APPLICATION OF MAP/REDUCE PARADIGM IN SUPERCOMPUTING SYSTEMS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Gündüz Vehbi Demirci

August, 2013

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Cevdet Aykanat (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Hakan Ferhatosmanoğlu

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Enis Çetin

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

APPLICATION OF MAP/REDUCE PARADIGM IN SUPERCOMPUTING SYSTEMS

Gündüz Vehbi Demirci
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
August, 2013

Map/Reduce is a framework first introduced by Google in order to rapidly develop big data analytic applications on distributed computing systems. Even though the Map/Reduce paradigm had a game changing impact on certain fields of computer science such as information retrieval and data mining, it did not have such an impact on the scientific computing domain yet. The current implementations of Map/Reduce are especially designed for commodity PC clusters, where failures of compute nodes are common and inter-processor communication is slow. However, scientific computing applications are usually executed on high performance computing (HPC) systems and such systems provide high communication bandwidth with low message latency where failures of processors are rare. Therefore, Map/Reduce framework causes performance degradation and becomes less preferable in scientific computing domain. Due to these reasons, specific implementations of Map/Reduce paradigm are needed for scientific computing domain. Among the existing implementations, we focus our attention on the MapReduce-MPI (MR-MPI) library developed at Sandia National Labs. In this thesis, we argue that by utilizing MR-MPI Library, the Map/Reduce programming paradigm can be successfully utilized for scientific computing applications that require scalability and performance. We tested MR-MPI Library in HPC systems with several fundamental algorithms that are frequently used in scientific computing and data mining domains. Implemented algorithms include all-pair-similarity-search (APSS), all-pair-shortest-path (APSP), and page-rank (PR). Tests were performed on well-known large-scale HPC systems IBM BlueGene/Q (Juqueen) and Cray XE6 (Hermit) to examine scalability and speedup of these algorithms.

Keywords: Map/Reduce, Big Data, Data mining, Information Retrieval, Distributed Computing Systems.

ÖZET

MAP/REDUCE PARADİGMASININ SÜPER BİLGİSAYAR SİSTEMLERİNDE UYGULANMASİ

Gündüz Vehbi Demirci
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr.Cevdet Aykanat
Ağustos, 2013

Map/Reduce, büyük veri uygulamalarının hızlı bir şekilde geliştirilebilmesi için ilk kez Google tarafından ortaya atılan bir uygulama çatısıdır. Map/Reduce paradigmasının, bilgisayar bilimlerinin veri madenciliği, bilgi sistemleri gibi alanlarında büyük etkisi olmasına rağmen, bilimsel hesaplama alanına böyle bir etkisi olmamıştır. Mevcut Map/Reduce uygulamaları özellikle hata oranı yüksek olan ve iletişim hızı düşük olan dağıtık bellekli bilgisayar kümeler için geliştirilmiştir. Bununla birlikte, bilimsel hesaplama uygulamaları genellikle yüksek performanslı bilgisayar sistemleri üzerinde çalıştırılmaktadır ve bu sistemler yüksek bant genişlikli ve düşük gecikmeli iletişim sağlarlar ve bu sistemlerde hata oranı azdır. Bu yüzden, Map/Reduce paradigması bilimsel hesaplama alanında performans azalmasına neden olmaktadır ve bu yüzden daha az tercih edilmektedir. Bu nedenlerden dolayı, bilimsel hesaplama uygulamaları için özel Map/Reduce uygulamaları gerekmektedir. Mevcut olan uygulamalar arasından biz dikkatimizi Sandia Ulusal laboratuvarları tarafından geliştirilen MapReduce-MPI (MR-MPI) kütüphanesi üzerine odakladık. Bu tezde, MR-MPI kütüphanesinden faydalanarak Map/Reduce paradigmasının ölçeklenebilirlik ve performans gerektiren bilimsel hesaplama alanında da kullanılabilecegini savunduk. MR-MPI kütüphanesini bilimsel hesaplama ve veri madenciliğinde sıklıkla kullanılan algoritmalarla yüksek performanslı bilgisayar sistemlerinde test ettik. Tatbik ettiğimiz algoritmalar arasında APSS, APSP, ve PR algoritmalrı vardır. Bu algoritmaların ölçeklenebilirliğini ve hızlanmasını incelemek için yaptığımız testler IBM BlueGene/Q (Juqueen) ve Cray XE6 (Hermit) sistemlerinde gerçekleştirildi.

Anahtar sözcükler: Map/Reduce, Büyük Veri, Veri Madenciliği, Bilgi Sistemleri, Dağıtık Bilgisayar Sistemleri.

Acknowledgement

I would like to express my gratitude to Prof. Dr. Cevdet Aykanat for his supervision and guidance during the study of this thesis. I would like to thank my thesis committee members Assoc. Prof. Dr. Hakan Ferhatosmanoğlu, Prof. Dr. Enis Çetin for reading my thesis and providing valuable comments. I received generous support from Ata Turk during the development of this thesis. Finally, I would like to thank my family for their support and trust in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Big data applications that require processing of huge amount of data are of great importance due to the need of contemporary computation problems. These kinds of problems frequently occur especially in the fields of data mining, bioinformatics and scientific computing. To meet the demands of such applications, Map/Reduce programming framework was first introduced by Google and it became a standard way of developing such applications [1]. Usage of this paradigm is becoming more widespread due to its several other advantages.

The Map/Reduce paradigm originated from functional programming, where higher order functions map and reduce are applied to a list of elements to return a value. In addition, this framework provides a runtime system that manages mapper and reducer tasks, providing automatic scalability, fault tolerance, and auto-parallelization. With the help of this framework, it is possible to ignore complex parallel programming structures like message passing and synchronization and the programmer only needs to design a mapper and a reducer function for each distinct map/reduce phase. Along with reducing programming complexity, another important feature of Map/Reduce is that it can operate on massive data sets. That is, Map/Reduce is designed for scalability instead of speedup. Depending on the architecture it is meant to run, Map/Reduce framework can be implemented in many different ways. For example, one implementation may take advantage of shared or distributed memory architectures and another may

take advantage of larger collection of networked machines [1].

Even though the Map/Reduce paradigm had a game changing impact on certain fields of computer science such as information retrieval and data mining, it did not have such an impact on the scientific computing domain yet. Main reason for this is the current implementations of the Map/Reduce, which are especially designed for commodity PC clusters where failures of compute nodes are common and communication interconnection between nodes is slow. On the other hand, scientific computing applications are usually executed on high performance computing (HPC) systems and such systems provide high communication bandwidth with low message latency. Also, failures of compute nodes on these systems are rare. Therefore, the Map/Reduce framework, which is designed for automated fault-tolerance and thus causes some performance decrease becomes less preferrable in scientific computing domain. Because of these reasons, special implementations of Map/Reduce paradigm is needed for scientific computing domain [2, 3, 4]. Among these implementations, we decided to focus attention on the MapReduce-MPI (MR-MPI) library [5, 2] developed at Sandia National Labs. This is due to the following properties of MR-MPI; it provides a lightweight Map/Reduce implementation developed in C++ and it uses the MPI library for inter-process communication, which enables MR-MPI to be used on HPC platforms without an extra overhead because MPI is well optimized for such systems.

In this thesis, we argue that utilizing MR-MPI Library; the Map/Reduce programming paradigm can be successfully adopted for scientific computing applications that require scalability and performance. HPC systems generally lack virtual memory at compute nodes, because the only external memory available to the nodes is a parallel files system which is accessed by all compute nodes concurrently. This memory bound on each compute node prevents applications to scale to huge data sizes that exceed the total aggregate memory available on the system. Usage of this library helps to deal with virtual memory problems existing on such computing systems. More importantly, Map/Reduce paradigm also provides ease of parallel programming, and it needs programmer to only provide map and reduce functions, hiding parallel programming complexity.

We tested MR-MPI Library in HPC systems with several fundamental algorithms that are frequently used in scientific computing and data mining domains. Implemented algorithms include all-pair-similarity-search (APSS), all-pair-shortest-path (APSP), and page-rank (PR). Tests were performed to see scalability and speedup of these algorithms. We used Juqeen [6] and Hermit [7] HPC systems in our test. The Juqeen system is an IBM BlueGene/Q machine whereas Hermit system is a Cray XE6 machine. These systems are all distributed memory systems and data storage network is separated from compute nodes. Also, message passing between compute nodes are performed via MPI Library which is well optimized for these kinds of systems.

During our tests we generally preferred to use realistic data sets which represent social events or link structure of the web. Additionally, in some cases we also created synthetic data sets. For instance, we created randomly generated graphs that are recursively generated with power-law degree distributions for APSP tests. These type of graphs are commonly used to represent social networks [2].

The remaining parts of this thesis are organized as follows: Chapter 2 gives detailed background information about Map/Reduce paradigm and execution frameworks. Chapter 3 explains the algorithms that we implemented, Chapter 4 presents the details of the MR-MPI implementation of the selected applications. Finally Chapter 5 provides the experimental results and Chapter 6 concludes.

Chapter 2

Background

Map/Reduce framework is first introduced by Google [1]. It is designed to ease parallel programming for distributed computing systems. In addition, Map/Reduce framework also provides scalability, availability, and fault-tolerance for the computing systems that is used by Google. To work properly, the framework needs a distributed file system or data store that can fulfill requirements of the framework. Therefore, Google uses the Map/Reduce framework on top of Google File System (GFS) [8] or BigTable [9] services. Since Map/Reduce framework is not publicly available, open-source implementations of such system are emerged [10, 11]. One of the most famous implementations of the Map/Reduce framework is Hadoop. It runs on Hadoop Distributed File System (HDFS) [12] and designed for commodity PC architectures. In this thesis, both Google and Hadoop Map/Reduce implementations will be referred to explain the framework and runtime system.

2.1 Map/Reduce Programming Paradigm

Map/Reduce is a programming model which is originated from functional programming languages such as Lisp and ML [13]. Among the important features of such languages are higher order functions in which a function can take another

function as an argument. In this model, a higher order function `map` is provided with a user defined function f which is then applied to given list of elements. After this operation has been performed, higher order `reduce` function with another user defined function g, applies the function g to the list of elements which are produced by the previous map operation. Illustration of these operations are given in Figure 2.1.

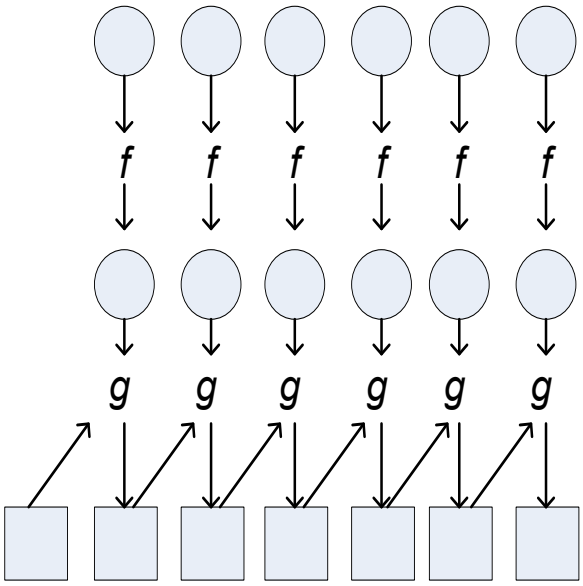

Figure 2.1: Illustration of map and reduce higher order functions: map takes user defined f and reduce takes g, both apply user defined functions to list of elements and reduce aggregates the results.

Application of the above-mentioned map and reduce functions to each element of the input list can be performed separately, which means each element can be processed independently. Using this fact, it is possible to distribute the input elements to different processors and apply user defined functions on them without performing any communication. In this sense, one can exploit functional programming paradigm in order to provide auto-parallelization in parallel computing environments. Map/Reduce basically uses this idea and uses a map function to apply the user defined f to the input data. Input data is stored across a distributed file system running on large number of compute nodes. Distributed file systems store the data by partitioning it into chunks where the size of chunks

is pre-determined. Each chunk is generally replicated to more than one node and each node can serve these chunks to clients. Knowing the fact that the data is partitioned into chunks and chunks are distributed across the nodes, map task invocations are executed on each node in order to process the whole data stored in the system. Map invocations produce intermediate key-value pairs, which are then written back to the distributed file system. This step often causes performance decrease in iterative algorithms that needs more than one Map/Reduce job phase; because writing huge amount of data to the distributed file system causes high latency between the iterations. At this time of execution, different compute nodes may have different key-value pairs with same key field.

Before executing the reduce phase, all key-value pairs are hashed according to their key fields and the range of values, which are produced by the hash function, is divided into the number of reducer tasks. Each reducer task reads intermediate key-value pairs from the distributed file system according to the assignment of hash values to the reducers. Since key-value pairs with the same key can be stored on different nodes, reducer task invocations need to read these key-value pairs from remote compute nodes. To improve performance, run time scheduler of the framework tries to assign reducer tasks to compute nodes, which are close to other nodes that reducer task needs to communicate. In other words, runtime scheduler tries to improve locality during the communication phase. Following this step, key-value pairs with the same key, which are all read and stored locally, are merged into key-multi-value objects. These two steps that are performed after the map phase are called "distributed shuffle" and "sort" in Hadoop implementation of the Map/Reduce framework. Having all the key-multi-value objects, reducer function applies a user defined function to all key-multi-value objects one-by-one and produces the final key-value objects as a result. Later on, these key-value objects are all written back to the distributed file system again and can be used for further Map/Reduce iterations.

2.2 Distributed File Systems

Distributed file systems are well studied in the computer science literature [14, 15, 16, 17]. They differ in their design according to architectures they are designed for and requirements of the domain to be used for. In HPC systems, data is stored on separate network of data nodes, which are only used for data storage. Besides, computation is performed on a network of compute nodes and storing large amounts of data at these nodes is not generally possible. Therefore, link between data nodes and compute nodes can be a bottleneck in data intensive applications. On the other hand, commodity PC clusters do not have that separation between data nodes and compute nodes. All nodes are generally identical and have the ability to store data and perform computations on it. Additionally, component failures are the norm for commodity PC clusters [1], whereas this is not the case for HPC systems. Therefore, a Map/Reduce framework that is designed for commodity PC clusters is powerful when it is backed with a scalable, available and fault tolerant distributed file systems. In addition, since the data that is to be processed by Map/Reduce framework is gigantic, it must be stored on a distributed file system running on large number of compute nodes; because generally it is not possible to fit even the partitioned input data to the memory of a single machine.

The first Map/Reduce implementation, which was introduced by Google, runs on top of Google File System (GFS) [8], which is itself a distributed file system. This system is designed for commodity PC clusters, where failures of nodes are common. Moreover these kind of clusters are not capable of providing low latency messaging, but it is possible to achieve high bandwidth in these systems by using batched messages that have large message sizes. For this reason, GFS uses large chunk sizes when compared to other distributed file systems.

Illustration of GFS system [1] is provided in Figure 2.2 and working principles of the GFS are as follows. GFS consists of a master node and chunk servers. Master node keeps meta data information such as name space, chunk to server mappings, and location of replicas. Chunk servers are actual nodes, where data is stored and served. All chunks are saved to local disk available at the nodes

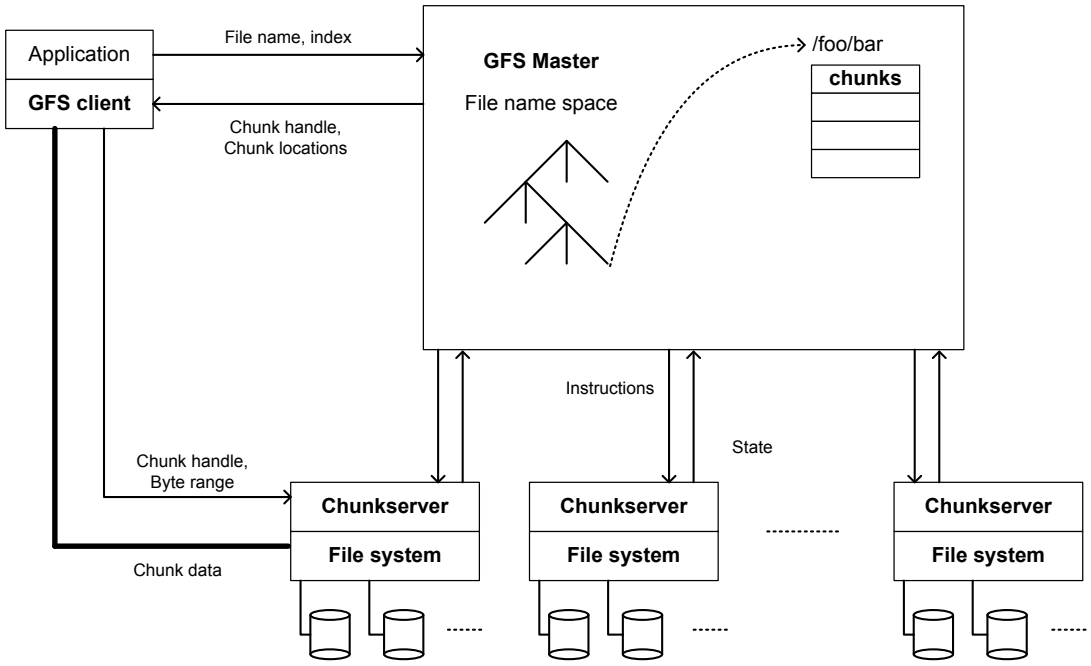

Figure 2.2: Illustration of GFS architecture

using linux file system. In addition, these chunks are replicated to other chunk servers which can be at different racks to provide availability and fault-tolerance. Clients using GFS, transfer data only with chunk servers minimizing participation of master node in order to prevent master node being a bottleneck. Moreover, Map/Reduce task invocations are scheduled to run on the same compute nodes as chunk server deamons in order to provide locality for certain operations. A map task reads chunks residing at the same compute node and produces intermediate key-value pairs, which are later written back to the distributed file system. Reduce tasks are also scheduled to run at the same nodes as GFS deamons. These tasks read some data locally but some chunks may reside at remote nodes. Therefore, these chunks are gathered via messaging protocols available at the cluster infrastructure.

As mentioned earlier, Hadoop framework is an open-source implementation of the Map/Reduce framework. The framework is developed with Java programming language by an open-source software development community Apache Foundation [18]. Hadoop implementation of Map/Reduce runs on top of Hadoop distributed file system. Architecture of the Hadoop Distributed File System (HDFS)

is very similar to GFS and provides almost same functionalities.

HPC systems have also distributed file systems, but these systems generally run on a different network of machines than the machines used solely for computation. This separation is useful especially to scale the computation network; but on the other hand, the link between computation and data storage networks can be a bottleneck in data intensive applications. Obviously, reading and writing operations performed on huge chunks of data that is stored across distributed file systems in HPC platforms is not a good strategy to provide performance and scalability in certain applications. Therefore, MR-MPI library, which is a MPI-based Map/Reduce framework developed for HPC systems, does not write intermediate key-value pairs to the distributed file system. Instead, it uses MPI Library to communicate these key-value pairs among compute nodes using the high speed interconnection network available. Interaction between compute nodes and data nodes occur only when it is not possible to fit intermediate key-value pairs to the internal memory available at compute nodes. This problem is handled by copying these key-value pairs to the distributed file system using I/O operations provided by the system. By this way, avoiding high latency between map and reduce steps is possible and key-value pairs are distributed to reducer processors much faster. These properties of the MR-MPI library differentiate it from other Map/Reduce implementations and enable usage of this library effectively in highly iterative algorithms that run on HPC systems.

Lustre [19] and GPFS [16] distributed file systems are the most commonly used distributed file systems in HPC systems. They run on a separate network of machines for data storing and serving purposes. In Lustre [19], the system consists of three main components which are meta-data server (MDS), object storage servers (OSS) and clients. The meta-data server keeps file names, permissions, directories, and file layout. MDS only involves in pathname and permission operations. All I/O operations such as block allocation, reading, and writing are performed with directly OSSes avoiding MDS being a bottleneck. Whenever clients need to access some portion of a file, offset calculations are carried out on the client with logical object volume layer. Therefore, clients only communicate with OSSes while performing I/O operations, where MDS only controls file

access and informs clients about layout of the objects that constitute the files. Additionally, Lustre uses distributed lock manager in order to protect integrity of files and meta-data information.

Another commonly used distributed file system in HPC systems is General Parallel File System (GPFS) [16] which is developed by IBM. GPFS is shared-disk file system for large clusters and has a different architecture then GFS, Hadoop, and Lustre. Firstly, the GPFS uses shared-disk architecture which also provides high scalability. The system consists of a cluster of nodes and a disk subsystem. These components are connected thorough a switching fabric and files are stripped across all the disks available in the system. Distributed locking is used to synchronize parallel I/O operations that are performed by multiple nodes. Secondly, GPFS fully supports Posix file semantics, whereas previously mentioned file systems are only capable of providing some subset of Posix semantics. Additionally, meta-data information too is distributed across shared-disk file system, where this is not the case for GFS or HDFS.

2.3 Map/Reduce Execution Framework

Map/Reduce programs consist of user defined map and reduce functions in addition to configuration codes that set up the runtime environment and define map/reduce phases explicitly. When a user submits a Map/Reduce job, runtime system take care of things like fault-tolerance, starting the execution of MR program, scheduling, and synchronization of map and reduce invocations transparently. With the help of this framework, it is possible to ignore complex parallel programming structures like message passing and synchronization and the programmer only needs to design a mapper and a reducer function for each distinct map/reduce phase.

The sample architecture of a sample Hadoop cluster is shown in Figure 2.3 [13]. All Map/Reduce jobs are submitted to the job submission node where the job-tracker is being executed. The job-tracker takes care of starting and monitoring

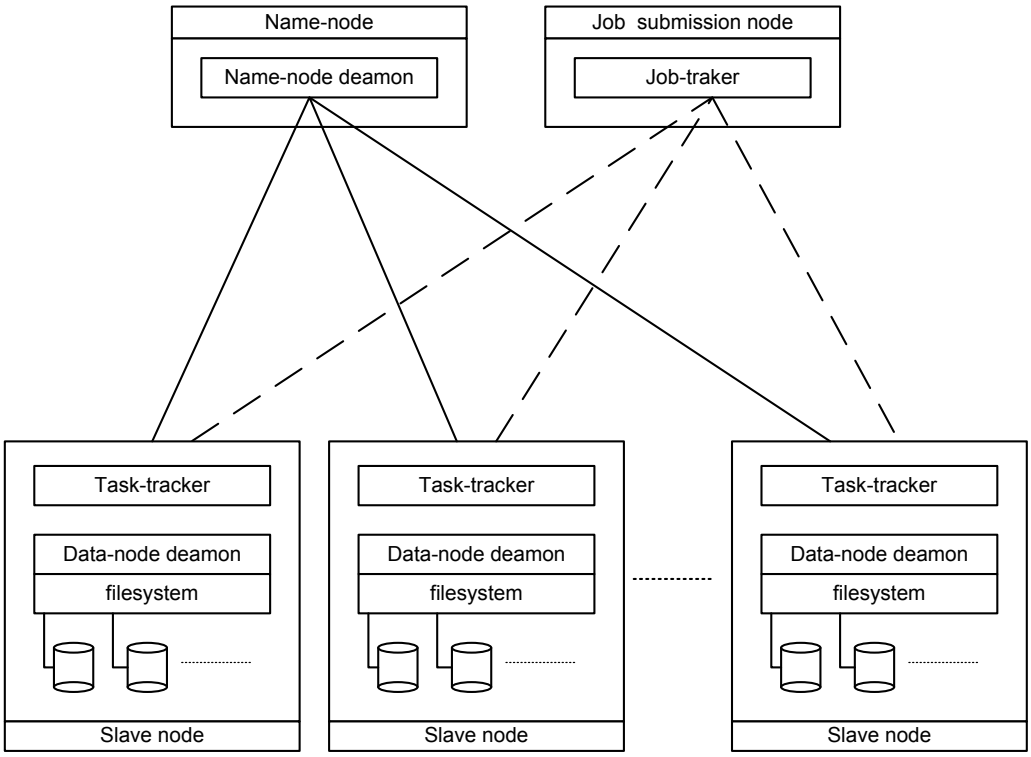

Figure 2.3: Illustration of Hadoop cluster environment

of jobs, scheduling, and coordination of MR tasks. Therefore, job-tracker assigns mapper and reducer tasks to task-trackers. The task-trackers are responsible for executing user defined map and reduce tasks if they are available. Large proportion of the nodes in the Hadoop cluster are slave nodes which run both the task-tracker for executing map and reduce tasks and the distributed file system deamons to store and serve data.

Execution of the Map/Reduce job on a cluster is depicted in Figure 2.4 [1]. When a user submits a Map/Reduce job, the map tasks are distributed to nodes which also store the data chunks. These chunks can be processed separately on different machines requiring no communication. Following the map phase, the intermediate key-value pairs are partitioned according to their key fields. This step is usually carried up by hashing the key fields and partitioning the value range to the number of reducer tasks so that each key-value range induces a reducer task. When partitioning is complete, master node assigns reducer tasks to slave nodes and informs them about the partitions they are to process. Master node

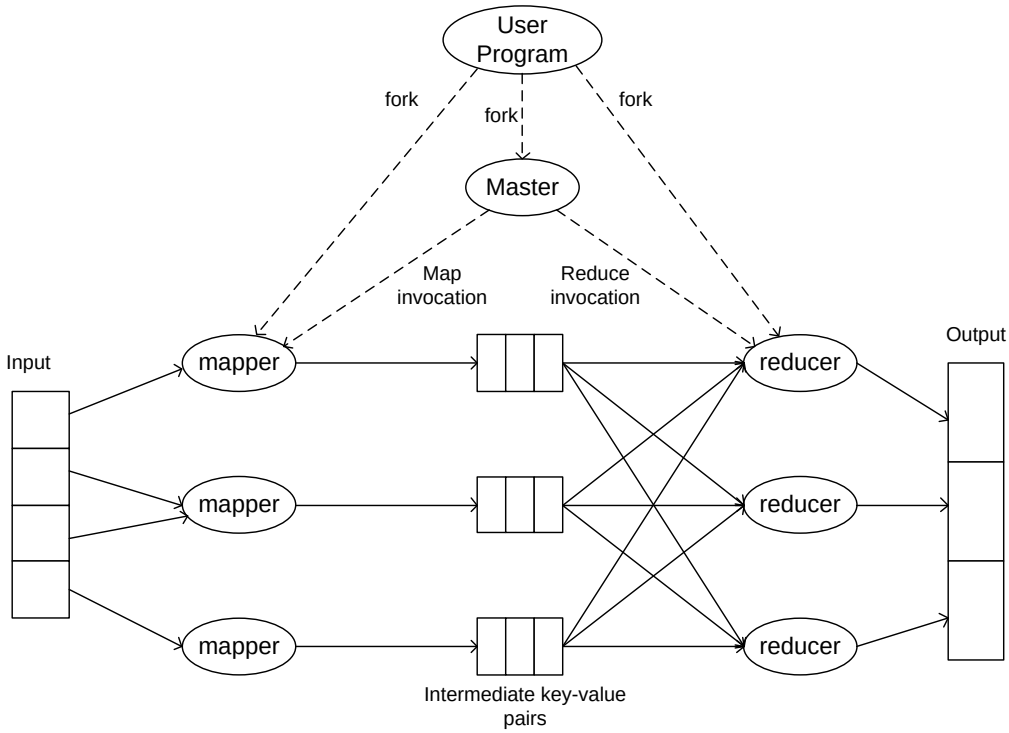

Figure 2.4: Illustration of Map/Reduce execution.

also cares about the location of reducer task to provide locality. For instance, it assigns reducer tasks to slave nodes which are close to the nodes that reducer tasks need to communicate in order to read the key-value pairs assigned on them. After this step the reducers begin to read their input key-value pairs. Whenever a reducer reads all of its intermediate key-value pairs, it sorts the key-value pairs according to their key fields and groups them according to key values. Following the sort operation, all key-value pairs with the same key are merged together to form key-multi-value objects, which are later passed to the user defined reduce function. Reducer function processes these key-multi-value objects one by one and produces final key-value objects, which can be used in further Map/Reduce phases. In Google and Hadoop implementation of Map/Reduce, final key-value objects produced by reduce step are always written back to the distributed file system by the reducer tasks. As mentioned before this property usually causes performance decrease in iterative algorithms.

2.4 MR/MPI Library

MR-MPI library is a light-weight implementation of Map/Reduce programming paradigm and it is designed for MPI-based cluster systems. It is developed in C++ language and uses MPI library for communication operations because MPI library is well optimized for such systems. With the help of this framework, it is possible to ignore complex parallel programming structures like message passing and synchronization and the programmer only needs to design a mapper and a reducer function for each distinct Map/Reduce phase. Another important feature of this library is that huge data sets that do not fit within the aggregate memory of such systems can be processed using the built-in out-of-core algorithms. This means that, memory pages can be swapped between main memory and parallel file system allowing huge datasets to be processed.

To use this library, the programmer writes a main program which runs like a regular MPI code. Main program makes calls to the MR-MPI library for certain operations but most importantly provides user defined functions for map and reduce functions. Library performs map, reduce and data shuffle operations synchronously and uses MPI_Send and MPI_Recv functions between processors to communicate large aggregated messages to improve bandwidth and reduce latency costs [2].

One of the strongest aspects of the library is its in-core and out-of-core operation modes. Whenever a processor creates a Map/Reduce object, it allocates pages of memory to store the key-value or key-multi-value pairs of this object. The size of these pages can be determined by the programmer. During execution, if the size of a page is exceeded, the page is written to the parallel file system and the freed space is used for the new operations. Whenever an old page is required by the program, it must be explicitly indicated by the programmer. If adequate page size is given and proper library settings are chosen, then library automatically informs that the page size is exceeded. On the other hand, out-of-core execution mode helps processing huge data sets whose size exceed total aggregate memory of the cluster. In addition, MR-MPI also provides flexibility

and lets the programmer call MPI functions directly within the code. This is important when global state information needs to be exchanged between processors. One example is the convergence information exchange done by processors during the page rank computations, which is done for terminating the Map/Reduce iterations correctly.

The basic objects on which MR-MPI Library operates are key-value pairs. Keys and values can be of any data type and library treats these as byte strings. Although a value can be NULL, a key-value object always needs a valid key. Key-value objects are stored in MapReduce objects, which must be initialized within the main program. Along with the key-value objects another important type of objects are key-multi-value objects. These objects store multiple values for the same key.

A typical Map/Reduce program using MR-MPI library consists of at least three calls to the MR-MPI library to perform map, collate and reduce operations. In the map phase, the key-value pairs are produced. Map function can accept as input a file name or existing key-value objects or the provided function can generate the key-value objects itself. In MR-MPI, the produced key-value objects are stored in the allocated pages and all the operations on these objects are performed locally requiring no communication.

The collate phase of MR-MPI corresponds to the shuffle operation in the Hadoop based Map/Reduce implementations. This operation produces key-multi-value objects by grouping key-value objects with the same key into a single key-multi-value. Collate requires communication, since different processors may have different key-value pairs with the same key. Key-value pairs are partitioned just before the communication phase according to a hash function which is available by default in the library itself but also can be provided by the programmer. After the partitioning step the MPI_All_To_All library function is used to communicate key-value pairs between processors.

The reduce phase is started whenever key-value pairs are collected by all processors and this time each processor has its own set of keys. This means that the key-value pairs with the same key can not occur on different processors.

Before calling the reduce function the processor sorts the key-value pairs in order to reorganize key-value pairs into key-multi-value pairs. If key-value pairs do not fit in to the local memory external memory sort algorithms are used to perform the operation. Following this step, the reduce function is called for each distinct key-multi-value object.

MR-MPI library extends also the basic functionality provided by Map/Reduce paradigm. It has additional functionalities that can be utilized for speed-up. In the original Map/Reduce framework, it is required to submit each Map/Reduce phase as a separate job, which causes a decrease in performance. In contrast, MR-MPI library does not have such requirements, which leads to a performance increase especially in iterative algorithms like graph algorithms. In the original Map/Reduce framework, initial key-value pairs produced by a map phase are written to the disk system waiting for the reducer tasks to read their own partitions via remote procedure calls. In the MR-MPI library, whenever a mapper task produces its all key-value pairs, it is not obligated to write all of these key-value pairs to the disk but instead, it is possible to communicate these key-value pairs with reducer tasks while storing them in memory. MR-MPI also provides additional functions to manipulate key-value pairs between map tasks and reduce tasks. For example one can reduce some of the key-value pairs and produce new key-values from them. Later it is possible to unite old key-value pairs which are not reduced with the new key-value pairs for further reduction operations. With MR-MPI lots of further optimizations can be achieved while designing new efficient Map/Reduce algorithms. We believe that MR-MPI library has the possibility to make a great impact on scientific computing since it eases parallel programming while providing high scalability for HPC platforms.

Chapter 3

Applications

To test the efficiency of Map/Reduce paradigm in parallelizing scientific computing applications over HPC systems, we selected three of the most frequently used fundamental operations in scientific computing. These operations are: Sparse matrix vector multiplication (SpMxV), multiplication of two large matrices (MxM), and repeated multiplication of two matrices (rMxM). Since we wanted to observe efficiency on actual applications instead of basic operations, we implemented the All Pairs Similarity Search (APSS), All Pairs Shortest Paths (APSP), and PageRank (PR) applications, which extensively depend on SpMxV, MxM, and rMxM operations, respectively.

3.1 All Pairs Similarity Search (APSS)

In the APSS application, given a large set of sparse vectors $V = \{v_1, v_2, \ldots, v_n\}$ representing a high dimensional data, we want to find all pairs of vectors whose similarity measure are above a given threshold value ε. The similarity of two vectors v_i and v_j is computed by a function $sim(v_i, v_j)$. Some of the applications of APSS problem are; query refinement for web search, collaborative filtering, near duplicate document detection and elimination, and coalition detection [20]. Main difficulty that is observed while solving these kinds of problems is the scale

of the problems. That is to say that dimension of the data and number of input vectors can be huge depending on the problem domain. To cope with this issue, some approximate solutions are proposed in literature instead of finding exact solutions for the problems. These approximate solutions generally consider reducing dimension and number of input vectors [21, 22, 23]. Finding exact solution for the APSS problem can be easily carried out by putting all the input vectors in rows of a matrix and multiplying it with its transpose, and by changing inner product operation between row and column vectors to $sim(v_i, v_j)$ function. The resulting matrix contains the similarity measures between all pairs of vectors. Due to the reasons that are mentioned, implementing APSS algorithm with MR-MPI library and running on HPC systems would be a good practice to show handiness of the MR-MPI library in parallelizing the multiplication of two sparse matrices. Note again that APSS algorithm is quite similar to the matrix multiplication algorithm, only difference between the two being the operator used between inner products of the row and column vectors of the input matrices. Therefore, testing this algorithm with above mentioned configurations also provides substantial information about applicability of Map/Reduce paradigm to other scientific computing problems that require sparse matrix multiplication.

3.2 All Pairs Shortest Paths (APSP)

Given a directed graph $G = (V, E)$ and edge weighting function $w : E \to \mathbb{R}$, APSP finds a least-weight path between every vertices $u, v \in V$. The weight of a path is the sum of its constituent edges : $w(p) = \sum_{(i,j)\in p} w(i, j)$. It is assumed that vertices are numbered $1, 2, \ldots, |V|$, and adjacency of the nodes are represented by a matrix (w_{ij}) as given below:

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E \end{cases} \tag{3.1}$$

Output of the algorithm is (l_{ij}) where each element of the matrix shows the shortest path lengths between all nodes. As is well known, the shortest path

problem in a given graph exhibits the following optimal substructure property: each subpath of a shortest path is also a shortest path. Utilizing this optimal substructure property, two distinct dynamic programming (DP) formulations are given in the literature [24] for solving the APSP problem: matrix-multiplication-based algorithm and Floyd-Warshall algorithm. We discuss these two algorithm in the following two subsections respectively.

3.2.1 APSP via Matrix Multiplication

Let $l_{ij}^{(m)}$ denote the minimum weight path from i to j having at most m edges. Formula 3.2 defines the base case where $m = 0$. That means a node has a shortest path only to itself since it is not possible to have a shortest path between any two node having zero edges.

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases} \tag{3.2}$$

If $m \geq 1$, then the recursive formula (3.3) calculates the shortest path between any two nodes that has at most m edges.

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1\leq k\leq n}\{l_{ik}^{(m-1)} + w_{kj}\}\right) \\ &= \min_{1\leq k\leq n}\left(l_{ik}^{(m-1)} + w_{kj}\right) \end{aligned} \tag{3.3}$$

Using the recursive definition given in (3.3) the shortest path weights can be computed in a bottom-up manner as follows.

Algorithm 1: Extend Shortest Path.

Require: L, W
1: $N = L.nrows$
2: **for** $k = 0$ **to** N **do**
3: **for** $j = 1$ **to** N **do**
4: **for** $k = 1$ **to** N **do**
5: $l_{ij} = \min(l_{ij}, l_{ik} + w_k j)$

As one can see, algorithm 1 is very similar to matrix multiplication algorithm

if we make the following operator substitutions:

$$\min \leftarrow + \ ,$$
$$+ \leftarrow . \ .$$

In this way, computing all-pair-shortest-path problem is achieved by multiplying current distance matrix with adjacency matrix. In this context, multiplication is performed by substituting the above mentioned operators. Each multiplication increments the power of the distance matrix by one. Power of a matrix indicates the maximum number of edges that can constitute a shortest path between any two nodes. Besides, there can be at most $|V| - 1$ edges in a shortest path in a directed graph. Therefore, incrementing the power of distance matrix to $|V| - 1$ by multiplying distance and adjacency matrices at least $|V| - 1$ times solves the APSP problem. Following equations show the calculation of the final distance matrix.

$$\begin{aligned} L^{(1)} &= L^{(0)}.W, \\ L^{(2)} &= L^{(1)}.W, \\ &\vdots \\ L^{(|V|-1)} &= L^{(|V|-2)}.W \end{aligned} \tag{3.4}$$

3.2.2 APSP via Repeated Squaring

The result of the repeated matrix multiplications can be obtained much faster by using associativity rule of multiplication of distance and adjacency matrices. Instead of incrementing the power of distance matrix by one in each iteration, one can use repeated squaring method to get the final distance matrix. As it is known that if the input directed graph doesn't contain negative weight cycles, then $L^{(m)} = L^{(|V-1|)}$ for all integers $m \geq |V| - 1$. Therefore it is possible to

compute the $L^{(|V|-1)}$ matrix with $\lceil \log (|V|-1) \rceil$ matrix multiplications.

$$
\begin{aligned}
L^{(1)} &= W, \\
L^{(2)} &= L^{(1)}.L^{(1)} = W.W, \\
L^{(4)} &= L^{(2)}.L^{(2)} = W^{(2)}.W^{(2)}, \\
L^{(8)} &= L^{(4)}.L^{(4)} = W^{(4)}.W^{(4)}, \\
&\vdots \\
L^{(2^{\lceil \log (|V|-1) \rceil})} &= L^{(2^{\lceil \log (|V|-1) \rceil - 1})}.L^{(2^{\lceil \log (|V|-1) \rceil - 1})} \\
&= W^{(2^{\lceil \log (|V|-1) \rceil - 1})}.W^{(2^{\lceil \log (|V|-1) \rceil - 1})}
\end{aligned}
\tag{3.5}
$$

One can see that, if directed graph G doesn't have negative weight cycles and $2^{\lceil \log (|V|-1) \rceil}) \geq |V|-1$, final distance matrix $L^{(2^{\lceil \log (|V|-1) \rceil})}$ is equal to the matrix $L^{(|V|-1)}$.

3.2.3 APSP via Floyd-Warshall Algorithm

Another elegant way of solving APSP problem is Foyd-Warshall (FW) algorithm. The FW algorithm differs from repeated squaring method in dynamic programming formulation and also runs faster on a non-parallel machine. Repeated squaring algorithm has a run time complexity of $\Theta(V^3 \log V)$ whereas Floyd-Warshall has $\Theta(V^3)$. Given the input directed graph $G = (V, E)$, Floyd-Warshall algorithm approaches to DP formulation in a different manner. Firstly, dividing main problem into subproblems, it considers constructing shortest paths from some subset of vertices of V. Lets say we have a shortest path p between the vertices $i, j \in V$ with all intermediate vertices are selected form the subset $\{v_1, v_2, \ldots, v_k\}$ of V. If v_k is not an intermediate vertex of path p, then all intermediate vertices of p are in the subset $\{v_1, v_2, \ldots, v_{k-1}\}$. On the other hand, if v_k is an intermediate vertex of the path, then we can decompose the path p into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$. Because k is not an intermediate vertex for both p_1 and p_2, intermediate vertices of p_1 and p_2 must be selected from the subset $\{v_1, v_2, \ldots, v_{k-1}\}$. Based on these facts, recursive formulation 3.6 solves the problem of APSP. Let d_{ij}^k be the length of a shortest path between vertices $i, j \in V$, and all intermediate vertices are chosen

from the subset $\{v_1, v_2, \ldots, v_k\}$ of V, then the base case of the formulation is for $k = 0$, and $d_{ij}^{(0)} = w_i j$ if edge $(i, j) \in E$:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases} \tag{3.6}$$

Using recurrence (3.6), algorithm 2 computes the APSP. At each iteration of the for loop in line 3, row k and column k is used to update whole distance matrix.

Algorithm 2: Floyd-Warshall

Require: W
1: $N = W.nrows$
2: $D = W$
3: **for** $k = 1$ **to** N **do**
4: **for** $i = 1$ **to** N **do**
5: **for** $j = 1$ **to** N **do**
6: $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
7: $returnD$

3.3 PageRank

PageRank is an algorithm that shows the importance of web pages in the Web. Importance of a web page is determined by the hyperlink structure between the pages. The Web can be represented by a directed graph, where pages are represented by vertices and hyperlinks by directed edges. The algorithm basically depends on the random surfer model in which an imaginary web surfer visits web pages and randomly clicks the hyperlinks on the pages, which corresponds to traversing the directed graph representation of the Web. Hence, PageRank value of a page shows the probability of a random walk over the link structure of the Web after certain number of steps. If a page has z out-links pointing to other web pages, then the surfer chooses one of them randomly with probability $1/z$. In addition, the surfer might also want to visit a random page with a probability α instead of following the links on the page. Using information above, the

PageRank of a page n is computed with the following formula:

$$P(n) = \alpha \left(\frac{1}{|G|} \right) + (1 - \alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)} \tag{3.7}$$

In formula (3.7), $|G|$ is the total number of nodes, α is is the probability of choosing a random page, $L(n)$ is set of pages that has hyperlink to n, and $C(m)$ is the out-degree of a vertex m. Considering the probability of surfer being on node m with probability value of $P(m)$, the surfer randomly chooses a link on the web page with probability $1/C(m)$. Therefore, probability contribution of node m to n is $P(m)/C(m)$. As one can see from the above formula, probability of a web page n is calculated by summing probability contributions from all other pages that have a link to it. Using recursive equation (3.7) Algorithm 3 computes PageRank values of a given web graph.

Algorithm 3: PageRank

Require: G
1: init oldPR, newPR vectors
2: $n \leftarrow |G.V|$
3: **for all** $v \in G.V$ **do**
4: $oldPR[v] \leftarrow 1/n$
5: **while** Convergence is not achieved **do**
6: $c \leftarrow 0$
7: **for all** v that has no out-links **do**
8: $c \leftarrow c + \alpha * \frac{oldPR[v]}{n}$
9: **for all** $v \in G.V$ **do**
10: $newPR[v] \leftarrow c + \frac{(1-\alpha)}{n}$
11: **for all** $u \in L(v)$ **do**
12: $y[v] \leftarrow y[v] + \frac{(1-\alpha)*oldPR[u]}{C(u)}$
13: Normalize y-vector
14: Check convergence

We can also formulate the PageRank algorithm in matrix notation. To do this, we need to define a transition matrix of the Web. The transition matrix $M = (m_{ij})$ consists of n rows and n columns if there are n pages. The matrix entry m_{ij} is defned as $1/out\_deg(j)$ (number of edges leaving a vertex) if web page j has at least one link form j to i.

$$m_{ij} \quad = \quad \begin{cases} 0 & \text{if } (j,i) \notin G \\ \frac{1}{out\_deg(j)} & \text{if } (j,i) \in G \end{cases}$$

Given a vector v_0 representing the PageRank values of the nodes, all pages have the same probability value $1/n$, since random surfer can be at any one of them with equal probability. If we multiply the initial vector v_0 by transition matrix M we get a second probability distribution vector Mv_0, which shows the probability of being on a node after one step. Continuing this step, we can also multiply Mv_0 with M and get $M(Mv_0) = M^2v_0$ vector which shows a probability distribution after second step. It is known that, if the graph represented by M is strongly connected, distribution vector v approaches to a limit value which satisfies $Mv = \lambda v$. This vector v is the eigenvector of M with eigenvalue λ. Actually, if M is a column stochastic matrix, then all column values add up to 1. The vector v is the principle eigenvector which corresponds to the largest eigenvalue with value 1.The eigenvector v shows the probability distribution of surfer being on a page after large number of steps. To find the principle eigenvector of the matrix M we start with initial vector v_0 and multiply it with M to find $v_1 = Mv_0$. Carrying out this step we multiply v_1 and M to find $v_2 = Mv_1$. After a certain number of steps the convergence will be achieved and final resulting vector will be the principle eigenvector.

Even though PageRank computes probability distribution of web pages, it is not an exact solution but an approximation of probability distribution of the pages unless some cases are handled properly, since it depends on the assumption that input graph is strongly connected. In fact, it is not possible to reach from any node to any other node in a real world Web graph, which means there is no single strongly connected component in the graph. For instance, there may be some nodes that have no out-links to other pages. These pages are named as dangling nodes. If these pages are not handled properly, the total probability distribution that adds up to one will loss some of its proportion, since probability mass arriving at these nodes will not be transfered to other vertices. So, one proper way to handle this situation is to distribute the whole probability values of these dangling nodes to all nodes in the graph evenly. Using information above, one can calculate PageRank distribution of web pages with the following formula given in matrix notation:

$$v' = \alpha Mv + (1 - \alpha)e/n \tag{3.8}$$

Equation 3.8 correctly calculates PageRank distribution in case there are no dangling nodes in the graph. Therefore, probability calculation of theses dangling nodes must be calculated separately. In addition to dangling nodes that need extra care to calculate the PageRank values of the pages are pages that have self-links named as spider-traps. If there are spider-traps in a directed graph, these pages get all probability mass during PageRank computation. Hence, PageRank values found at the end of computation are not actual values. Solution of these kinds of problems are out of scope of the thesis, so they will not be covered in here.

As one can see from the formulation (3.8), PageRank calculation requires repeated sparse matrix-vector multiplications on huge data sets. So, if the size of the data sets are considered, one can see that the main difficulty of the PageRank algorithm is scalability. Hence, to be able to calculate PageRank for huge data sets, one need to benefit from distributed computing systems consisting of large number CPUs and memory systems.

Chapter 4

MR-MPI Implementation Details

In this chapter, implementation details of the applications that are described in Chapter 3 will be covered. All codes are developed in C++ programming language and MR-MPI library is used for parallelization of these algorithms. Moreover, all the codes are tested on the Juqeen [6] and Hermit [7] supercomputing systems. All these supercomputing systems have a distributed file system that is separated from their network of compute nodes. Juqeen uses GPFS and Hermit uses Lustre distributed file systems for high performance parallel I/O operations. Huge data sets that are used for testing purposes are stored on these distributed file systems by partitioning into chunks. Later, MPI parallel I/O functions are used to read these data sets from distributed file systems. With the help of the MPI library, huge amount of data is drawn to computation network in a short period of time. All data sets that we used were graphs that represent social networks or link structure of the Web. These data sets were in the form of $(i, j)(m_{ij})$ key-value pairs showing coordinates of the non-zero elements of matrices which can be used to represent adjacency structure of graphs.

4.1 All Pairs Similarity Search (APSS)

In order to solve the APSS problem, we used a method that a matrix representing the input data is multiplied by it's transpose. The matrix is formed as placing vectors which represent high dimensional data at the row positions of the matrix. Also, matrix multiplication is performed by using a similarity function between matrix elements instead of plus and multiplication operators that is used by standart matrix multiplication.

Matrix multiplication algorithms can be implemented in two different ways with Map/Reduce paradigm. One way performs multiplication with only single Map/Reduce phase and the other performs in two distinct Map/Reduce phases. We decided to use the later two-phase approach; because this approach is much more scalable than the former one. In two-phase approach, column vectors of the first matrix are multiplied by the row vectors of second matrix and each vector multiplication produces a matrix consisting of partial results which are later combined to produce resulting matrix. On the other hand, in one-phase approaches row vectors of the first matrix are multiplied by the column vectors of the second matrix and inner product operation is performed between the row and column vectors. After the inner product operation, we get a single element corresponding to one of the entries of the resulting matrix. Moreover, one-phase approach needs more replication of the input matrix elements compared to the two-phase approach [25]. Therefore, one-phase approach performs more communication, and thus the two-phase approach is more efficient even it needs two distinct Map/Reduce job phases. Additionally, even though second approach needs two phases, MR-MPI library does not need intermediate key-value pairs to be written to the distributed file system, which provides low latency between Map/Reduce phases. Hence, using the two-phase approach with MR-MPI library does not have such overhead that is seen in other implementations of Map/Reduce. Additionally, we regarded scalability more; because memory available at compute nodes was limited.

Algorithm 4 shows the psuedo-code implementation of the APSS algorithm.

At the beginning of the algorithm, two MR-MPI library objects named as E and T are initialized. Object E stores key-value objects which are read from distributed file system using parallel I/O functions of the MPI library. Key-value pairs are in the form of $(i, j)(m_{ij})$ that corresponds to non-zero entries in the input matrix. Here (i, j) is the key field that i and j correspond to row and column coordinates of the non-zero element and the (m_{ij}) field is the actual non-zero element of the matrix. As for object T, it is an empty object; but filled by mapping key-value pairs stored in the object E. Whenever library object E is filled by key-value

Algorithm 4: APSS

Require: $E = (row_i, col_j)(m_{ij})$
1: Init MR objects E and T
2: $T \leftarrow E.map(mapper1DRW)$
3: $E \leftarrow E.map(mapper1DCW)$
4: $E \leftarrow T.add()$
5: $delete\ T$
6: $E.collate(NULL)$
7: $E \leftarrow E.reduce(reducerOP)$
8: $E \leftarrow E.map(mapper1DCW)$
9: $E.collate(NULL)$
10: $E \leftarrow E.reduce(reducerSP)$
11: $returnE$

objects, E is mapped with user defined function $mapper1DRW$ function in line 2 of the algorithm. After this operation is performed, resulting key-value pairs are stored in object T. In this function, input key value objects $(i, j)(m_{ij})$ are transformed into $(i)(j, m_{ij})$ key-value pairs. This operation maps the input matrix row-wise for the matrix multiplication as given in algorithm 4. In line 3, the second library object E is mapped with user defined function $mapper1DCW$. This function basically maps input matrix column-wise by transforming the $(i, j)(m_{ij})$ key value pairs into $(j)(i, m_{ij})$ pairs. In line 4, the key-value pairs in the object T are all added to object E and object T is deleted. Whenever the add operation is finished, the $collate()$ function is called where all the key-value pairs are hashed according to key fields and are distributed to processors. When collate phase finishes, non-zero entires of row vectors of the matrix E and column vectors of the matrix T with the same keyfield are all collected by the same processor. After this operation had been performed, all the key-value objects with the same

key are merged into key-multi value object in the form of $(j)(i, [m_{ij}])$. Then, in line 7 the reduce operation is performed with user defined function $reducerOP$. Within the function $reducerOP$, row vector entries and column vector entries are separated. Following this step, an outer vector product operation is performed and a new intermediate matrix with partial entries is formed. For this reason, we prefered the two-phase approach while implementing the APSS algorithm. All of the entries in the form of $(i, j)(m'_{ij})$ of the intermediate matrix is than mapped column-wise in line 8 of the algorithm with the same function used in line 2. Then, collate operation is performed in line 9 of the algorithm to sum each partial results to form final resulting matrix. Each column of the each intermediate matrix produced by the outer product operations are mapped and distributed to processors. After the collate phase, partial results in different intermediate matrices which have the same column indices are gathered by the same processor. Therefore, it is possible to compute columns of the resulting matrix locally by all processors. In other words,the resulting matrix are partitioned by columns and each processor is responsible to compute some subset of columns of the result matrix. In line 11 resulting matrix is computed and stored as key-value pairs in the library object E.

4.2 All Pairs Shortest Paths (APSP)

The input graph is represented by an adjacency matrix and the adjacency matrix is multiplied $\log n$ times by itself where n is the number of nodes in the graph. In the second method, we implemented Floyd-Warshall (FW) algorithm in which a different approach for dynamic programming formulation is used. Moreover, the Floyd-Warshall algorithm is also different compared to the matrix multiplication algorithm. In this algorithm there is an n iteration main loop where at each iteration i, row i and column i are used to compute each element of the distance matrix for further iterations. As mentioned above, in repeated squaring method a sparse matrix is repeatedly multiplied by itself and after each iteration, resulting matrix becomes much denser. Hence, it was not possible to test this algorithm with huge matrices; because algorithm itself has running time complexity

of $\Theta(n^3 \log n)$. In addition, Floyd-Warshall algorithm also has high asymptotic complexity that it is $\Theta(n^3)$. As mentioned in Section 3.2.2, while performing repeated squaring method we used two phase matrix multiplication algorithm because of the same reasons.

4.2.1 APSP via Repeated Squaring

Algorithm 5 shows the repeated squaring (RSQ) method to calculate all pair shortest paths between vertices. The RSQ algorithm is very similar to APSS

Algorithm 5: RSQ

Require: $E = (row_i, col_j)(m_{ij}), N$
1: Init MR objects E and T
2: $M = 1$
3: **while** $M < N - 1$ **do**
4: $T \leftarrow E.map(mapper1DRW)$
5: $E \leftarrow E.map(mapper1DCW)$
6: $E \leftarrow T.add()$
7: $delete\ T$
8: $E.collate(NULL)$
9: $E \leftarrow E.reduce(reducerOP)$
10: $E \leftarrow E.map(mapper1DCW)$
11: $E.collate(NULL)$
12: $E \leftarrow E.reduce(reducerSP)$
13: $M \leftarrow M * 2$
14: $returnE$

algorithm; since core operation for the two algorithm is the matrix multiplication. In line 1 MR-MPI library objects are initialized and object E is filed up by key-value pair that are read from distributed file system. Following this the variable M which shows the iteration number is set to 1. Therefore, the while loop between lines 3 and 13 iterates $\lceil \log(|N| - 1) \rceil / 2$ times. Within the while loop, two-phase matrix multiplication is performed by changing certain operator changes as mentioned in Section 3.2.2. In line 13, variable M is doubled. Whenever while loop finishes, resulting object E holds key-value pairs which correspond to final distance matrix entries.

4.2.2 APSP via Floyd-Warshall (FW) Algorithm

In the FW algorithm each iteration i, row i and column i are used to update all elements of the distance matrix. The number of columns and rows in a distance matrix is n. Therefore, main loop in Floyd-Warshall algorithm iterates n times.

Algorithm 6: Floyd Warshall

Require: $E = (v_i, v_j)(w_{ij}), N$
1: $E \leftarrow E.map(mapper1DCW)$
2: $E.collate()$
3: $E.reduce(reducer1DCW)$
4: **for** $k = 0$ **to** $N - 1$ **do**
5: $\quad T \leftarrow E.map(kthColMapper)$
6: $\quad T.collate(NULL)$
7: $\quad T.reduce(kthColReducer)$
8: $\quad E.convert()$
9: $\quad E.reduce(kthIterationReducer)$

The algorithm requires MR-MPI object E that is filled up by key-value pairs corresponding to each matrix element of an adjacency matrix. The key-value pairs are in the form of $(v_i, v_j)(w_{ij})$, where v_i, v_j are source and target nodes and w_{ij} is the distance between the two. In lines 1 to 3, as given in the algorithm 6, the MR-MPI object E is mapped with user defined function $mapper1DCW$. User defined function $mapper1DCW$ converts the key-value pairs in the form of $(v_i, v_j)(w_{ij})$ to the form of $(v_j)(v_i, w_{ij})$. This operation maps each column of the input matrix. Hence, a column-wise partitioning is attained between reducer tasks. Whenever the lines 1 to 3 are executed, the key-value pairs corresponding to the same column entries in the distance matrix are gathered by the same processors. Therefore, distance calculation at each iteration can be performed locally. The for loop in line 4 iterates n times and at each iteration, column k is replicated to all reducer tasks. This operation is performed by mapping object E with user defined function $kthColMapper$ and by adding the resulting key-value pairs to the library object T in line 5. Column k is replicated by the number of reducer tasks with this function. After mapping operation had been performed in line 6, the $collate()$ function is called on object T which distributes all copies of column k to reducer tasks. Following this step, in line 7 reduce function is called

on object T with user defined function $kthColReducer$ which stores all column entries of kth column in a vector which is stored in memory. Following this, in line 6, the $convert()$ function of object E is called. The $convert()$ function converts all key-value pairs to key-multi-value objects by merging all key-value pairs with the same key into a single element. The key of the resulting key-multi-value object is the old key of key-value pairs that are merged and the key-multi-value object stores all the values. In the case of algorithm 6, the $convert()$ function call only converts key-value objects to key-multi-value objects because key fields of all key-value objects are distinct in processor. In other words, this operation is obligation of the MR-MPI library that reduce functions can be performed on MR-MPI library objects that have key-multi-value objects. In this regard, if a library object has key-value pairs, then it is not possible to call reduce on that object which forces programmer to convert the key-value objects to key-multi-value objects. Afterwards, in line 9 of the algorithm, reduce function which is provided with user defined $kthIterationReducer$ function is called on library object E. In this reduce phase, local portion of the kth row is separated from other key-value pairs and stored in a vector in memory; since it is used to update distance matrix entries that are stored in th object E. Following this, the values of all key-value pairs are updated using kth column and local portion of kth row according to recursive formulation given in (3.6). Whenever update operation is completed, all updated key-value pairs are again stored in object E in the form of $(v_j)(v_i, w_{ij})$.

4.3 PageRank

As mentioned in chapter 3.3, PageRank algorithm can be performed by multiplying a sparse transition matrix with a dense PageRank distribution vector repeatedly until convergence is achieved. One proper way for matrix vector multiplication with Map/Reduce paradigm is partitioning the matrix one dimensional by columns and partitioning the vector conformable with column partitioning of the matrix. In other words, one needs to distribute the input vector and matrix in a way so that column i of the matrix and row i of the vector goes to

same processor to perform multiplication operation locally. While implementing PageRank algorithm, we used parallel I/O operations to read input matrix which represents the Web graph. Matrix elements are represented by key-value pairs in the form of $(i, j)(a_{ij})$ where i and j correspond to row and column indices and a_{ij} corresponds to non-zero element of the matrix. Whenever input key-value pairs are completely read from the distributed file system, these key-value pairs are mapped column-wise to produce key-value pairs in Then, new key value pairs are distributed to processors with $aggregate()$ function call of the MR-MPI library using a user defined hash function. With the help of the hash function, it is possible to designate the processors to which non-zero elements of a column are assigned. Following this step, input vector partitions that are conformable with matrix partitioning are initialized by each processor locally and each vector element has a value of $1/n$ where n is the number of web pages.

Before beginning to PageRank computation, one step that must be taken is finding dangling vertices which correspond to columns that have all entries equal to zero. This operation can be performed locally by each processor; since all columns that have at least one non-zero and entries of the input vector that are conformable with column-wise matrix partitioning are locally available. Adding key-value pairs of the input vector to the respective pairs of matrix, one can call the $convert()$ function to create key-multi-value objects using all the key-value objects. Hence, key-value pairs with key i which correspond to non-zero elements of column i and key-value objects with key i that is the ith row of the input vector will be merged together to form key-multi-value objects. In this way, calling a reduce function on all the key-multi-value objects, columns that do not have any non-zero elements can be found easily by just looking to key-multi-value objects that have only one value element; since the only key-value object with a key i which corresponds to dangling nodes is the ith row entry of the input vector.

Implementation of PageRank is provided in the Algorithm 7. The algorithm requires MR-MPI library objects A, M, x, and y. Firstly, object A stores the input matrix elements in the form of $(j)(i, a_{ij})$ which corresponds to column-wise partitioning of the matrix. Secondly, object M is required to keep indices of the empty columns of the matrix in the form of $(j)(NULL)$. Lastly, x is the

initial vector of PageRank distributions in the form of $(i)(x_i)$ and y is an empty object which will be later used to store the result vector of the first matrix vector multiplication.

Algorithm 7: PageRank

Require: $A = (col_j)(row_i, m_{ij}), M = (col_j)(NULL), x = (row_i)(1/n), y$

1: **while** $residiul < tolerance$ **do**
2: $c = 0$
3: $M.add(x)$
4: $M.convert()$
5: $M.reduce(reducerComputeAdjustment)$
6: $c = MPI\_Allreduce(c, MPI\_SUM)/n$
7: $y.add(x)$
8: $y.add(A)$
9: $y.convert()$
10: $y.reduce(reducerIP)$
11: $y.collate()$
12: $y.reduce(reducerSUM)$
13: $y_{max} = 0$
14: $y.map(mapperMAX)$
15: $y = MPI\_Allreduce(ymax, MPI\_MAX)$
16: $y.map(mapperScale)$
17: $x.add(y)$
18: $x.convert()$
19: $residual = 0$
20: $x.reduce(reducerComputeResidual)$
21: $residual = MPI\_Allreduce(residual, MPI\_MAX)$
22: $x = y.copy()$
23: $returnx$

The while loop in line 1 iterates until the convergence is achieved. In each iteration, the following operations are performed in succession. Firstly, in lines 1 to 5, computations that are required by dangling nodes are performed. To do that, key-value pairs of the object x are added to the object M in line 3 and *convert*() function is called on M to convert key-value objects in to key-multi-value objects. After this operation is performed, the reduce function with user defined function *reducerComputeAdjustment* is involved. With the help of this function, total probability mass arriving at dangling nodes that are stored locally is calculated at each processor. Then in line 6, using MPI_Allreduce function, the whole probability mass arriving at all dangling nodes is globally summed and divided by n in order to make adjustments to the probability distribution of all

vertices. The only communication required to perform these operations is global reduction operation. All the other operations are performed locally due to the pre-aggregation step performed at the beginning of the program.

Next step of the algorithm is multiplication of transition matrix A by vector x. The multiplication operation is performed firstly in lines 7 to 12 by adding key-value pairs of objects x and A to the object y and calling $convert()$ function which produces key-multi-value objects in the form of $(j)(x_j, [i, a_{ij}])$. The vector entry x_j can be differentiated from other entries that are recieved from the nonzero entries of matrix A by using some library specific functionalities. So, it is possible to perform multiplication of vector entry x_j by all matrix elements a_{ij} in the same column. The results of each separate multiplication operation is a partial result of the final corresponding vector entry y_i. For instance, if x_j is multiplied with matrix entry a_{ij}, then the result of this operation is a partial result of the final vector entry y_i. Each partial result is mapped with their row indices and added to object y as a key-value pair in the form of $(i)(y_i)$ for the later steps of the PageRank algorithm. Finally, to come up with the final resulting vector y, a second phase of Map/Reduce is needed to sum all partial results that contribute to same y-vector entry. Therefore, after the first reduce phase, the $collate()$ function of the library is called on object y to collect all partial results of the same y-vector entry on the same processor. After the collate operation, it is now possible to compute the final y-vector entries locally on each processor. The reduce operation performed on line 12 performs the summation of the partial results that are gathered from other processors and produces resulting y-vector entries in the form of $(i)(y_i)$ key-value pairs.

After the computation of the output vector y, it is time to check convergence criteria to be able to properly end the execution of the algorithm. In the algorithm 7, in lines 13 to 21, difference value between x and y vector named as residual is computed. In lines 13 to 15, maximum y vector entry which is the infinity norm of the vector is found by a single step map phase which is called with user define function $mapperMAX$ and does not require communication. Additionally, probability value c which is calculated in line 6 is added to each entry of vector y by $y_i = y_i + y_{max}$. Then, the global reduction operation is performed using

MPI library to find the global maximum y-vector entry is found in line 15. Using the global maximum y-vector entry y_{max}, all y vector entries are scaled by map function which is called with $mapperScale$ function. With this map phase, each entry of y is scaled by $y_i = y_i/y_{max}$ operation. Finally, in lines 17 to 21, the difference of the two vectors x and y is found and the infinity norm of the difference is computed. In line 17 key-value pairs of y is added to x and $convert()$ function is called. After the convert operation, the reduce function is called with user defined function $reducerComputeResidual$ on object x. In the reducer function infinity norm of the difference of the vectors is found by $\max(|x_i - y_i|)$ operation; but this operation is performed locally in which local maximum is found. Therefore, globally maximum entry of the difference vector must be found in line 21 where global reduction is used among processors. The entry that has maximum value of the difference vector is copied to variable $residual$ which is later used to check whether convergence is achieved or not. Following that operation, library object y is copied to object x which will be used in further iteration until convergence is achieved. Whenever the while loop is completed, we have the resulting PageRank distribution vector which is normalized using infinity norm.

Chapter 5

Experimental Results

In this chapter, the results of the experiments that are performed on HPC systems will be provided. Experiments are performed to measure the total execution time in order to observe performance metrics such as speedup and scalability. All of the applications are implemented using MR-MPI library for HPC systems. Algorithmic and implementation details of the applications are elaborated in Chapter 3 and 4 respectively. In addition to experimental results, a data set properties and hardware features of HPC systems are also provided. The experiments are performed on the two HPC systems namely as Juqeen and Hermit. The following tables 5.1 and 5.2 provide hardware features of these two systems.

During the experiments, several real world graph datasets which represent social events or link structure of the Web are used. For instance, we used LiveJournal which is taken from Stanford University Large Network Dataset Collection [26]. The LiveJournal is an on-line community in which significant fraction of members are highly active. It allows members to maintain journals and to declare which other members are their friends. Each node in the graph corresponds to a member in the network and an edge is added between two nodes if respective members become friends. An other important real world data set is uk-2007 Web graph which is part of DELIS dataset [27]. The DELIS dataset is a collection of web graphs focusing on *.uk* domains which is collected by monthly snapshots.

Table 5.1: Hardware configuration of the Juqeen (IBM BlueGene/Q) system

Features	Description
Compute Nodes	IBM PowerPC A2, @ 1.6 GHz, 16 cores, 16 GB SDRAM-DDR3 per node
Number of CNs	28672 nodes
Number of Racks	28 racks
Total number of cores	458,752 cores
Overall main memory	448 TB
Networks	5D Torus 40 GBps; 2.5 sec latency, Collective network part of the 5D Torus, Global Barrier/Interrupt part of 5D Torus, 1 GB Control Network System Boot, Debug, Monitoring
Distributed File System	GPFS
Peak performance	5.9 Petaflops

Table 5.2: Hardware configuration of the Hermit (Cray XE6) system

Features	Description
Compute Nodes	AMD Opteron(tm) 6276 (Interlagos) processors (2 per node) @ 2.3 GHz (up to 3.2 GHz with TurboCore) 32GB RAM/node 2GB/Core
Number of CNs	3552 nodes
Number of Racks	28 racks
Total number of cores	113.664 cores
Overall main memory	126 TB
Networks	Gemini Interconnect (3D torus)
Distributed File System	Lustre
Peak performance	1 Petaflops

Additionally, we also used randomly generated graphs that are recursively generated with power-law degree distributions. These random graphs generally used to represent social networks. The properties of these datasets are presented in Table 5.3.

Table 5.3: Dataset Properties

Name	Type	Nodes	Edges	Description
Random Graph1	Directed	262,144	10,485,760	Randomly generated
Random Graph2	Directed	256	2560	social network
Random Graph3	Directed	512	5120	
Random Graph4	Directed	1024	10240	
LiveJournal	Directed	4,847,571	68,993,773	LiveJournal online social network
uk-2007	Directed	105,896,435	3,738,733,649	Web graph that is part of the DELIS project

5.1 APSS Experiments

The APSS experiments are performed with *RandomGraph*1 and *LiveJournal* datasets whose properties are presented in Figure 5.3. These experiments are all performed in Hermit Supercomputing system which is described in Table 5.2. As mentioned earlier, performing APSS algorithm on social graphs can be used in recommendation systems in which friendship suggestions are offered to people if their friendship profile is similar to someone. Here we present experimental results of the APSS algorithm in Table 5.4. The corresponding graphs to the run time experiments are presented in Figures 5.1 and 5.2.

In Figure 5.1, APSS algorithm where the *RandomGraph*1 is given as input is run on 128, 256 and 512 CPU cores. As one can see the Map/Reduce implementation of APSS scales up to 512 cores improving total running time. There is some efficiency loss Between 256 and 512 cores due to parallelization overheads but it is possible to gain higher speedups by increasing input size. In Figure 5.2, APSS is run on $1K$, $2K$, and $4K$ cores with input data of *LiveJournal* graph.

A super linear speedup is achieved between $1K$ and $2K$ cores since the size of the data that is assigned to each processor is bigger then the size of the memory available at each core which causes I/O operations to swap pages of key-value pairs between memory and distributed file system. Unfortunately, only a slight improvement is achieved by increasing the number of cores to $2K$ due to efficiency loss that shows that algorithm did not scale. It is important to note that parallelization overheads grow at least linearly as the number of processors increases. Moreover, Hermit super computing system has an interconnection network that provides low bisection width compared to Juqeen system which causes Hermit system to have higher parallelization overheads since MPI_Alltoall communication is performed among the processors in the collate phase of the library. As the results show, MR-MPI library provides high scalability up to a certain number of processors if it is provided with right input size to keep efficiency at a fixed value. The scalability can further be improved by optimizing both algorithm itself, and the library specific configurations, and using a supercomputing system which provides higher bisection width. Additionally, as one can see the huge data sets can be processed on the HPC systems in a very small period of time which is not possible with other parallel computing systems.

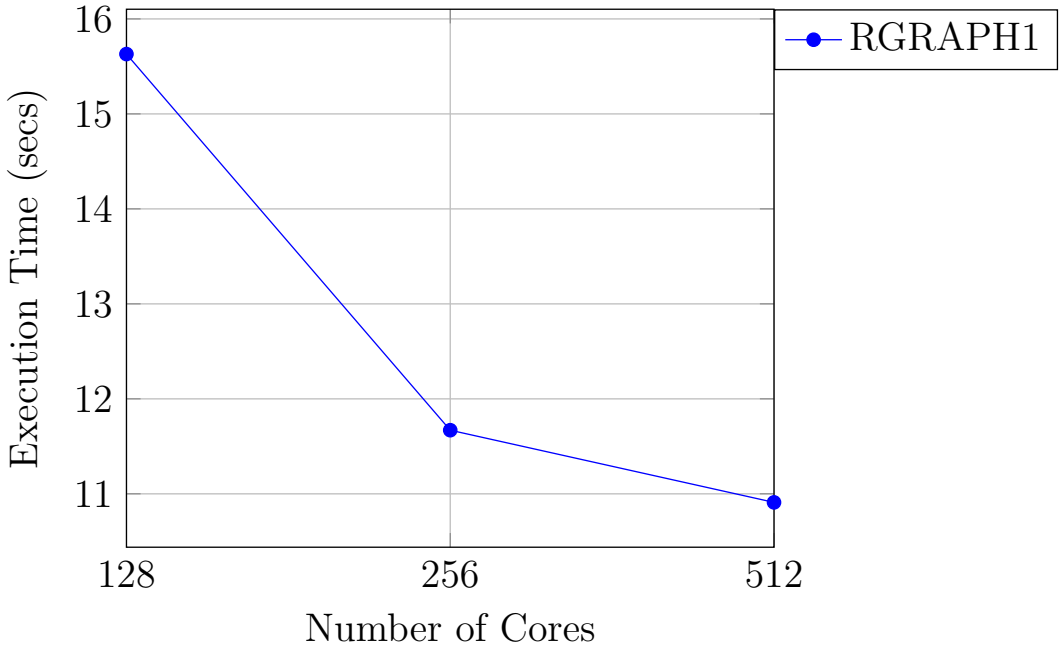

Figure 5.1: APSS algorithm run on *RandomGraph*1

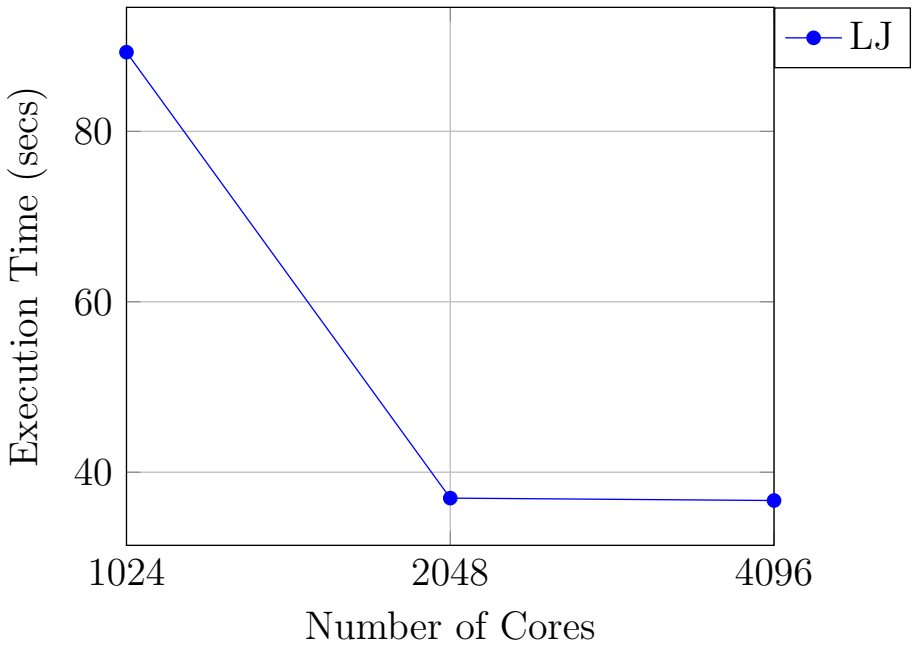

Figure 5.2: APSS algorithm run on *LiveJournal*

Table 5.4: Execution time results gathered from the APSS experiments

	RG1			LJ		
Number of procs.	128	256	512	1024	2048	4096
Execution time (secs):	15.63	11.67	10.91	89.29	36.96	36.68

5.2 APSP Experiments

Performing APSP on a graph might be used to find betweenness centrality measure of nodes in the graph in big data applications. Betweenness centrality of a node is equal to the number of shortest paths passing through that vertex. The measure is used to find a node's centrality in a network. The APSP experiments are performed on Hermit Supercomputing system using data sets *RandomGraph*2, *RandomGraph*3, and *RandomGraph*4. As mentioned earlier two different algorithms are used to solve the APSP problem. The first is RSQ method where adjacency matrix is multiplied with itself repeatedly and the second one is the FW algorithm. Algorithmic details of these algorithms are provided in Section 3.2.2 and 3.2.3 respectively.

Here, the experimental results of the two algorithms are presented in Figure 5.3 and in Table 5.5. As one can see the three different experiments are performed for each algorithm. In the experiments, we increased both the input size and the number of processors to compare execution time of the algorithms. Increasing the number of vertices corresponds to increasing the problem size cubically due to the serial execution complexity of the algorithms that are $\Theta(V^3 \log V)$ for the RSQ and $\Theta(V^3)$ for the FW. We used 128, 256, and 512 CPUs for the *randomGraph*2, *randomGraph*3, and *randomGraph*4 respectively. In each experiment RSQ method performed significantly better than FW algorithm even FW algorithm has lower complexity for the serial execution. The reason is that the number of Map/Reduce iterations is $\log n$ for RSQ, whereas it is n for the FW algorithm. While performing these experiments we observed that even though the MR-MPI library provides low latency between Map/Reduce phases, the large number of iterations still contains substantial overheads. Moreover, it is observed that the Hermit system offers an interconnection network that has low bisection width which causes more parallelization overheads compared to Juqeen system. The reason for that is MR-MPI library performs MPI_Alltoall communication between the Map/Reduce phases which cause each processor to communicate with every processor that is allocated for the program execution. Therefore, it is more

important to prefer algorithms that requires small number of Map/Reduce iterations and round minimization is more important then reduced computational complexity, at least for small enough dataset sizes.

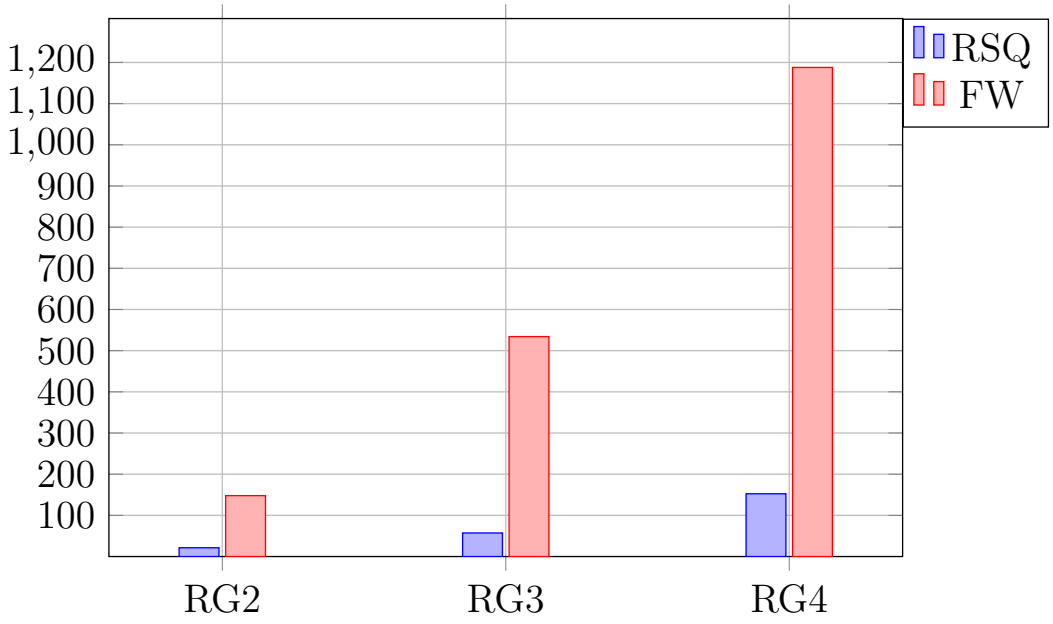

Figure 5.3: APSP algorithm run on *RandomGraph*2(RG2), *RandomGraph*3 (RG3), and *RandomGraph*4 (RG4)

Table 5.5: Execution time results gathered from the APSP experiments

	rgraph1 $k = 128$	rgraph2 $k = 256$	rgraph3 $k = 512$
RSQ	21.36	57.22	152.47
FW	147.91	534.19	1187.81

5.3 PageRank Experiments

PageRank experiments are performed on Juqeen supercomputing system using data set uk-2007. The uk-2007 data set is a huge sparse matrix which is a directed graph representing link structure of web sites under .uk domain. This huge sparse matrix is repeatedly multiplied with a dense PageRank distribution vector until the convergence is achieved. The experiments are performed using 256, 512, $1K$,

and $2K$ CPU cores and the total execution time of the algorithm using those configurations are provided in Figure 5.4 Table 5.6. Figure 5.4, a super linear speedup is achieved between the 256 and 512 cores. The reason is that the data that is assigned to each processor is bigger than the size of the memory available for each core which causes I/O operations to swap pages of key-value pairs between memory and distributed file system. If the input data is distributed to more CPU cores, than it is possible to fit the partitioned data to local memory available for each CPU core. Moreover, between the 512 and $2K$ cores almost linear speedup is achieved which shows the efficiency and the scalability of the MR-MPI implementation of the PageRank algorithm. On the other hand, the scalability of the PageRank implementation also depends on the architecture of the Juqeen system; since it has an interconnection network which provides higher bisection width compared to Hermit system. The results of the experiments show us the low parallelization overhead of the MR-MPI library especially when used on Juqeen supercomputing system. Due to low parallelization overheads, it is possible to scale to large number of processors to process huge data sets efficiently which shows usefulness of the MR-MPI library for the big data applications.

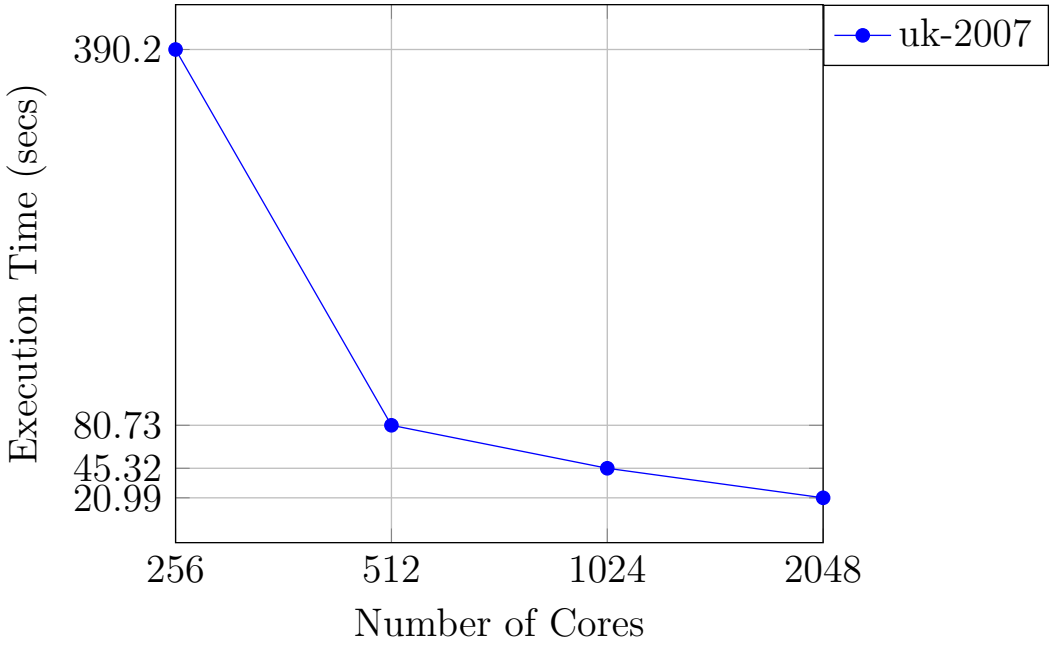

Figure 5.4: PageRank algorithm run with uk-2007 data set

Table 5.6: Execution time results gathered from the PageRank experiments

Number of procs.	256	512	1024	2048
Execution time (secs):	390.20	80.73	45.32	20.99

Chapter 6

Conclusion

Experimentally it was verified that Map/Reduce paradigm can be useful in HPC systems; since it provides both ease of parallel programming and scalability for the data intensive applications. Especially, using a library such as MR-MPI which is designed for HPC systems is beneficial because of it's certain aspects. First of all, the MR-MPI library has more functionalities than the other Map/Reduce frameworks such as Hadoop and Google. Additionally, the library itself is designed for HPC system where communication is much more faster than writing chunks of data to a distributed file system. Therefore, the intermediate key-value pairs are distributed among processors by message passing which decreases latency between Map/Reduce phases. The benefits of the library and it's being the only implementation for HPC systems currently, makes it the best choice for such systems. On the other hand, even the MR-MPI decreases latency between Map/Reduce phases and provides much more performance compared to the other implementations, it is not so sufficiently efficient in iterative algorithms especially when used in HPC system that have an interconnection networks having low bisection width. While performing the scalability tests of the applications using MR-MPI library, it was observed that the properties of the interconnection network have significant impact on parallelization overheads of the MR-MPI library. For this reason, Juqeen system provided more scalability compared to Hermit system. Therefore, trying to decrease the number of Map/Reduce job phases

while designing Map/Reduce algorithms is a good strategy to improve runtime and scalability of applications. Besides, it was observed that data intensive applications can also be run efficiently on HPC systems using MR-MPI library.

The applications in the of data mining and information retrieval which are generally preferred to run on commodity PC clusters, can also be run efficiently on HPC systems. Because it is possible to store huge data sets and process these data sets efficiently on HPC systems.

The Map/Reduce paradigm is used more frequently in data mining and information retrieval domains. During our studies, we implemented sparse matrix multiplication and sparse matrix vector multiplication algorithms, which are fundamental operations in such domains. It is also known that the implemented algorithms are also frequently used in scientific computing domain which shows that Map/Reduce paradigm can also be used in scientific computing applications as well.

As a future work, we consider to implement some optimizations which try to decrease communication among processors between Map/Reduce phases by preprocessing input data to provide locality to reducer tasks. Additionally, it is possible to configure size and number of memory pages which are required by the library to store key-value pairs. These pages are swapped transparently between memory and distributed file system during execution of an algorithm. These configurations also need some optimizations to achieve more scalability and speedup. Also the algorithms might be designed in considering the memory page configurations.

In conclusion, Map/Reduce paradigm and variations of frameworks are well understood during our studies. Additionally, designing efficient Map/Reduce algorithms for HPC systems is a well guide for future works. Also, we observe that research in big data applications attracts more interest than ever in these days. Therefore, we believe that concentrating on the tools that are used to solve big data problems is generally of high importance.

Bibliography

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[2] S. J. Plimpton and K. D. Devine, "Mapreduce in mpi for large-scale graph algorithms," *Parallel Comput.*, vol. 37, pp. 610–632, Sept. 2011.

[3] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. O. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, and D. E. Shaw, "A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 56:1–56:12, IEEE Press, 2008.

[4] J. Ekanayake and G. Fox, "High performance parallel computing with clouds and cloud technologies," *Cloud Computing*, 2010.

[5] "Mapreduce-mpi library." `http://mapreduce.sandia.gov/`. Accessed: 2013-07-11.

[6] "Jlich supercomputing centre." `www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html`. Accessed: 2013-07-11.

[7] "High performance computing center stuttgart." `www.hlrs.de/systems/platforms/cray-xe6-hermit/`. Accessed: 2013-07-11.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, (New York, NY, USA), pp. 29–43, ACM, 2003.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, (Berkeley, CA, USA), pp. 15–15, USENIX Association, 2006.

[10] "Mapreduce-disco project." `http://discoproject.org/`. Accessed: 2013-07-11.

[11] "Apache hadoop." `http://hadoop.apache.org/`. Accessed: 2013-07-11.

[12] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 2010.

[13] J. Lin and C. Dyer, "Data-intensive text processing with mapreduce," in *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, NAACL-Tutorials '09, (Stroudsburg, PA, USA), pp. 1–2, Association for Computational Linguistics, 2009.

[14] L. felipe Cabrera and D. D. E. Long, "Swift: Using distributed disk striping to provide high i/o data rates," in *In Fall 1991 USENIX*, 1991.

[15] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Trans. Comput. Syst.*, vol. 6, pp. 51–81, Feb. 1988.

[16] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST*, pp. 231–244, 2002.

[17] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: a scalable distributed file system," *SIGOPS Oper. Syst. Rev.*, vol. 31, pp. 224–237, Oct. 1997.

[18] "Apache foundation." `http://www.apache.org/`. Accessed: 2013-07-11.

[19] P. Schwan, "Lustre: Building a file system for 1,000-node clusters," p. 9, 2003.

[20] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *Proceedings of the 16th international conference on World Wide Web*, WWW '07, (New York, NY, USA), pp. 131–140, ACM, 2007.

[21] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, STOC '02, (New York, NY, USA), pp. 380–388, ACM, 2002.

[22] R. Fagin, R. Kumar, and D. Sivakumar, "Efficient similarity search and classification via rank aggregation," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, (New York, NY, USA), pp. 301–312, ACM, 2003.

[23] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, (San Francisco, CA, USA), pp. 518–529, Morgan Kaufmann Publishers Inc., 1999.

[24] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[25] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2011.

[26] "Stanford university large network dataset collection." `http://snap.stanford.edu/data/`. Accessed: 2013-07-11.

[27] P. Boldi, M. Santini, and S. Vigna, "A large time-aware graph," *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008.