# An Analysis of Parallel Programming Techniques for Data Intensive Computation

**Anuradharthi Thiruvenkata Ramani and John A. Chandy**
Department of Electrical and Computer Engineering, University of Connecticut, Storrs, CT, USA

**Abstract**— *Data intensive applications have too much data to analyze quickly and in its entirety. The ability to extract valuable information in real time remains a intimidating challenge. Efficient distributed parallel programming techniques are the key in achieving the performance requirements for such computations. The most common cluster data parallel programming methodology is to use message passing to communicate between computation nodes. Recently, the MapReduce model has been introduced as a more efficient mechanism to enable easy development of parallel applications to process large amounts of data on large clusters. In this paper we analyze and compare the performance of MapReduce and message passing in general and specifically for two data intensive computations: (a) WordCount and (b) Blast Like Alignment Tool (BLAT).*

## 1. Introduction

Computer science today is rapidly moving towards parallelism as a means to prevail against increasing problem sizes and the declining rate of clock speed improvements. Parallel programming developed as a means of improving performance and efficiency. In parallel programming, the computation is divided into parts, each of which can be executed concurrently. The instructions from each part have the ability to run simultaneously on different CPUs that can exist on a single machine, or they can be on a set of computers connected through network.

In order to process large amounts of data, the computations have to be distributed across multiple machines in order to finish in a reasonable amount of time. The issues parallelizing the computation, distribute the data and handle failures collude to complicate the original simple computation with large amounts of complex code to deal with these issues. Not only are parallel programs faster, they can also be used to solve problems on large datasets using non-local resources. When you have a set of computers connected on a network, you have a vast pool of CPUs, and you often have the ability to read and write very large files with a distributed file system. As a reaction of complexity, choosing between different distributed parallel programming techniques to accommodate exponentially growing data with a finite amount of time to do the required analysis becomes difficult. The major contribution of this work is to analyze the two major distributed cluster parallel programming techniques - namely MapReduce and message passing for data intensive applications.

Section 2 describes in detail the two parallel computing techniques with the architecture overview. Section 3 details the data intensive applications. In Section 4, we describe the distributed application programming analysis and partitioning techniques for the large data sets. Section 5 has the performance measurements of MapReduce and Message Passing for the chosen applications along with the timing analysis between the two programming techniques.

## 2. Parallel Computing

The evolution of parallel computer architectures has recently created new trends and challenges for both parallel application developers and end users. Parallel computing is a form of computation in which many calculations are performed simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (in parallel). Parallel Computing are categorized by the level of parallelism employed with multi-core and multi-processor comprised of many processing units within a computer, while clusters, massively parallel processors (MPP), and grids use multiple computers to work on the same task.

Parallel computer programs are more difficult to write than sequential ones, because the simultaneous executions introduces several new potential hazards like data dependencies which can lead to the race conditions and other software bugs. Understanding data dependencies is fundamental in implementing parallel algorithms. No program can run more quickly than the longest chain of dependent calculations (known as the critical path), since calculations that depend upon prior calculations in the chain must be executed in order. However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel. Communication and synchronization between the different subtasks is typically one of the greatest barriers to getting good parallel program performance. The speed-up of a program as a result of parallelization is given by Amdahl's law.

### 2.1 Message Passing

Message passing is the most commonly used paradigm for cluster-based parallel programming. The Message Passing

Interface (MPI) [1], [2] is the standard used to code parallel computing applications based on this paradigm. Derived Datatypes, Dynamic process management, Point-to-point and collective communication are supported by MPI. The MPI interface was developed with the intent of comprehending all of the available message-passing constructs and features of various MPP and networked clusters so that programs would execute on any type of system. MPI's portability feature supports copying, compiling and execution of a program written for one architecture on another without modification. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

Single Program Multiple data(SPMD) is one of the ways to achieve parallel computing where a single program is split across a group of processes to execute concurrently with localized data. MPI is designed to support SPMD model. So when developing an application to be executed on a single MPP, then MPI has the advantage of higher communication performance and the ever desired ease of portability. MPI has a much richer set of communication functions which makes it advantageous to use on an application with special communication mode requirements.

MPI's [3], [4] implementation language in general is different from the language it supports at runtime. Most MPI implementations are a combination of C, C++ and assembly language, and target C, C++, and Fortran programmers. MPI-CH and MPI-Ram are libraries that have implemented MPI functions in C and C++. However, Java programs cannot use these libraries directly, for this reason mpiJava was designed to bridge Java applications to the underlying MPI-CH or MPI-Ram functions. To achieve the promised high communication performance, some of the notable forfeits were the lack of interoperability between two MPI implementations, and lack of ability to write a fault tolerant application. The only guarantee MPI specification provides is ability to quit the program on error. The mpiJava interface is designed to provide essential virtual topology, synchronization, and communication functionality between a set of computers in a language-independent and language-specific way. To achieve the desired maximum performance, each CPU is assigned only a single process during runtime.

However, even mpiJava has not been able to handle and recover faults automatically. mpiJava has static nature, wherein, the processes are created during the initialization and assigned process numbers, every process keeps details of every other processes, new processes cannot be generated during runtime. This static nature of mpiJava favors in achieving the higher performance. The main features include the checkpointing done at frequent intervals and restarting a compute node on failure. This can be quite inefficient when involving a larger system. When the mpiJava tasks reside on the same data node, the communication can be over the switch network or through shared memory concept. But when there are large amounts of data that has to be communicated over the network it results in a significant amount of I/O traffic sometimes leading to Network Contention.

## 2.2 MapReduce

MapReduce [5], [6] is a programming paradigm and framework developed by Google for simplified, parallel data processing on clusters built of commodity hardware. As its name implies, it was inspired by the Lisp (list-processing) functions reduce, also known as accumulate, and map. However, the MapReduce model behaves slightly differently when it comes to all of the input, output, and intermediate data which are expressed as key/value pairs. The MapReduce infrastructure is designed to handle partitioning and distribution of input key/value pair data across a cluster. The Model also employs a namenode and jobtracker to distribute and execute the task across all of the available nodes. Each of the mapper function is assigned a input split it operates on and produces the intermediate key/value pairs. The combiner does the process of shuffling and sorting all the intermediate results. Finally these sorted keys are assigned to the reducer, which produces the output key/value pairs.

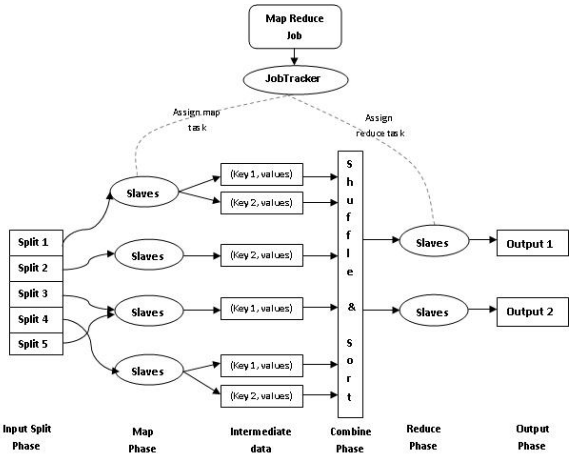


Figure 1: MapReduce Execution Overview

The advantage of MapReduce is the ability to distribute and process map and reduce operations. The only concern is that the MapReduce job should have the capability of splitting the job and be executing them in parallel. There might be some limitation to the map job arising due to data availability and location of CPU's near the data. Similarly, the reduce job works on the results from the mapper stored in intermediate local disks. This technique works well with large data sets. In addition to parallelism it also provides a fairly good failure recovery. Due to the replication of available data, even if a worker node fails, the job could be rescheduled to another node where the data is replicated. In MapReduce [8], [7] model, each of the node has local hard drives where the intermediate files are stored. These files

can be remotely read by the reducers. Thus MapReduce is a shared-nothing architecture. But all the MapReduce jobs read from and write data to HDFS [9] which is shared by all the nodes. HDFS takes care of the replication placing them on compute nodes around the cluster based on the specified replication factor. Due to this shared-nothing architecture scalability can be easily achieved, and the integrated HDFS makes thousands of disks appear like one.

**a) Execution Overview:** Figure 1 illustrates the MapReduce Execution Overview.

The major functions in MapReduce's dataflow is detailed below:

1) Input Reader : The Map/Reduce framework relies on the InputFormat of the job to validate them, split the input file(s) into logical InputSplit instances of size 16MB to 64MB, each of which is then assigned to an individual Mapper, and provide RecordReader to collect the input record from logical split and present a record-oriented view to be processed by the Mapper. Typically the input reader reads data from HDFS and generates key/value pairs.
2) Mapper : Map function, specified by the user program, takes an input split and produces a set of intermediate key/value pairs. The intermediate key/value pairs are stored on the local disks which are remotely read by the Reducer function. The input and output types of the Map function are often different.
3) Combiner : The Combiner function, specified by the user program, remotely reads the intermediate key/value pairs processed by different Map jobs located on the local disks. It then shuffles and sorts the key/value pairs based on the intermediate key.
4) Reducer : The Reduce function, also specified by the user program, accepts an intermediate key and a set of values for that key from the combiner function. It then merges the set of values in to a smaller set of values. A reducer always receives key/value pairs as input and produces key/value pairs as output. For large datasets, the reducer may perform the reduce action in multiple steps.
5) Output Writer : The Output Writer writes the output of the reduce to stable storage, usually the Hadoop distributed file system.
6) HDFS: The Hadoop Distributed File System (HDFS) [9], [10] is a distributed file system designed to run on commodity hardware. Figure 2 depicts the architecture. There are many similarities with the existing distributes file system whereas the notable differences include high fault tolerance and ability to be built from commodity hardware. HDFS is designed to support large datasets, provide high data bandwidth and scale to thousands to nodes.

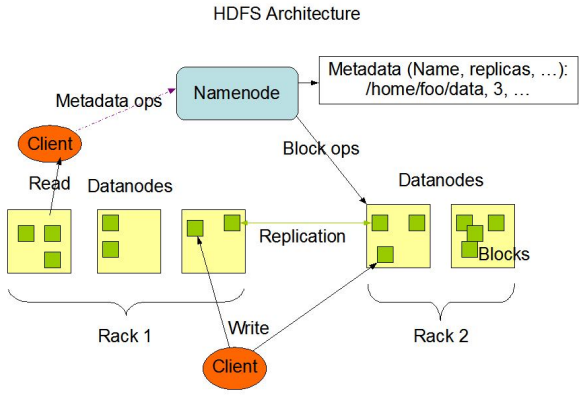


Figure 2: HDFS Architecture [9]

HDFS applications are designed to fall under the model of write-once-read-many. So a files that are written once and are accessed multiple times result in high throughput data access. So the parallel computing on large data sets could be one among them. Also when the computation is carried out near data the network congestion could be reduced resulting in increased throughput of the system. HDFS provides interface to move application close to data. HDFS is designed to be portable on different platforms. HDFS uses the master/slave architecture. HDFS has a single master, NameNode, which keeps track of all the slaves, and manages the namespace. There are many slaves, the DataNode which carry out the process instructed by the master node. The input file(s) is usually split across the available datanodes. The mapping of the blocks on the DataNodes is taken care of by the NameNode. NameNode creates, deletes and renames the file blocks on the DataNode. DataNode also performs these operations when instructed by the NameNode.

## 3. Data Intensive Computation

Data Intensive Computing [11], [12], [13] is acquiring, analyzing, and processing volumes of data. The ever advancing technology provides the ability to handle these large data sets with economical storage, and faster bandwidth. These data sets evolve from a variety of applications ranging from scientific research(e.g., bioinformatics, climate change), transactional data (e.g., payroll, accounting) and environment(e.g., ozone layer, water acoustics). The challenge of these large amount of data is to extract valuable information in real-time. This problem could be

1) accommodate ever-increasing data
2) process and obtain valuable information
3) convert information into human readable data

Data intensive computing requires a well structured set of principles. Many data-intensive applications can be parallelized to achieve the required output specification. The data-

intensive applications demand reliability, high fault tolerance, real-time response and availability.

In this paper, we analyze two data intensive applications - the simple Word Count application and the much more complicated BLAT application. Word count is derived from a Hadoop application and has been ported to MPI and BLAT was an initial MPI application that has been ported to Hadoop.

### 3.1 Word Count

WordCount is an application that counts how often a word occurs, the total number of words, lines and characters in a text file. Knowing the numbers of words is sometime important, for instance when we are required to stay certain minimum or maximum bounds. The input is text files and the output is text files, each line of which contains a word and the count of how often it occurred, separated by a tab.

### 3.2 BLAT

The second data intensive application to be considered is BLAT (BLAST Like Alignment Tool) [14], [15]. BLAT is one of the most widely used bioinformatics tool. This is commonly used to look up the location of a sequence in the genome or determine the exon structure of an mRNA. BLAT is an alignment tool like BLAST, but it is structured differently. On DNA, BLAT works by keeping an index of an entire genome in memory. Thus, the target database of BLAT is not a set of GenBank sequences, but instead an index derived from the assembly of the entire genome. The index – which uses less than a gigabyte of RAM – consists of all non-overlapping 11-mers except for those heavily involved in repeats. This smaller size means that BLAT is far more easily mirrored. BLAT of DNA is designed to quickly find sequences of 95% and greater similarity of length 40 bases or more. It may miss more divergent or short sequence alignments. BLAT on proteins finds sequences of 80% and greater similarity of length 20 amino acids or more.

The protein index requires slightly more than 2 gigabytes of RAM. In practice – due to sequence divergence rates over evolutionary time – DNA BLAT works well within humans and primates, while protein BLAT continues to find good matches within terrestrial vertebrates and even earlier organisms for conserved proteins. Within humans, protein BLAT gives a much better picture of gene families (paralogs) than DNA BLAT. The genome itself is not kept in memory, allowing BLAT to deliver high performance on a reasonably priced Linux box. The index is used to find areas of probable homology, which are then loaded into memory for a detailed alignment. Protein BLAT works in a similar manner, except with 4-mers rather than 11-mers. The protein index takes a little more than 2 gigabytes. The output of BLAT is flexible. By default it is a simple tab-delimited file which describes the alignment, but which does not include the sequence of the alignment itself. Building an index of the genome typically takes 10 to 15 minutes.

The input file format for the BLAT application is the FASTA format. In bioinformatics, the FASTA format is a text-based format for representing either nucleic acid sequences or peptide sequences, in which base pairs or amino acids are represented using single-letter codes. The format also allows for sequence names and comments to precede the sequences. The simplicity of FASTA format makes it easy to manipulate and parse sequences using text-processing tools. Both the database and query input files are FASTA format. When BLAT is run on a genome set, it is analogous to a database being queried. The working of the application is similar to the grep application which does the query matching on the available database. The resulting output would be the sequences names, matched sequence position, and matching length from the database FASTA file.

### 3.3 Other Examples

As parallel computing become larger and faster, it becomes feasible to solve problems that previously took too long to run. Parallel computing is used in a wide range of fields, from bioinformatics (to do protein folding) to economics (to do simulation in mathematical finance). Some common examples that can be considered applicable to the distributed parallel programming techniques are Count of URL Access Frequency, Matrix Multiplication, Inverted Index, Distributed Sort, and Graph traversal.

## 4. Distributed Application Programming Analysis

The criteria to choose among the available data-intensive computing technique are the amount of data involved, the computing capabilities and the underlying storage. The distributed parallel programming techniques should have high-level parallelism, easier programming, high reliability, higher performance, and economical.

Table 1 summarizes the basic characteristics comparison between MPI and MapReduce.

Table 1: Comparison of MPI and MapReduce characteristics

| Characteristics | MPI | MapReduce |
|---|---|---|
| Synchronization | Message passing and barrier options | Map jobs followed by Reduce jobs |
| Communication | Fine-Grained | Coarse-grained |
| Partitioning | Performed by developer | By HDFS and specified partitioning technique |
| Fault Tolerance | Sensitive to failing components and relies on applications to tolerate failure | Reschedules the job on a node failure |
| Prog. Languages | Supports C, C++, Java and Fortran | Supports Java, Hadoop Streaming supports C++ |

## 4.1 Comparison of MapReduce and Message Passing programming

This section analyzes the data intensive applications which we implemented using Hadoop-MapReduce and mpiJava. It also describes the challenges we faced in implementing them.

### 4.1.1 Programming Styles

Although MapReduce supports streaming and pipes, currently there is no support for using applications written in C language. BLAT is a bioinformatics genome tool written in C. So in order to use this application under the MapReduce model, the solution we came up with this is to write a SWIG wrapper functions for the map and reduce tasks in Java, and use these wrapper functions to execute the BLAT genome searching. The input to the map function, database and the query files were placed in the HDFS. The map wrapper function reads the database file, and according to the specified fileinput format creates the input split and stores them in the specified list of datanodes. Each map task will process one of these input splits and the resulting intermediate output will have the query matching done at the inputsplit received by the map function.

This intermediate output is saved in the HDFS. Hadoop's runtime collects these locations and sends them to the appropriate reduce tasks. The reduce tasks now have the partial output. The reduce tasks performs the operation of sorting the received output and merging them. The merged output file is then stored in HDFS and the location is passed to the user program, which then performs a similar operation to merge all of them in to a single output file. When programming for MapReduce the main concentration was on to getting the core application done rather than data partitioning, memory management, communication between jobs, synchronization parallelizing the code. All of these was taken care of by the Hadoop framework.

The technique adopted to implement the mpiJava version was the similar master/slave model. Message passing is well-suited to handling computations where a task is divided up into subtasks, with most of the processes used to compute the subtasks and a few processes (often just one process) managing the tasks. The manager is called the "master" and the others the "workers" or the "slaves". When implementing using mpiJava programming required careful tuning. There was a considerable time that has to be spent for synchronization, data partitioning and parallelizing the code. But by considering parallel libraries in different parallel threads and avoiding overlapping of communication and computation, we could obtain maximum performance-oriented standard, trading off hardware latency and hardware bandwidth.

## 4.2 Partitioning Techniques

An important step in testing the programming is developing useful and comprehensive test data. Traditionally, we rely on partitioning to get parallelism in I/O, and to reduce network traffic. Partitioning is a way to divide a large data sets into separately divided chunks; the contents of each chunk being determined by the partitioning technique. The multiple pieces of a large data sets give the data administrator a lot of flexibility and manageability in dealing with them. This also can boost performance.

### 4.2.1 Load Balancing

Load balancing partitions the data as near to equal size as possible, ensuring an even load across your all the processors. The boundaries are identified by determining the total size of the data and the actual number of the resulting partitions. This is the most common type of partitioning and is often useful for applications that manage historical data, especially data warehouses.

### 4.2.2 Intelligent Partitioning

For the WordCount application, Load Balancing would suite to be best for consideration. As each of the worker would have nearly equal sized input file to work on. But when considering the BLAT query computation, this might not be work well. The large data set when partitioned equally may sometime result in splitting the existing query match sequence. When this happens we might not get this existing match in the output due to the improper partitioning employed. So for this type of critical conditions we came up with a solution where we would determine the boundary of the equal split. And from the succeeding partition replicate the beginning of the partition equal to the query length and append to the previous partition. Thus the resulting partitions size would be split length plus the query length.

# 5. Quantitative Analysis

Once the distributed parallel programming technique, data-intensive application and partitioning techniques has been chosen the next step is to evaluate the performance.

## 5.1 WordCount

For the WordCount application, we evaluate the overall performance of MapReduce and mpiJava implementations by measuring the total execution time as the number of computing nodes increases. For the WordCount application, the input file sizes were chosen to be 90MB and 5.4GB. The result is tabulated in Table 2. From Figure 3 we observe that MapReduce performs fairly well for large input data sets, but when we evaluate the performance of MapReduce and mpiJava for the smaller data file we observe that there is a considerable overhead in MapReduce. Startup overhead is caused by the propagation of program to all worker nodes. This also involves the formatting of namespace in the NameNode, copying of the input data file to the HDFS and to notify the map nodes about the job assigned. When we

analyzed how the actual time was spent during a WordCount application run on a single compute node using MapReduce technique, it turned out that the startup overhead took about 10 sec of the total computation time. The actual Map job took about 49 sec and the shuffling, sorting and reducing took 2.9 sec. However, when the same job was carried out on increased compute nodes, the overhead startup time was dominated by the reduced MapReduce computation, thus resulting in a performance similar to that of the mpiJava. Thus the startup overhead was nearly eliminated when we had 8 compute nodes.

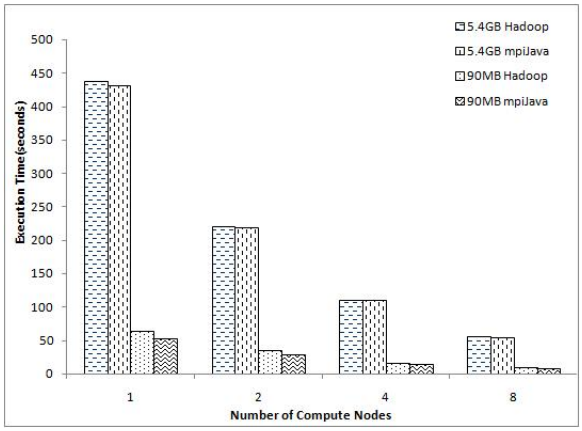


Figure 3: WordCount - execution time vs. number of compute nodes - 90MB and 5.4GB input file

Table 2: WordCount execution time for MapReduce and mpiJava for different input file sizes

| | 90MB Filesize | | 5.4GB Filesize | |
|---|---|---|---|---|
| Compute Nodes | MapReduce | mpiJava | MapReduce | mpiJava |
| 1 | 62.8 | 52.1 | 436.9 | 431.2 |
| 2 | 33.7 | 27.6 | 219.5 | 218.5 |
| 4 | 15.8 | 13.2 | 109.2 | 108.7 |
| 8 | 7.91 | 7.21 | 54.6 | 54.1 |

## 5.2 BLAT

To evaluate the MapReduce and mpiJava technique for BLAT we first measured the total execution time it takes to process data by increasing the amount of data. The number of compute nodes were fixed at 8. This shows that as the data grows the total execution time it takes with MapReduce seems to be getting better than the mpiJava. Figure 4 depicts the result.

We performed another benchmark to see how MapReduce and mpiJava implementations scale as the number of computing nodes increases. For the BLAT application, the input file sizes were fixed as 938 MB and 1.9 GB. This was run on a cluster of 8 computing nodes, increasing in order of 1, 2, 4 and 8. The results are tabulated in Table 3. As we increase the nodes, the number of map tasks were also increased so

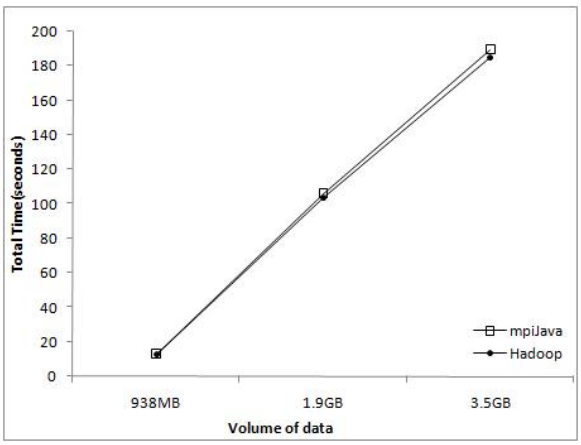


Figure 4: BLAT, Execution time vs. Volume of data on 8 compute nodes

that each of the map tasks processes almost equal amount of data in every run.

Table 3: BLAT execution time for MapReduce and mpiJava for different input file sizes

| | 938MB Filesize | | 1.9GB File size | |
|---|---|---|---|---|
| Compute Nodes | MapReduce | mpiJava | MapReduce | mpiJava |
| 1 | 102.31 | 100.3 | 844.4 | 842.7 |
| 2 | 53.80 | 49.30 | 329.2 | 326.1 |
| 4 | 26.08 | 26.01 | 214.1 | 213.6 |
| 8 | 13.10 | 12.98 | 106.3 | 106.1 |

MapReduce and mpiJava both show similar performance. The overhead induced by the MapReduce implementations has negligible effect on the overall computation as the amount of data accessed in each analysis is large. Figure 5 highlights the scalability of the MapReduce technique. This also indicates how the speedup is achieved with the increase in number of compute nodes.

# 6. Conclusion

In this paper, we have presented our analysis of distributed parallel programming techniques MapReduce and mpiJava for data intensive computing applications. BLAT represents a data intensive computation that can be implemented in MapReduce to gain scalability and efficiency. We have used our implementations to analyze up to 2GB of data. WordCount illustrates that MapReduce can be slower due to the introduced startup overhead for smaller datasets. However, this overhead becomes negligible as the input data size grows. We analyzed the distributed programming techniques with 5.4GB of data.

Comparing the performance of the two data intensive applications using MapReduce and mpiJava led to the following observations:

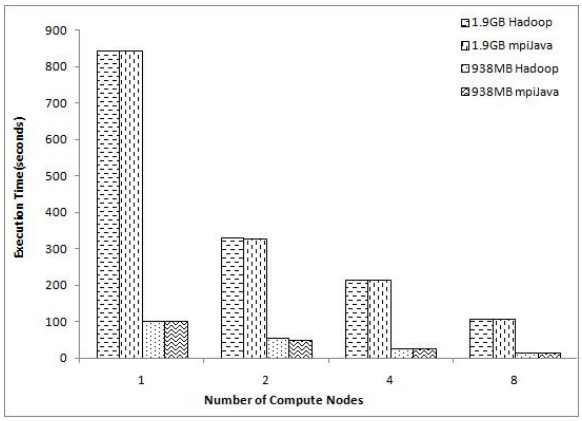


Figure 5: BLAT - execution time vs. number of compute nodes - 1.9GB and 938MB input file

- Most data-intensive computations can benefit from the MapReduce technique to achieve speedup and scalability.
- As the amount of data and computation increases the startup overhead induced by MapReduce diminishes.
- MapReduce provides valuable fault tolerance. Even if a data node fails, due to the replication feature, the computation could be successfully completed. However, in mpiJava when a link is broken the whole computation will have to be rerun again.
- MapReduce provides coarse-grained parallelism. Therefore, the I/O or network access does not create a bottleneck.
- MPI is very sensitive to failing components - thus, programming requires careful tuning.

Our experience shows that mpiJava could be used directly for data intensive computations. However, we strongly believe that some features such as fault tolerance, ease of parallelization, usage of known programming languages, easier scalability, great flexibility in placement, and scheduling exhibited by MapReduce could be effectively used by most of data intensive computations which fall under the MapReduce model.

# 7. Acknowledgment

This work was supported in part by the National Science Foundation High End Computing University Research Activity program under award number CCF-0621448.

# References

[1] *Message Passing Interface Forum. MPI: A Message Passing Interface Standard, June 1995.* `http://www-unix.mcs.anl.gov/mpi/`.

[2] *Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, July 1997.* `http://www.mpi-forum.org`.

[3] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, Timothy S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation", *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pp. 97–104, September.

[4] I. Foster, N. Karonis, "A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems", *Proc. 1998 SC Conference*, November, 1998.

[5] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, pp. 107-113, January 2008.

[6] *Apache Hadoop* `http://hadoop.apache.org/core/`.

[7] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 13-24, 2007.

[8] Matei Zaharia and Andy Konwinski and Anthony D. Joseph and Randy Katz and Ion Stoica, "Improving MapReduce Performance in Heterogeneous Environments", *Proceedings of the Tenth Symposium on Operating System Design and Implementation*, 2008.

[9] *HDFS Architecture*, `http://hadoop.apache.org/core/docs/current/hdfs_design.html`.

[10] Sanjay Ghemawat and Howard Gobioff and Shun-Tak Leung, "The Google file system", *Proceedings of the 19th ACM Symposium on Operating System Principles*, pp. 29-43, October 2003.

[11] Pacific NorthWest National Laboratory, *Data Intensive Computing*. `http://dicomputing.pnl.gov/`.

[12] Randy E Bryant, Carnegie Mellon University, *Data-Intensive Scalable Computing*. `http://research.yahoo.com/files/BryantDISC.pdf`.

[13] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox, "MapReduce for Data Intensive Scientific Analyses," *4th IEEE International Conference on e-Science*, pp.277-284, 2008.

[14] W. James Kent, *BLAT Source code*. `http://www.soe.ucsc.edu/~kent/src/`.

[15] W. James Kent, "BLAT - The BLAST-Like Alignment Tool," *Genome Research 2002*, vol. 12, pp. 656-664, March 2002.