BDMPI: Conquering BigData with Small Clusters using MPI

Dominique LaSalle
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, Minnesota 55455, USA
lasalle@cs.umn.edu

George Karypis
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, Minnesota 55455, USA
karypis@cs.umn.edu

ABSTRACT

The problem of processing 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 disk-based execution of distributed-memory parallel programs. This system, called BDMPI, 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 parallel distributed memory disk-based 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 (GraphChi) as well as on a small cluster (Hadoop).

Keywords

BigData, MPI, out-of-core, parallel processing

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 BigData, in order to gain scientific insights, increase situational awareness, improve services, and generate economic value. The amount of data coupled with the complexity of the analysis, makes approaches that rely on distributed computing and keep only a subset of the data in memory, the only scalable paradigm for developing BigData analysis applications.

DISCS-2013 November 18, 2013, Denver, CO, USA.

ACM 978-1-4503-2506-6/13/11 ...$15.00.
http://dx.doi.org/10.1145/2534645.2534652.

Low-level libraries like MPI [3] provide a flexible message passing interface that allows the development of complex high performance codes, and despite their engineering costs, have been used to develop a vast array of scientific applications. Higher level specialized message passing frameworks have been developed such as Pregel [12] and GraphLab [11]. These all follow the BSP model proposed by Valiant [16], and are meant to lower the engineering cost of developing large scale graph processing codes.

Similarly, many low-level codes have been developed to make use of disk space for performing computation on data too large to fit in DRAM. Although efficient, these approaches require a non-trivial engineering effort and are often application specific [6, 15]. High-level frameworks have also been proposed to utilize disk to handle massive problem sizes. GraphChi [10] is capable of performing graph based computations in a disk-based fashion, but does not currently support execution in a distributed environment.

A high-level framework that provides both distributed execution and disk-based computations is the tremendously popular MapReduce [7] and its open source implementation Hadoop [1]. MapReduce scales to very large clusters and extremely large datasets, especially for computations that can be performed by a single scan through the data. However, its ability to efficiently express and execute iterative computations is limited, as it results in unnecessary data movement.

The objective of this work is to provide a disk-based distributed computing framework that can achieve high performance while simultaneously be flexible enough to allow the expression of computations required by a wide-range of applications. The current lack of such tools represents an important problem because analysts seeking to operate on large datasets must either undertake a significant engineering effort of building distributed disk-based applications, or settle for the operational inefficiency of poorly fitted frameworks. This significantly impedes the broad application of BigData analysis approaches and their ability to scale to very large datasets.

The key insight underlying our approach is the observation that scalable distributed memory parallel algorithms (e.g., those written in MPI) tend to exhibit three characteristics. First, they follow a BSP computational model and perform as many local computations as possible before communicating with other processing elements. Second, these computations often involve the entire set of data that each processing element is assigned to, or a well-defined non-trivial subset of that data. Third, given a fixed problem size, the amount of

data that each processing element needs to operate on (i.e., the data assigned to each processing element), decreases as the number of processing elements increases. As a result, the memory footprint that each processing element requires for solving a given problem can be decreased by simply increasing the number of processing elements used to perform the computations.

Relying on these observations, we developed an approach for achieving efficient BigData analysis by coupling scalable distributed memory parallel programs written in MPI with an intelligent runtime system that amortizes the cost of loading data from the disk over the maximal amount of computations that can be performed. Given an MPI application, our approach solves a problem of a certain size by (i) executing the application using a sufficiently large number of processes so that to ensure that each process fits within the physical memory available on each compute node, (ii) mapping multiple processes to each of the available nodes, and (iii) coordinating the execution of these processes using a node-level co-operative multi-tasking approach that amortizes the cost of loading data from disk over the maximal amount of computations that can be performed. BDMPI's runtime exploits the natural blocking points that exist in MPI programs to co-operatively schedule the execution of the different processes and the associated data loading.

We experimentally evaluated the performance of BDMPI on three problems (K-means, PageRank, and stocastic gradient descent). Our experiments show that BDMPI programs lead to substantial performance improvements over existing approaches. 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.

2. OVERVIEW OF BDMPI

BDMPI is implemented as a layer between an MPI program and any of the existing implementations of MPI, and whenever appropriate, it relies on disk-based message buffering. BDMPI uses virtual memory and the OS's swapping mechanism to provide the programmer with the illusion that the parallel program is operating on all the data in memory. However, BDMPI can also be used for expressing distributed disk-based computations, in which case the loading/storing of the data from/to disk will be done explicitly by the programmer.

A BDMPI application is simply 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. The optional parameter `nrunning` specifies the maximum number of slave processes that can be running at any given time on a node. If this parameter is not specified, the default value is one. 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` $\times$ `nslaves` processes, each process executing `progname`. In this environment, these processes will make up the `MPI_COMM_WORLD` communicator, and as such from the application's point of view, it is equivalent to performing:

```
mpiexec -np nnodes*nslaves
        progname [arg1] [arg2] ...
```

BDMPI uses two key elements in order to enable the efficient execution of BigData analysis applications. 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*.

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. The programmer can rely on the system's swapping mechanism for memory migration to and from disk, or explicitly read/write the bulk of the processes' memory to/from disk at possible blocking/resumption points.

Coupling constrained memory over-subscription with node-level co-operative multi-tasking is the key idea 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, because each node has sufficient amount of physical memory to accommodate all the processes that are allowed to run, to prevent memory thrashing (i.e., repeated and frequent page faults).

BDMPI dramatically lowers the burden of developing parallel distributed memory disk-based programs by allowing programmers to only use the message passing interface described in the next section. This also increases the portability of programs, because when the memory of a system is sufficient, the program will execute as normal MPI program, and the same program can be executed seamlessly on systems with different disk configurations.

3. 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 3.0 that BDMPI implements, which needs to be linked with the application code. The subset of the MPI 3.0 specification that is currently implemented in BDMPI is shown in Table 1. This contains a

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_Allgather, BDMPI_Reduce, BDMPI_Allreduce, BDMPI_Gather[v], BDMPI_Scatter[v], BDMPI_Alltoall[v]`

reasonable set of MPI functions for developing a wide-range of message passing programs. We plan to expand this set of functions by including additional MPI functions related to prefix scan operations and topology management.

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. Our implementation of BDMPI can take advantage of MPI's `PMPI_` prefix to provide an API that is a direct drop-in replacement of the standard MPI API.

3.1 Master and Slave Processes

The execution of a BDMPI application 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 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 application 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 process 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 each slave process calls the `BDMPI_Finalize` function, the master process removes them from the set of slave processes involved in co-operative multi-tasking, and their execution resumes to follow the regular pre-emptive multi-tasking.

All communication/synchronization operations between slave processes go via their corresponding 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.

3.2 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 process blocks by waiting for a message on its go-queue and the master process signals that a process can resume execution by sending a message to the go-queue of that process. 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.

A slave process can be in one of the following states: *running* (a process is currently running), *rblocked* (a process is blocked due to an MPI receive operation), *cblocked* (a process is blocked due to an MPI collective operation), *runnable* (a process can be scheduled for execution if resources are available), and *finalized* (a process has called `BDMPI_Finalize`). The blocking/resumption of the slave processes is done jointly by the implementation of the MPI functions in `bdmpilib` and the master process. If a slave process calls an MPI function that leads to a blocking condition (more on that later), it notifies the master process and then blocks by waiting for a message on its go-queue. The master process updates the state of the slave process to rblocked/cblocked and proceeds to select another runnable process to resume its execution by sending a message to its go-queue. When a slave process 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 process is at a runnable state, the master selects for resumption the process that has the highest fraction of its virtual memory already mapped on the physical memory. This is done so that to minimize the cost of establishing memory residency of the resumed process.

3.3 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. This approach 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. In case of disk-based buffering, the message is written to the disk by the slave process, and the name of the file used to store it is communicated to the master process. If the destination slave is on a different node, then the master process of the sender notifies that master process of the destination and sends the data to it via an `MPI_Send` operation. The receiving master process 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 process via POSIX shared memory and stored locally or sent to the remote slave's master node. In any of these cases, the master process 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 application. If the corresponding send operation has not been posted, then the slave blocks by waiting on its go-queue. In that case, the master process also changes the state of that slave from running to rblocked. When a slave resumes execution (because its master process 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 master processes do not maintain information about the posted receive operations but instead only maintain information about the send operations.

3.4 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 process that is entering a barrier operation and then blocks by waiting for a message on its go-queue. At the same time, the master process changes the state of that process to cblocked. Each master process 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 master processes return from that `MPI_Barrier` call, they change the state of all their slave processes 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 process 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 called when all but one slave has not 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.

The `BDMPI_Bcast` operation is performed using a two-phase protocol. In the first phase, each calling slave notifies its master process that 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 process resumes execution, it notifies the master that is ready to receive the data, and it gets the data via the shared memory. A similar protocol is used for implementing the `BDMPI_Reduce` and `BDMPI_Allreduce` operations.

The implementation of the other collective communication operations is different depending on the number of nodes involved, and the details are omitted here due to space constraints.

4. EXPERIMENTAL SETUP

4.1 Benchmark Applications and their Implementations

We evaluated the performance of BDMPI using three applications: (i) PageRank on an unweighted directed graph (`PR`) [13], (ii) spherical K-means clustering of sparse high-dimensional vectors (`SphKMeans`) [8], and (iii) matrix factorization using stochastic gradient descent (SGD) for recommender systems (`MF`) [14].

Our MPI implementation of `PR` 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]. Our MPI implementation of `SphKMeans` uses an identical one-dimensional row-wise decomposition of the sparse matrix to be clustered as the `PR` 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 and having it randomly 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 centroids, compute the new centroids for their local rows, and then global centroids are calculated via reduction. This process terminates when no rows have been reassigned. Our MPI implementation of `MF` follows the parallelization approach described in [14] 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. To ensure conflicting updates to U and V are not made, the only active blocks at any given time will be those on the diagonal (and shifted diagonal) so that no two active blocks will have the same rows or columns. 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 it 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. 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.

For all applications, the input data were replicated to all the nodes of the cluster and the processes took turn in reading their assigned data. 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 used the PageRank and stocastic gradient descent implementations provided by GraphChi 0.2 [10] for comparison on the single-node experiments. For distributed PageRank we used the implementation from Pegasus 2.0 [9]. For distributed K-means we used the version provided with 0.7 of Mahout [2].

4.2 Datasets

For these experiments we used three different datasets. For the PageRank experiments we used an undirected version of the uk-2007-05 [5] web graph, with 105 million vertices and 3.3 billion edges. 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 stocastic gradient descent experiments, we used the dataset from the NetFlix Prize [4], replicated 40 times. The resulting sparse matrix has 19 million rows, 18 thousand columns, and 4 billion non-zero entries.

4.3 System Configuration

These experiments were run on a dedicated cluster consisting of four machines, each equipped with an Intel Core i7 @ 3.4GHz processor, 4GB of memory, and a Seagate Barracuda 7200RPM 1.0TB hard drive, running version 3.5.0-23 of the Linux kernel. Each machine has 300GB swap and scratch partitions. For the Hadoop [1] based algorithms, we used version 1.1.2 with data replication disabled.

5. RESULTS

For the three problems we gathered results on (PageRank, K-means, and stocastic gradient descent) we ran 10 iterations and present the average time per iteration below. This time includes the load and storing of input, output, and intermediate data. That is, the total execution time divided by the number of iterations. We terminated some of the runs before they finished all 10 iterations due to high runtime, and are marked with a "$*$" in the results where the average of fewer than 10 iterations was taken. We generated 100 clusters for the K-means problem and 20 factors for stocastic gradient descent.

Table 2: Performance on a Single Node

Algorithm	Implementation	# Slaves	Time (min.)
PageRank	PR	40	34.74
	PR	64	31.97
	PR-mlock	40	28.59
	PR-mlock	64	33.41
	PR-ooc	40	19.00
	PR-ooc	64	24.16
	GraphChi	-	288.87
KMeans	SphKMeans	20	39.55
	SphKMeans-mlock	20	36.55
	SphKMeans-ooc	20	25.53
SGD	MF*	16	948.78
	MF	64	303.55
	MF-mlock	16	41.51
	MF-mlock	64	46.75
	MF-ooc	16	33.32
	MF-ooc	64	33.99
SGD-Row	MF1DR-mlock	16	17.23
	MF1DR-ooc	16	9.17
	MF2DR-mlock	16	24.06
	MF2DR-ooc	16	16.54
	GraphChi	-	31.78

5.1 Single Node

Table 2 shows the average runtime per iteration of the various implementations of the three presented problems on a single compute node. The `ooc` versions that explicitly manage their memory performed the best for each application. The `mlock` versions which implicitly manage their memory, were not far behind in terms of performance, taking 37% longer on average. The versions that use no implicit or explicit memory management, performed the slowest, taking 54% longer than the `ooc` versions on average, with the exception of the `MF` application. This shows that for many applications, performance can still be achieved with no memory management.

The random access of data elements in stocastic gradient descent differentiates it from the other two problems. Because each page gets swapped in individually as a single element on it is accessed, `MF` has extremely high runtimes. The `MF-mlock` and `MF-ooc` versions avoid this issue by loading all of the required pages of memory before computation starts. In this situation, memory management is vital to achieving acceptable performance. However, when memory is explicitly managed as in the `ooc` version or even implicitly managed as in the `mlock` version, this random access of data was not an issue.

Compared to GraphChi, BDMPI on a single node performs quite well. For the PageRank problem, the BDMPI versions were 8 to 15 times faster. Because the stocastic gradient descent implemented by GraphChi randomly traverses the rows of the matrix rather the individual non-zero values, it achieves a higher level of data locality. We refer to this as SGD-Row. To make an accurate comparison, we also implemented SGD-Row in `MF1DR` and `MF2DR`, which implement one-dimensional and two-dimensional decompositions of the matrix among the processes respectively. The im-

Table 3: Performance on Four Nodes

Algorithm	Implementation	# Slaves	Time (min.)
PageRank	PR	40	9.53
	PR	64	6.38
	PR-mlock	40	7.36
	PR-mlock	64	7.29
	PR-ooc	40	3.91
	PR-ooc	64	4.60
	Pegasus	-	234.93
KMeans	SphKMeans	20	9.65
	SphKMeans-mlock	20	8.30
	SphKMeans-ooc	20	6.71
	Mahout	-	1196.75
SGD	MF*	16	202.34
	MF*	64	69.24
	MF-mlock	16	10.14
	MF-mlock	64	10.86
	MF-ooc	16	8.04
	MF-ooc	64	7.67

proved data locality makes a significant difference, and the most competitive version, `MF1DR-ooc` was 3.5 times faster than the GraphChi implementation. This shows the robustness of BDMPI and the value of a light-weight solution.

5.2 Multi-Node

Table 3 shows the average runtime per iteration of the various implementations of the three applications on a four node compute cluster. All three versions of the three problems scaled to four nodes quite well, averaging a super-linear speedup of 4.2. These super-linear speedups exhibited for the BDMPI codes are due to the larger amount of aggregate memory available on four nodes compared to one (16GB vs. 4GB). The version of `MF` without memory management suffered from similar locality issues as it did running a single node. However, compared with those times, it averaged a speedup of 4.54. The only version that did not see a super-linear speedup, was that of `SphKMeans-ooc`, which had a speedup of 3.8.

The Hadoop implementations of PageRank and K-means were significantly slower than their BDMPI counter parts. The slowest BDMPI variant of PageRank which used 64 slaves and had no memory management, ran almost 25 times faster per iteration than the Pegasus implementation, and the fastest variant with explicit memory management and 40 slaves ran 60 times faster per iteration. Similarly, Mahout took several orders of magnitude longer than the BDMPI versions of K-means, with the version without memory management running 124 times faster, the `mlock` version running 144 times faster, and the `ooc` version running 178 times faster.

For both single node and multi-node runs, the `ooc` versions typically performed 48% (excluding the `MF` versions without memory management) better than those relying on the swap mechanism. This shows the cost of such mechanisms is non-trivial for moving large amounts of data, but still small enough to make relying on them a viable option in most cases. This gives developers a choice between pursuing maximum performance and minimizing (re-)engineering efforts.

6. CONCLUSION

In this paper we have presented BDMPI, a runtime system that provides functionality for message passing and disk based computations using MPI semantics. Using this system, we have demonstrated that high performance distributed disk-based codes can be developed without a heroic engineering effort. By utilizing the pre-existing semantics and flexibility of MPI, we leverage the knowledge and experience of the high performance computing community. The performance of the presented benchmarks significantly outperforms that of the competing Hadoop based implementations when run in a distributed manner, and even outperforms those implementations which use explicitly disk based frameworks.

7. REFERENCES

[1] Apache Hadoop®. `http://hadoop.apache.org`.

[2] Apache Mahout. `http://mahout.apache.org/`.

[3] MPI: A Message-Passing Interface Standard Version 3.0. `www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, 2012.

[4] James Bennett and Stan Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[5] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano 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.

[6] Rajesh Bordawekar and Alok Choudhary. Communication strategies for out-of-core programs on distributed memory machines. In *Proceedings of the 9th international conference on Supercomputing*, pages 395–403. ACM, 1995.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] Inderjit S Dhillon and Dharmendra S Modha. Concept decompositions for large sparse text data using clustering. *Machine learning*, 42(1-2):143–175, 2001.

[9] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[10] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[11] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[12] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz 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.

[13] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[14] Benjamin Recht and Christopher Ré. Parallel stochastic gradient algorithms for large-scale matrix completion. *Mathematical Programming Computation*, 5(2):201–226, 2013.

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

[16] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.