented and how the different classes of MPI functions are implemented as to adhere to its memory model.

5.1. Master and slave processes

The execution of a BDMPI program creates two sets of processes. The first are the MPI processes associated with the program being executed, which within BDMPI, they are referred to as the *slave* processes. The second is a set of processes, one on

Table 1
The MPI subset implemented by BDMPI.

`BDMPI_Init`, `BDMPI_Finalize`
`BDMPI_Comm_size`, `BDMPI_Comm_rank`, `BDMPI_Comm_dup`, `BDMPI_Comm_free`, `BDMPI_Comm_split`
`BDMPI_Send`, `BDMPI_Isend`, `BDMPI_Recv`, `BDMPI_Irecv`, `BDMPI_Sendrecv`
`BDMPI_Probe`, `BDMPI_Iprobe`, `BDMPI_Test`, `BDMPI_Wait`, `BDMPI_Get_count`
`BDMPI_Barrier`
`BDMPI_Bcast`, `BDMPI_Reduce`, `BDMPI_Allreduce`, `BDMPI_Scan`, `BDMPI_Gather[v]`, `BDMPI_Scatter[v]`, `BDMPI_Allgather[v]`, `BDMPI_Alltoall[v]`

each 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.

The master processes are implemented by a program called `bdmprun`, which itself is a parallel program written in MPI (not BDMPI). When a user program is invoked using `bdmpiexec`, the `bdmprun` program is first loaded on the nodes of the cluster and then proceeds to spawn the slave processes.

The organization of these processes into the set of slave processes for BDMPI's node-level co-operative multi-tasking execution model is done when each process calls its corresponding `BDMPI_Init` function. At this point, each slave is associated with the master process that spawned it, creates/opens various structures for master-to-slave interprocess communication, and receives from the master all the necessary information in order to setup the MPI execution environment (e.g., `BDMPI_COMM_WORLD`). Analogously, when a slave calls the `BDMPI_Finalize` function, its master removes it from the set of slave processes involved in co-operative multitasking, and its execution resumes to follow the regular pre-emptive multi-tasking.

All communication/synchronization operations between the slaves go via their master processes. These operations are facilitated using POSIX shared memory for master/slave bi-directional data transfers, POSIX message-queues for slave-to-master signaling, and MPI operations for intra-node communication/synchronization. For example, if a message is sent between two MPI processes p_i and p_j that are mapped on nodes n_x and n_y, then the communication will involve processes $p_i \rightarrow m_x \rightarrow m_y \rightarrow p_j$, where m_x and m_y are the master processes running on nodes n_x and n_y, respectively. Process p_i will signal m_x that it has a message for p_j and transfer data to m_x via shared memory (assuming that the message is sufficiently small), m_x will send the data to m_y via an `MPI_Send` operation, and m_y will send the data to p_j via shared memory.

The master processes service the various MPI operations by spawning different POSIX threads for handling them. In most cases, the lifetime of these threads is rather small, as they often involve updating various master state variables and moving small amounts of data from the slave's address space to the master's address space and vice versa. The only time that these threads can be alive for a long time is when they perform blocking MPI operations with masters of other nodes. The multi-threaded implementation of BDMPI's master processes improves the efficiency in handling requests from different slaves and other master processes. It also ensures that collective operations involving multiple disjoint subsets of slave processes across different nodes can proceed concurrently with no deadlocks.

5.2. Node-level cooperative multi-tasking

Node-level co-operative multi-tasking is achieved using POSIX message-queues. Each master creates `nslaves` message queues, one for each slave. We refer to these queues as *go-queues*. A slave blocks by waiting for a message on its go-queue and the master signals that a slave can resume execution by sending a message to the go-queue of that slave. Since Linux (and most other OSs that provide POSIX IPC support) blocks a process when the message queue that it is reading from is empty, this synchronization approach achieves the desired effect without having to explicitly modify the OS's scheduling mechanism.[2]

The master maintains information about the state of the various slaves, which they can be in one of the following states: running (a slave is currently running), *rblocked* (a slave is blocked due to an MPI receive operation), *cblocked* (a slave is blocked due to an MPI collective operation), *runnable* (a slave can be scheduled for execution if resources are available), and *finalized* (a slave has called `BDMPI_Finalize`). The reason for the differentiation between the rblocked and cblocked states is because messages can arrive in a different order than the corresponding calls to `MPI_Recv()` are made, whereas collective communication operations can only finish in the order in which they are called.

The blocking/resumption of the slaves is done jointly by the implementation of the MPI functions in `bdmpilib` and the masters. If a slave calls an MPI function that leads to a blocking condition (more on that later), it notifies its master and then

[2] The OS still schedules the processes that are ready to run in a pre-emptive multi-tasking fashion. However, because BDMPI controls the number of MPI processes that are ready to run, its execution will be similar to that of co-operative multi-tasking as long as it is the only program using the node.

Please cite this article in press as: D. LaSalle, G. Karypis, MPI for Big Data: New tricks for an old dog, Parallel Comput. (2014), http://dx.doi.org/10.1016/j.parco.2014.07.003

blocks by waiting for a message on its go-queue. The master updates the state of the slave to rblocked/cblocked and proceeds to select another runnable slave to resume its execution by sending a message to its go-queue. When a slave receives a message on its go-queue, it proceeds to complete the MPI function that resulted in its blocking and returns execution to the user's program. If more than one slave is at a runnable state, the master selects for resumption the slave that has the highest fraction of its virtual memory already mapped on the physical memory. This is done to minimize the cost of establishing memory residency of the resumed slave. While this is effective for most applications, it does not take into account dependencies between slaves on different nodes. It is possible in a multi-node setting that all of a node's slaves are in a blocked state waiting for communication from blocked slaves on remote nodes, causing the node to be idle. Developing a strategy that balances giving priority to memory residency and the number waiting communication operations is an area of ongoing research.

Since all communication/synchronization paths between the slaves go via their masters, each master knows when the conditions that led to the blocking of one of its slaves may have changed and modify their state from blocked to runnable.

5.3. Send and receive operations

The `BDMPI_Send` and `BDMPI_Isend` operations are performed using a buffered send approach. This is done in order to allow the sending process, once it has performed the necessary operations associated with buffering, to proceed with the rest of its computations. The advantage of this approach is that it maximizes the amount of time over which the running process can amortize the time it spent to establish memory residency.

The buffering of a message depends on its size and on whether the source and destination reside on the same node. If the size of the message is small, then the message is buffered in the memory of the destination's node master process, otherwise it is buffered on the destination's node disk. What constitutes a small message is controlled via a configuration parameter, which is currently one memory page (4 KB). In case of disk-based buffering, the message is written to the disk by the slave, and the name of the file used to store it is communicated to the master. If the destination slave is on a different node, then the master of the sender notifies that master of the destination and sends the data to it via an `MPI_Send` operation. The receiving master will then either store the data in its memory or write them to a file on its disk. In case of memory-based buffering, the data is copied to the master via POSIX shared memory and stored locally or sent to the remote slave's master node. In any of these cases, the master of the destination slave will also change the state of the destination slave to runnable if its current state is rblocked.

The `BDMPI_Recv` operation is performed as follows. The slave notifies its master about the required receive operation. If a corresponding send operation has already been completed (i.e., the data reside on the master's memory or on the local disk), then, depending on the size, the data is either copied to the slave's memory or the slave reads the data from the local disk. Once this is done, the `BDMPI_Recv` operation completes and control is returned to the program. If the corresponding send operation has not been posted, then the slave blocks by waiting on its go-queue. In that case, the master also changes the state of that slave from running to rblocked. When a slave resumes execution (because its master received a message destined for it) it will then check again if the corresponding send has been posted and it will either receive the data or block again. Note that this protocol is required because BDMPI's masters do not maintain information about the posted receive operations but instead only maintain information about the send operations. In the future we plan to investigate any performance benefits of maintaining such information on the masters. For simplicity, BDMPI's implementation of the `BDMPI_Irecv` does nothing other than setting the status information and uses an implementation similar to that for `BDMPI_Recv` when the corresponding `BDMPI_Wait` operation is invoked.

It can be shown that the above protocol ensures that as long as the program is deadlock-free based on MPI's point-to-point communication semantics, its BDMPI execution will also be deadlock-free. However, since BDMPI uses buffered sends, the reverse is not true. That is, a deadlock-free BDMPI program will not necessarily be a deadlock-free MPI program.

5.4. Collective operations

Depending on the specific collective operation and whether their associated communicator involves processes that span more than one node, BDMPI uses different strategies for implementing the various collective operations that it supports.

The `BDMPI_Barrier` operation is performed as follows. Each calling slave notifies its master that it is entering a barrier operation and then blocks by waiting for a message on its go-queue. At the same time, the master changes the state of that process to cblocked. Each master keeps track of the number of its slaves that have entered the barrier, and when that number is equal to the total number of its slaves in the communicator involved, it then calls `MPI_Barrier` to synchronize with the rest of the nodes involved in the communicator. Once the masters return from that `MPI_Barrier` call, they change the state of all their slaves associated with the communicator to runnable. As discussed earlier, the handling of the interactions between the slaves and their master is done by having the master spawn a different thread for each one of them. Within this framework, all but the last thread involved will exit as soon as they change the state of the calling slave to cblocked. It is the last thread (i.e., the one that will be spawned when all but one slave has entered the barrier) that will execute the `MPI_Barrier`. Thus, `MPI_Barrier` involves a communicator whose size is equal to the number of distinct nodes containing slaves in the underlying BDMPI communicator.

Please cite this article in press as: D. LaSalle, G. Karypis, MPI for Big Data: New tricks for an old dog, Parallel Comput. (2014), http://dx.doi.org/10.1016/j.parco.2014.07.003

The `BDMPI_Bcast` operation is performed using a two-phase protocol. In the first phase, each calling slave notifies its master that it is entering a broadcast operation and then blocks by waiting for a message on its go-queue. If the calling slave is the root of the broadcast, prior to blocking, it also copies the data to the master process' memory. When all local slaves have called the broadcast, the data is broadcast to all the master processors of the nodes involved using `MPI_Bcast`. Upon completion of this operation, the masters change the state of their local slaves to runnable. In the second phase, when a slave resumes execution, it notifies its master that it is ready to receive the data, and gets them via the shared memory.

The `BDMPI_Reduce` and `BDMPI_Allreduce` operations are implemented using a similar protocol, though in this case all slaves send their data to their masters, which perform the reduction operation. Similarly, when all local slaves have called the operation, the reduction across the entire system is performed by calling `MPI_Reduce` on a communicator associated with the nodes involved. Finally, in the second phase of this operation, the destination of the reduction operation (or all slaves in case of `BDMPI_Allreduce`) receives the data from its master via the shared memory.

Note that for the above three operations, the masters store the data involved in memory as opposed to buffering them on disk. The rationale for this is that since the amount of data involved does not increase with the size of the communicator, it does not create excessive memory requirements. Moreover, in order to ensure that this data is not swapped out, BDMPI has an option of locking it in physical memory.

The implementation of the other collective communication operations is different depending on the number of nodes involved. If more than one node is involved, these operations are implemented using `BDMPI_Send` and `BDMPI_Recv` operations or repeated calls to `BDMPI_Bcast` for the case of `BDMPI_Allgather`. If the number of nodes is one (i.e., all slaves in the communicator belong to the same node), the operations are performed using an analogous two-phase protocol, with the appropriate slave-to-master and master-to-slave data movement. The only difference is that based on the size of the data, they are either buffered in the memory of the master, or they are buffered on the disk of the node. This is done for two reasons. First, the amount of data involved is written and read only once, so there is little benefit for storing them in memory. Second, the aggregate amount of data can become very large (especially in the case of the all-to-all operation), which can lead to excessive memory swapping.

BDMPI uses two different states to differentiate between a slave blocked due to a receive operation or a collective communication operation (i.e., rblocked and cblocked). This is necessary for ensuring that a slave blocked on a collective operation does not become runnable because it received a message from another slave.

5.5. Communicator operations

The majority of the information associated with a communicator is maintained by the masters, and the communicator-related information maintained by the slaves is minimal (id, rank, and size). The masters maintain information related to the identity of the slave processes and their location across the nodes. In addition, each BDMPI communicator has an associated MPI communicator containing the set of masters involved, which is used for the MPI operations that the masters need to perform among them. Finally, BDMPI implements the `BDMPI_Comm_split` MPI function, which provides a flexible mechanism to subdivide an existing communicator.

5.6. BDMPI extensions

BDMPI provides a small number of functions that are not part of the MPI standard in order to enable multiple slaves to be running concurrently in a contention-free fashion, facilitate intra-node synchronization, and to allow the program to get information about its execution environment as it relates on how the processes are organized within each node. These functions are shown in Table 2.

The first two functions are used to indicate a section of the program during which only a single slave can be executing within each node. These critical sections are important for operations involving disk access (e.g., performing an `mlock` or file I/O), as it eliminated disk-access contention. Note that these critical sections are only relevant when `nrunning` is greater than one. These functions are implemented using POSIX semaphores.

The remaining functions have to do with extracting information from a communicator. The `_nsize`/`_nrank` functions return the number of nodes (i.e., masters) in the communicator and the rank of the slave's master in that communicator, respectively. The `_lsize`/`_lrank` functions return the number of other slaves residing on the same node as that of the calling slave and its rank, respectively. Finally, the `_rrank` returns the rank of the lowest ranked slave in the same node as that of the calling slave.

Table 2
BDMPI extensions.

`BDMPI_Enter_critical`, `BDMP_Exit_critical`
`BDMPI_Comm_nsize`, `BDMPI_Comm_nrank`, `BDMPI_Comm_lsize`, `BDMPI_Comm_lrank`, `BDMPI_Comm_rrank`

Please cite this article in press as: D. LaSalle, G. Karypis, MPI for Big Data: New tricks for an old dog, Parallel Comput. (2014), http://dx.doi.org/10.1016/j.parco.2014.07.003

BDMPI provides two additional built-in communicators: `BDMPI_COMM_CWORLD` and `BDMPI_COMM_NODE`. The first contains all the slaves across all the nodes numbered in a cyclic fashion, whereas the second contains all the slaves on the same node as that process. The first communicator is provided for programs that can achieve better load balance by splitting the ranks in a cyclic fashion across the nodes. The second communicator is provided so that the program can use it in order to perform parallel I/O at the node level or to create additional communicators that are aware of the two-level topology of the processes involved.

6. Experimental setup

6.1. Benchmark applications

We evaluated the performance of BDMPI using three applications: (i) PageRank on an unweighted graph [22], (ii) spherical K-means clustering of sparse high-dimensional vectors [23], and (iii) matrix factorization using stochastic gradient descent (SGD) for recommender systems [24].

Our MPI implementation of PageRank 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 [22]. Our MPI implementation of K-means uses an identical one-dimensional row-wise decomposition of the sparse matrix to be clustered as the PageRank implementation. The rows of that matrix correspond to the sparse vectors of the objects to be clustered. The K-way clustering starts by randomly selecting one of the processes p_i, which proceeds to select K of its rows as the centroids of the K clusters. Each iteration then proceeds as follows. Process p_i broadcasts the K centroids to all other processes. Processes assign their rows to the closest centroids, compute the new centroids for their local rows, and then determine the new global centroids via a reduction operation. This process terminates when no rows have been reassigned. Our MPI implementation of SGD follows the parallelization approach described in [24] and uses a $\sqrt{p} \times \sqrt{p}$ two-dimensional decomposition of the sparse rating matrix R to be factored into the product of U and V. Each iteration is broken down into $\sqrt{p}$ steps and in the ith step, computation is performed on the blocks along the ith diagonal. This ensures that at any given step, no two processes update the same entries of U and V. Note that in this formulation, at any given time, only $\sqrt{p}$ processes will be active performing SGD computations. Even though this is not acceptable on a p-processor dedicated parallel system, it is fine within the context of BDMPI execution, since multiple MPI processes are mapped on the same node.

For all of the above parallel formulations, we implemented three different variants. The first corresponds to their standard MPI implementations as described above. The second extends these implementations by inserting explicit function calls to lock in physical memory the data that is needed by each process in order to perform its computations and to unlock them when it is done. As a result of the memory locking calls (`mlock`), the OS maps from the swap file into the physical memory pages all the data associated with the address space been locked and any subsequent accesses to that data will not incur any page faults. The third corresponds to an implementation in which the input data and selective intermediate data are explicitly read from and written to the disk prior to and after their use (in the spirit of out-of-core formulations). This implementation was done in order to evaluate the OS overheads associated with swap file handling and demand loading. We will use the *mlock* and *ooc* suffixes to refer to these two alternative versions.

In all of these benchmarks, the input data were replicated to all the nodes of the cluster and the processes took turns in reading their assigned data via BDMPI's execution model. That is, when only one process per node is allowed to execute at a time, only one process per node can read a file at a time. The exception to this is that when four processes were allowed to run at a time, we modified the BDMPI code such that the four processes took turns while reading the input data on each node. As a result, the I/O was parallelized at the node-level and was serialized at the within node slave-level. The output data were sent to the zero rank process, which wrote them to the disk.

We also developed serial out-of-core versions of these algorithms in order to evaluate the performance that can be achieved by programs that have been explicitly optimized for out-of-core processing. We will denote these algorithms by *Serial-ooc*. The out-of-core implementation of PageRank keeps the page rank vectors (*current* and *next*) in memory. During each iteration, the graph is processed in chunks, and a push algorithm (as in our MPI implementation) is used to update the *next* PageRank vector. The out-of-core implementation of K-means keeps the centroids (*current* and *next*) in memory. The matrix and the row cluster assignment vector are read in chunks from the disk during each iteration. Once a chunk of the matrix has been processed (i.e., the new cluster memberships have been determined and the new centroids have been partially updated), the chunk of the cluster assignment vector is written back to disk. The out-of-core implementation of SGD uses a two-dimensional decomposition of the input matrix into chunks. During an iteration, each matrix chunk and corresponding segments of U and V are read from disk and updates are made, before saving the segments of U and V back to disk. Note that we process the chunks in a row-major order, as a result, the part of U corresponding to the current set of rows is read only once (at the start of processing the chunks of that row) and is written back to disk once (after all chunks have been processed).

We used the PageRank and SGD implementations provided by GraphChi 0.2 [17] for comparison on the single-node experiments. For distributed PageRank we used the implementation from Pegasus [14]. For distributed K-means we used the version provided with 0.7 of Mahout [13].

6.2. Datasets

For the PageRank strong scaling experiments we used uk-2007–05 [25] web graph, with 105 million vertices and 3.3 billion edges. We used the undirected version of this graph available as part of the *10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering* [26]. To ensure that the performance of the algorithms is not affected by a favorable ordering of the vertices, we renumbered the vertices of the graph randomly.

For the PageRank weak scaling experiments, we used the com-orkut [27] social network with three million vertices and 117 million edges as the base graph distributed to each slave process. In order to properly evaluate the scaling of BDMPI's intra- and inter-node communication operations, the graphs were re-wired such that one third of the edges are connected to vertices on other slaves within the same node, one third of the edges are connected to vertices on slaves located on adjacent nodes (i.e., node n has edges to nodes $n-1$ and $n+1$), and the final one third of edges are left as is. This ensures that for the weak scaling experiments we have a scalable communication pattern, yet still stress the BDMPI communication operations by sending a large volume of messages.

For the K-means experiments we used a sparse document-term matrix of newspaper articles with 30 million rows and 83 thousand columns containing 7.3 billion non-zeros. For the weak scaling experiments, each slave was assigned a submatrix with 764 thousand rows and 183 million non-zeros.

For the SGD experiments, we used the dataset from the NetFlix Prize [28], replicated 128 times to create an 8 × 16 block matrix, with 3.8 million rows, 284 thousand columns, and 12.8 billion non-zeros. For the SGD weak scaling experiments, we used the first half of the rows of the NetFlix dataset as the base matrix distributed to each slave.

6.3. System configuration

These experiments were run on two dedicated clusters. The first consisted of four Dell Optilex 9010s, each equipped with an Intel Core i7 @ 3.4 GHz processor, 4 GB of memory, and a Seagate Barracuda 7200RPM 1.0 TB hard drive. Because of BDMPI's dependence on the swap-file for data storage, the machines were set up with 300 GB swap partitions. The four machines run the Ubuntu 12.04.2 LTS distribution of the GNU/Linux operating system. For the Hadoop [2] based algorithms, we used version 1.1.2 of Hadoop and OpenJDK IcedTea6 1.12.5.

The second cluster was used for the weak scaling experiments of the PageRank and SGD benchmarks. The cluster consists of 20 compute nodes, each equipped with a Xeon E5–2620 @ 2.0 GHz processor, 4 GB of memory, and a 500 GB spinning hard drive. These machines each have a 64 GB swap file, and run the CentOS distribution of the GNU/Linux operating system.

7. Results

For the three benchmarks we gathered results by performing ten iterations. 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.

7.1. Performance of PageRank

Table 3 shows the performance achieved by the different programs on the PageRank benchmark.

Comparing the performance achieved by the various BDMPI versions, we see that BDMPI-ooc performs the best whereas the BDMPI version (i.e., the version that corresponds to the unmodified MPI implementation executed via BDMPI's system) performs the worst. However, the performance difference between these two implementations is within a factor of two. The performance achieved by BDMPI-mlock is in between the other two versions. These results indicate that there are benefits to be gained by optimizing an MPI code for BDMPI's runtime system and that bypassing the OS's VMM system does lead to performance improvements.

Comparing the results obtained on the four nodes over those obtained on a single node, we can see that most versions of BDMPI achieve super-linear speedups. This 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 suspension/resumption steps.

The performance achieved by the MPI version of the benchmark on a single node is better than that of the first two BDMPI versions, though its performance is worse than that of the BDMPI-ooc version. This result is somewhat surprising, since the single-node execution of the MPI version is nothing more than running the serial algorithm on the graph, and as such it relies entirely on the VMM system. However, this good performance can be attributed to the following two reasons. First, the BDMPI versions have to incur the overhead associated with the all-to-all communication for *pushing* the locally computed contributions of the PageRank vector to the slaves that are responsible for the corresponding vertices. Since the vertices of the input graph are ordered randomly and the single-node BDMPI experiments distribute the computations among twelve slaves, this step involves a non-trivial amount of communication. On the other hand, the single-node MPI experiment does not partition the graph and as such it does not incur that overhead. Second, the number of vertices in the graph is rather small and as a result, the PageRank vector being computed fits in the physical memory. If that vector cannot fit in the physical memory, the performance will degrade substantially. To verify this, we performed an experiment in which we simulated

Please cite this article in press as: D. LaSalle, G. Karypis, MPI for Big Data: New tricks for an old dog, Parallel Comput. (2014), http://dx.doi.org/10.1016/j.parco.2014.07.003

Table 3
PageRank performance.

Algorithm	Num. of nodes = 1	Num. of nodes = 4
BDMPI	19.86	4.34
BDMPI-mlock	15.11	3.89
BDMPI-ooc	9.98	2.35
MPI	14.84	10.25
Serial-ooc	5.43	N/A
GraphChi [8 GB]	45.90	N/A
Pegasus (Hadoop)	N/A	234.93

These results correspond to the number of minutes required to perform a single iteration of PageRank on the UK-2007–05 graph. Using a CSR structure, this graph takes 25.3 GB of memory, which distributed amongst the twelve slave processes is 2.1 GB per process. The single-node BDMPI runs were performed using 12 slave processes and the four-node runs were performed using 3 slave processes per node. All BDMPI experiments were obtained by setting `nrunning` to one. The MPI results were obtained by MPICH using `-np 1` and `-np 4`. Pegasus's PageRank job ran with a total of 1762 maps and eight reduces. The GraphChi results were obtained on a node with 8 GB of DRAM, as it was unable to run on a 4 GB node without swapping.

a graph that has four times the number of vertices. For that graph, the first iteration of the single-node MPI version did not finish after six hours, whereas the time required by a single iteration of BDMPI-ooc took about 47 min using 50 slaves. Also it is interesting to note that the MPI version does not scale well on four nodes, as it achieved a speedup of only 1.45. We believe the primary reason for this is that the MPI version now has to incur the overhead associated with the all-to-all communication discussed earlier (as it decomposes the graph among four nodes), which significantly increases its overall runtime and thus reduces the speedup.

The overall best single-node results were obtained by the Serial-ooc version. This is not surprising as this implementation has been explicitly optimized for out-of-core execution. Comparing the single-node performance of BDMPI against that of Serial-ooc, we see that the performance penalty associated with BDMPI's more general approach for out-of-core computations does incur some extra overheads. However, these overheads are not very significant, as the best BDMPI version is less than two times slower than the optimized serial out-of-core implementation.

Finally, both the GraphChi and the Pegasus versions performed significantly worse than any of the other versions. Compared to BDMPI-ooc, on a single node, GraphChi is 4.6 times slower, whereas on four nodes, Pegasus is 100 times slower. This is because these computational models do not allow the same flexibility as the MPI API, and as a result the implementations require substantially more operations and memory movement.

7.2. Performance of spherical *K*-means

Table 4 shows the results achieved by the different programs on the *K*-means benchmark. This table, in addition to the set of experiments in which the number of running slaves was set to one (i.e., "#R=1") also reports two additional sets of results. The first is for the case in which the maximum number of running slaves was set to four (i.e., "#R=4") and the second is for the case in which we used OpenMP to parallelize the cluster assignment phase of the computations. These results are reported under the "#T=4" columns and were obtained using four threads.

In terms of single-node performance, Serial-ooc performed the best, with BDMPI-ooc less than a minute behind in per-iteration time. BDMPI and BDMPI-mlock were 14% and 43% slower than BDMPI-ooc, respectively. This reversal of BDMPI and BDMPI-mlock's performance from the PageRank benchmark can be explained by the larger amount of computation required by *K*-means, where the overhead of page-faults could be offset. While this trend held for increasing the number of threads used, increasing the number of processes running resulted in BDMPI-mlock running 32% faster than BDMPI. This can be explained by the extra pressure put on the swap by BDMPI when all four running processes are generating page-faults. BDMPI is able to do better using four threads than BDMPI-mlock as it allows for computation and I/O to be overlapped, this is also why for BDMPI-mlock and BDMPI-ooc, four running processes perform better than four threads for intra-node parallelization.[3] The speedups are below the ideal 4× because a significant portion of the runtime is spent saving/loading data to/from the single spinning disk on each node. The use of SSDs or multiple spinning drives per process/thread may result in speedup closer to the ideal.

The close performance between the Serial-ooc version and the various BDMPI versions is due to the fact that unlike the PageRank benchmark, the *K*-means benchmark involves significantly more computations, which reduces the relative cost associated with data loading. Also similar to the PageRank benchmark, the four-node experiments show that BDMPI can achieve very good speedups, which in most cases range from 3.7 to 4.9. Finally, Mahout's Hadoop implementation of *K*-means was several orders of magnitude slower than the other methods we tested.

A notable difference between the *K*-means results and those of PageRank is that the performance achieved by the MPI version was worse than that achieved by all BDMPI versions on both a single node and on four nodes. We believe that

[3] When more than one slave is allowed to run, the corresponding BDMPI versions use the functions described in Section 5.6 to ensure that only one slave is accessing the disk.

Table 4
Spherical *K*-means performance.

Algorithm	#R=1/#T=1	#R=4/#T=1	#R=1/#T=4
Number of nodes = 1			
BDMPI	29.76	24.98	16.76
BDMPI-mlock	37.15	18.86	23.60
BDMPI-ooc	25.97	14.83	15.20
MPI	43.36	N/A	53.13
Serial-ooc	25.82	N/A	N/A
Number of nodes = 4			
BDMPI	7.45	5.75	3.75
BDMPI-mlock	7.59	3.98	4.82
BDMPI-ooc	6.98	4.35	4.14
MPI	18.13	N/A	13.51
Mahout (Hadoop)	N/A	1196.75	N/A

These results correspond to the number of minutes required to perform a single iteration of spherical *k*-means on the news dataset for $K = 100$. In a binary CSR structure the total matrix takes 56 GB of memory. "#R" is the maximum number of slave processes that can run concurrently on a single node. "#T" is the number of OpenMP threads used to perform the computations within each slave process. All BDMPI runs using "#R=1" were performed using 20 slave process, whereas the "#R=4" runs were performed using 80 slave processes. When using 20 slaves, each slave stored 2.8 GB of the matrix, and when using 80 slaves, each slave stored 717 MB of the matrix. Mahout's *K*-means job ran with a total of 1014 maps and one reduce. The MPI results were obtained by MPICH using `-np 1` and `-np 4`.

the reason for that is twofold. Where as a running BDMPI process fits within the available memory and thus its data needs to only be loaded from swap the first time its accessed, the MPI version does not fit within memory, and must migrate data two and from swap multiple times per iteration. This also explains the increase in runtime when using four threads on a single node, as the threads compete to keep their data resident in memory. On four nodes, a larger fraction of the data fits within memory and less pressure is put on the swap, and the runtime is decreased when using four threads per node. Second, *K*-means incurs a lower communication overhead than that of the PageRank algorithm (broadcast/reduction vs all-to-all), which reduces the overhead associated with using the 20 slaves in the single-node BDMPI versions. Also this lower parallel overhead is the reason that the speedup achieved by the MPI version of *K*-means on four nodes is higher than the corresponding speedup achieved on the PageRank benchmark.

In terms of strong scaling, both BDMPI and BDMPI-mlock exhibited super-linear speedups between 4.3 and 4.9 using no intra-node parallelization, four threads, and four running processes per node, as a larger portion of the 56 GB of data fit in the increased aggregate memory of the system (16 GB compared to 4 GB). BDMPI-ooc did not benefit from this as each slave process reads and writes all of its data from and to the disk at resumption and suspension.

7.3. Performance of stochastic gradient descent

Table 5 shows the performance of the SGD benchmark. Results from two versions of SGD are presented. The first one randomly traverses the elements of the matrix, and the second randomly traverses the rows. The row-wise traversal has better data locality and is faster.

Comparing the runtimes of the different BDMPI versions we see that the relative ranking of the three versions mirrors that of the PageRank benchmark. The BDMPI-ooc performs the best, whereas the simple BDMPI version performs the worst. However unlike the other two benchmarks, the performance of the simple BDMPI version is substantially worst than the other two versions. This is due to the randomized traversal of the non-zero elements of the matrix and associated factors, which lead to an essentially random access over the swap file. This poor performance is even worse for both the single and four-node MPI versions, neither of which manage to finish a single iteration in 50 h.

The relatively good performance achieved by BDMPI-mlock and BDMPI-ooc is due to their loading of data into memory before processing, which greatly reduces the latency of the first access to each page of memory. In fact, their single-node performance relative to the Serial-ooc is very competitive, with BDMPI-mlock and BDMPI-ooc requiring at most 66% and 7% more time, respectively.

The speedups achieved by the different BDMPI versions on four nodes are super-linear, which is consistent with similar trends observed on the other benchmarks. As it was the case with the earlier results, this can be attributed to the increase in the aggregate amount of physical memory.

The results for the experiments in which the number of running slaves was set to four ("#R=4") are also consistent with the earlier observations. Because multiple slave processes incur page faults concurrently, the performance of the simple BDMPI version degrades. However, the performance of the other two versions improves, with BDMPI-ooc improving more than BDMPI-mlock. This is because BDMPI-mlock's cost of prefetching the data is higher than that of BDMPI-ooc, and since this step is serialized across the running slaves, it limits the overall speedup that it can obtain.

Finally, the last row of Table 5 shows the performance achieved by GraphChi's SGD implementation. GraphChi's implementation keeps both the user and item factors in memory and also visits the rows of the entire matrix in a random order.

Please cite this article in press as: D. LaSalle, G. Karypis, MPI for Big Data: New tricks for an old dog, Parallel Comput. (2014), http://dx.doi.org/10.1016/j.parco.2014.07.003

Table 5
Stochastic gradient descent performance.

Algorithm	Num. of nodes = 1		Num. of nodes = 4	
	#R=1	#R=4	#R=1	#R=4
Element-wise random traversal				
BDMPI	756.13	2251.67	196.31	562.13
BDMPI-mlock	103.31	68.68	24.40	11.03
BDMPI-ooc	66.77	30.70	16.55	8.93
MPI	> 3000			
Serial-ooc	62.16	N/A	N/A	N/A
Row-wise random traversal				
BDMPI	663.24	2078.83	168.16	545.17
BDMPI-mlock	58.99	54.18	14.25	10.03
BDMPI-ooc	29.83	15.44	7.36	4.03
MPI	> 3000			
Serial-ooc	28.29	N/A	N/A	N/A
GraphChi [8 GB]	59.78	N/A	N/A	N/A

These results correspond to the number of minutes required to perform a single iteration of stochastic gradient descent on the 128 copies of the NetFlix for 20 latent factors. In a binary CSR structure this matrix requires 96.2 GB of memory. In the MPI runs, none of the iterations finished within the allotted time. "#R" is the maximum number of slave processes that can run concurrently on a single node. All BDMPI runs were performed using 256 slave processes in a 16 × 16 configuration, where each slave stores 385 MB of the matrix. For the single node experiments, all these slave processes were mapped on the same node, whereas for the four-node experiments, they were equally distributed among the nodes. The GraphChi results were obtained on a node with 8 GB of DRAM, as it was unable to run on a 4 GB node without swapping.

Table 6
PageRank weak scaling.

	Iter. Runtime (m)				
Num. of nodes	1	5	10	15	20
Mil. of edges	937.4	4687.4	9374.8	14,062.2	18,749.6
BDMPI	2.50	3.30	3.47	3.69	3.77
BDMPI-mlock	3.06	3.93	4.16	4.40	4.56
BDMPI-ooc	2.26	2.68	2.85	2.97	3.17

These results correspond to the number of minutes required to perform a single iteration of PageRank. Each node contains eight slave processes, and each slave process has 3,072,441 vertices and 117,185,083 edges. Each slave has 1.8 GB of data, resulting in an aggregate 14.4 GB of data per node (each node has only 4 GB of DRAM).

This row-wise traversal has better locality than the row-wise traversal used by BDMPI and Serial-ooc versions, as the latter perform the row-wise traversal within each block of the 16 × 16 decomposition of the matrix (traversing 1/16 of each row at a time). Despite these, BDMPI-mlock was 1.3% faster and BDMPI-ooc was 100% faster on a single node.

7.4. Weak scaling

PageRank. Table 6 shows the scaling of the three BDMPI implementations for the PageRank benchmark, where we increase the number of nodes used, while keeping the work per node constant. When running beyond a single node, we see a jump in the per iteration runtime due to the communication of large amounts of data between nodes. As described in Section 5.3, the sending of large messages between nodes in BDMPI requires writing and reading the message to disk twice, once on the local node, and once on the remote node. As one third of the edges are between vertices on different nodes in this experiment, these disk operations account for a large part of the 56% increase on average in the per iteration runtime when running five nodes compared to one.

The 23.3% increase in runtime among the three BDMPI implementations from five nodes to 20 can be attributed to the varying time it took to read/write from/to the disk. As we increased the number of nodes, we increased the likely-hood at each iteration of having to wait for a node exhibiting slow I/O performance.

K-means. Table 7 shows the scaling of the three BDMPI implementations for the *K*-means benchmark, where we increase the number of nodes used while keeping the work per node constant. As with PageRank, we see the time per iteration climb as we increase the number of nodes and the problem size.

Due to the higher ratio of computation to communication in this benchmark, we see superior scaling compared to that of the PageRank benchmark. From one to five nodes, the average iteration time increased by 3.8% across all three versions. From five to 20 nodes, the average iteration time increased by 9.2% across all three versions.

With relatively sequential memory access and high computation to communication ratio, the *K*-means benchmark is an ideal application for BDMPI as shown by its ability scale.

Table 7
K-means weak scaling.

	Iter. Runtime (m)				
Num. of nodes	1	5	10	15	20
Mil. of non-zeros	183.6	734.4	1468.8	2203.3	2937.7
BDMPI	6.66	6.86	6.91	6.95	7.04
BDMPI-mlock	7.68	8.12	8.25	8.40	8.45
BDMPI-ooc	6.45	6.63	7.15	7.49	8.08

These results correspond to the number of minutes required to perform a single iteration of *K*-Means. Each node contains eight slave processes, and each slave process has 183,609,600 non-zeros of the sparse matrix. Each slave has 1.4 GB of data, resulting in an aggregate 11.2 GB of data per node (each node has only 4 GB of DRAM).

Table 8
Stochastic gradient descent weak scaling.

	Iter. Runtime (m)		
Num. of Nodes	1	4	16
Mil. of non-zeros	803.2	3212.9	12,851.7
BDMPI	35.32	38.53	49.13
BDMPI-mlock	6.17	6.59	6.92
BDMPI-ooc	4.64	4.80	4.94

These results correspond to the number of minutes required to perform a single iteration of stochastic gradient descent using element-wise random traversal for 20 latent factors. Each node has 16 slave processes and each slave has 50,201,924 non-zeros of the sparse matrix. Each slave has 440 MB of data, resulting in an aggregate 6.88 GB of data per node (each node has only 4 GB of DRAM).

SGD. Table 8 shows the scaling of the three BDMPI implementations for the SGD benchmark, where we increase the number of nodes used while keeping the work per node constant. For SGD, we only use one, four, and 16 nodes due to our $\sqrt{p} \times \sqrt{p}$ decomposition as explained in Section 6.1. As with the weak scaling of the PageRank benchmark, we see an immediate increase in runtime as the cost cross-node communication is added. This increase is much larger for BDMPI and BDMPI-mlock as the communication structures of the BDMPI runtime and underlying MPI runtime may not be resident in memory at the start of a communication operation. This is most pronounced for the BDMPI-mlock version, in which running on four nodes we have a 27% increase in runtime. However, going from four nodes to 16, we only have a 4.2% increase in runtime.

For the BDMPI version of the SGD benchmark, a large increase in runtime exists when running on 16 nodes instead of 4. The random-element access pattern of this benchmark causes significant page thrashing and increased iteration runtime variability. As we increase the problem size and the number of nodes used, more nodes must wait idly at the end of each iteration. When using 16 nodes, we observed variations in per-iteration runtime as high as 5.8 min.

BDMPI-oc, as it does not rely on swap, does not suffer from its communication structures possibly not being resident in memory, and its runtime only increases by 1.7% when run on four nodes, and 3.0% when running on 16 nodes.

8. Conclusion

The current options for developers today looking to process Big Data on commodity clusters or workstations forces them to choose between undertaking a heroic engineering effort and sacrificing portability, or attempting to fit a new computational paradigm to their problem which in many cases can mean sacrificing performance and using a non-intuitive formulation. Our solution to this problem, BDMPI, fills this gap by providing developers seeking performance from their Big Data applications an extremely flexible framework. By using the existing MPI API, we ensure not only that the wide range of problems MPI has been used to express can also be expressed in BDMPI, but we also leverage the existing knowledge and experience that has been gained over the past twenty years since its introduction. Moreover, our experiments showed that BDMPI offers performance close to that of direct out-of-core implementations and provides significant performance gains over existing technologies such as Hadoop and GraphChi.

Acknowledgments

This work was supported in part by NSF (IOS-0820730, IIS-0905220, OCI-1048018, CNS-1162405, and IIS-1247632) 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.

References

[1] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[2] ApacheTM Hadoop. <http://hadoop.apache.org>.
[3] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, 2010, pp. 10–10.
[4] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM, 2010, pp. 135–146.
[5] S. Seo, E.J. Yoon, J. Kim, S. Jin, J.-S. Kim, S. Maeng, Hama: an efficient matrix computation with the mapreduce framework, in: IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2010, pp. 721–726.
[6] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein, Graphlab: a new parallel framework for machine learning, in: Conference on Uncertainty in Artificial Intelligence (UAI), Catalina Island, California, 2010.
[7] ApacheTMGiraph. <http://giraph.apache.org..
[8] MPI: a message-passing interface standard version 3.0, 2012. <www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf>.
[9] J.S. Vitter, External memory algorithms and data structures: dealing with massive data, ACM Comput. Surv. (CsUR) 33 (2) (2001) 209–271.
[10] R. Bordawekar, A. Choudhary, Communication strategies for out-of-core programs on distributed memory machines, in: Proceedings of the 9th International Conference on Supercomputing, ACM, 1995, pp. 395–403.
[11] S. Toledo, A survey of out-of-core algorithms in numerical linear algebra, External Memory Algorithms Visual. 50 (1999) 161–179.
[12] M.F. Pace, BSP vs MapReduce, Proc. Comput. Sci. 9 (2012) 246–255.
[13] ApacheTMMahout. <http://mahout.apache.org/>.
[14] U. Kang, C.E. Tsourakakis, C. Faloutsos, Pegasus: a peta-scale graph mining system implementation and observations, in: Ninth IEEE International Conference on Data Mining, ICDM'09, IEEE, 2009, pp. 229–238.
[15] Y. Bu, B. Howe, M. Balazinska, M.D. Ernst, HaLoop: efficient iterative data processing on large clusters, Proc. VLDB Endow. 3 (1–2) (2010) 285–296.
[16] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[17] A. Kyrola, G. Blelloch, C. Guestrin, Graphchi: large-scale graph computation on just a pc, in: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2012, pp. 31–46.
[18] L. Kale, Parallel programming with charm: an overview, Parallel Programming Laboratory, University of Illinois at Urbana-Champaign, Tech. Rep.
[19] M. Potnuru, Automatic out-of-core exceution support for charm++ (Master's thesis), University of Illinois at Urbana-Champaign, 2003.
[20] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing, Benjamin/Cummings, Redwood City, 1994.
[21] R. Rabenseifner, G. Hager, G. Jost, Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes, in: 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, IEEE, 2009, pp. 427–436.
[22] L. Page, S. Brin, R. Motwani, T. Winograd, The PageRank citation ranking: bringing order to the web.
[23] I.S. Dhillon, D.S. Modha, Concept decompositions for large sparse text data using clustering, Mach. learn. 42 (1–2) (2001) 143–175.
[24] B. Recht, C. Re, Parallel stochastic gradient algorithms for large-scale matrix completion, Math. Prog. Comput. 5 (2) (2013) 201–226. <http://dx.doi.org/10.1007/s12532-013-0053-8>.
[25] P. Boldi, M. Rosa, M. Santini, 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.
[26] D.A. Bader, H. Meyerhenke, P. Sanders, D. Wagner (Eds.), Graph Partitioning and Graph Clustering – 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13–14, 2012, Proceedings, Contemporary Mathematics, vol. 588, American Mathematical Society, 2013.
[27] J. Yang, J. Leskovec, Defining and evaluating network communities based on ground-truth, in: Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics, ACM, 2012, p. 3.
[28] J. Bennett, S. Lanning, The netflix prize, in: Proceedings of KDD cup and workshop, vol. 2007, 2007, p. 35.

Please cite this article in press as: D. LaSalle, G. Karypis, MPI for Big Data: New tricks for an old dog, Parallel Comput. (2014), http://dx.doi.org/10.1016/j.parco.2014.07.003