Scalable Scientific Computing Algorithms Using MapReduce

by

Jingen Xiang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

Abstract

Cloud computing systems, like MapReduce and Pregel, provide a scalable and fault tolerant environment for running computations at massive scale. However, these systems are designed primarily for data intensive computational tasks, while a large class of problems in scientific computing and business analytics are computationally intensive (i.e., they require a lot of CPU in addition to I/O). In this thesis, we investigate the use of cloud computing systems, in particular MapReduce, for computationally intensive problems, focusing on two classic problems that arise in scientific computing and also in analytics: maximum clique and matrix inversion.

The key contribution that enables us to effectively use MapReduce to solve the maximum clique problem on dense graphs is a recursive partitioning method that partitions the graph into several subgraphs of similar size and running time complexity. After partitioning, the maximum cliques of the different partitions can be computed independently, and the computation is sped up using a branch and bound method. Our experiments show that our approach leads to good scalability, which is unachievable by other partitioning methods since they result in partitions of different sizes and hence lead to load imbalance. Our method is more scalable than an MPI algorithm, and is simpler and more fault tolerant.

For the matrix inversion problem, we show that a recursive block LU decomposition allows us to effectively compute in parallel both the lower triangular (L) and upper triangular (U) matrices using MapReduce. After computing the L and U matrices, their inverses are computed using MapReduce. The inverse of the original matrix, which is the product of the inverses of the L and U matrices, is also obtained using MapReduce. Our technique is the first matrix inversion technique that uses MapReduce. We show experimentally that our technique has good scalability, and it is simpler and more fault tolerant than MPI implementations such as ScaLAPACK.

Acknowledgements

I would like to express the deepest appreciation to my supervisor Professor Ashraf Aboulnaga, who patiently guided me, supported me and helped me throughout my years' academic life at University of Waterloo. Without his guidance and persistent help, this thesis would not have been possible.

I would also like to thank my thesis readers, Professors Hans De Sterck and Grant Weddell.

I give my many thanks to my colleagues, coauthors, and friends at University of Waterloo, in particular Cong Guo and Huangdong Meng, for their help.

I also thank my wife and my parents for their love, understanding, and support.

Dedication

This thesis is dedicated to my two lovely sons: Joseph and Jonathan.

Table of Contents

Chapter 1

Introduction

Ever since the computer was invented, scientific computing, also called computational science, plays a vital role in most kinds of scientific activities. Many algorithms or software packages, such as Matlab, Octave, and Maple, can run on a single computer or on a parallel computing cluster with multiple computers. Many of the scientific computing implementations for clusters rely on parallel programming frameworks such as the *message passing interface* (MPI) [27, 48]. As the size of computer clusters increases to thousands or even tens of thousands of *nodes* (i.e., computers), traditional parallel programming frameworks such as MPI run into limitations due to lack of scalability and fault tolerance.

In contrast, cloud computing systems such as *MapReduce* [19], which is our focus in this thesis, are designed with scalability and fault tolerance as primary design objectives. The MapReduce model is, however, mainly designed for data intensive computational tasks, while most of scientific computing is compute intensive. Our goal in this thesis is to explore the use of MapReduce as a framework for solving scientific computing problems. We focus in particular on two problems: finding the maximum clique of a graph and matrix inversion. In addition, we present an overview of the solution to a third problem: finding the eigenvalues and eigenvectors of a matrix. By using MapReduce for these problems, we benefit from the scalability, fault tolerance, broad adoption, simple programming model, and rich software ecosystem that MapReduce provides.

In this chapter, we first briefly introduce the idea of scientific computing in Section 1.1, as well as the problems in scientific computing that we focus on in this thesis: the maximum clique problem, the matrix inversion problem, and the matrix eigenvalues and eigenvectors problem. Cloud computing is discussed in Section 1.2 and we introduce the MapReduce model in Section 1.3. Hadoop, which is an open source implementation of

MapReduce, is introduced in Section 1.4.

1.1 Scientific Computing

Scientific computing is concerned with analyzing and solving scientific problems using computers. Scientific computing tasks include numerical simulations, model fitting and data analysis, computational optimization, among others. Many software packages such as Matlab, Mathematica, SciLab, GNU Octave, Python with SciPy, and ScaLAPACK have been developed to solve scientific computing problems. However, most of these packages run on a single computer, or on a cluster using frameworks such as MPI that do not offer scalability and fault tolerance.

In this thesis, we present algorithms to solve important scientific computing problems, namely the maximum clique problem, the matrix inversion problem, and the matrix eigenvalue and eigenvector problem, using the MapReduce cloud computing framework. Our results demonstrate that one can indeed use cloud computing to solve scientific problems with good scalability and fault tolerance.

Next, we introduce the maximum clique problem and briefly review its applications in various fields of scientific research. The matrix inversion problem is then introduced, followed by the eigenvalue and eigenvector problem.

1.1.1 The Maximum Clique Problem

A clique is a complete graph, that is, a graph where any two vertices are connected by an edge. The maximum clique within a graph G is the largest size subgraph of G that is a clique. We focus on the maximum clique problem because it is a computationally hard problem with practical applications.

We motivate our work by presenting some applications of the maximum clique problem. A key motivating application for us comes from the area of coding theory, and in particular quantum error correcting codes. A code $\mathcal{C}$ of length n is a subset of all n-bit strings. The Hamming distance $d(\mathbf{x}, \mathbf{y})$ between two n-bit strings $\mathbf{x}$ and $\mathbf{y}$ is the number of positions in which $\mathbf{x}$ and $\mathbf{y}$ differ. The (minimum) distance d of a code $\mathcal{C}$ is the smallest distance between distinct codewords, which is proportional to the number of errors that the code can correct. For a given n and d, finding the largest possible code cardinality K (i.e., the number of codewords in the code) is then a maximum clique problem [39, 49].

The corresponding graph, called the coding graph, contains 2^n vertices where each vertex corresponds to an n-bit string. The edges are given by the pairs of strings $(\mathbf{x}, \mathbf{y})$ with Hamming distance greater than or equal to d. The vertices of the maximum clique of this graph represent the codewords, and the number of vertices of the maximum clique is the cardinality K. The case for quantum error-correcting codes is more complicated, but for a large class of codes, the problem can be reduced to a maximum clique problem similar to the classical case by modifying the definition of distance d using a framework known as *codeword stabilized (CWS) quantum codes* [11, 12]. In our experiments, we show orders of magnitude improvement in the time required to compute the maximum clique for a real coding graph from a quantum error correcting code.

Another application of the maximum clique problem is in the study of Internet topology. The Internet consists of several independent *autonomous systems (ASes)*. An AS is a set of routers within a single administrative domain (e.g., an ISP, a company, or a university). An AS can be related to another AS in one of three ways: *provider-customer*, where one AS provides transit service to the other AS (e.g., an ISP providing service to its customer, who may be a smaller ISP), *peering*, where the two ASes exchange traffic to their customers for free, and *sibling*, where the two ASes exchange traffic freely without any limitation. Knowing how ASes are related to each other is useful for many network research questions such as predicting communication latency, selecting peering partners, and defending against attacks known as BGP prefix hijacking. However, AS relationship information is not public, so it must be inferred from information about the routing paths taken by different packets. Each network packet (BGP packet to be more precise) that arrives at a destination contains information about its *AS path*, the set of ASes that it traversed to get from its source to its destination. An assumption we can make about valid AS paths is that they follow the *valley-free model*. This model states that after traversing a provider-to-customer or peer-to-peer link, an AS path cannot traverse a customer-to-provider or peer-to-peer link [28]. According to the valley-free model, there is at most one peer-to-peer link in an AS path. If we are given a set of AS paths that are assumed to follow the valley-free model, we can find the maximum set of peer-to-peer links by solving the following maximum clique problem. Construct a graph with a vertex for every AS-to-AS link that appears in any AS path. If two AS-to-AS links never appear in the same AS path, add an edge between their corresponding vertices. Since a valid AS path contains at most one peer-to-peer link, we can assume that the maximum clique in this graph is the maximum set of peer-to-peer links [22].

The maximum clique problem also arises in the area of social networking. In the graph of a social network, vertices represent actors, and edges represent connections between actors. A clique is a sub-structure of a network in which actors are more strongly tied to each other than they are to other members of the network. The graph of a social network can become very dense. Take the recent study of the spread of steroids in baseball [43] as an example. As players change teams, new links are formed while old ties are still maintained. The graph becomes densely interconnected quickly, so finding the maximum clique becomes time-consuming.

In biochemistry, maximum clique is used to find the largest possible pharmacophore (a group of atoms responsible for an interaction) for a pair of 3D molecules. Each of the two molecules is represented by a graph with atoms as vertices. A correspondence graph G is constructed for the two graphs of the molecules G_1 and G_2, such that G has a vertex for each pair of vertices with one from G_1 and one from G_2. Two correspondence graph vertices $\{G_1(X), G_2(M)\}$ and $\{G_1(Y), G_2(N)\}$ are adjacent in G if there is an edge from $G_1(X)$ to $G_1(Y)$ in G_1 and an edge from $G_2(M)$ to $G_2(N)$ in G_2. Matching 3D molecular structures is equivalent to the maximum clique problem in G [29].

From these examples, we can see that the maximum clique problem has many applications in different scientific domains, which motivates our focus on this problem. Next, we turn our attention to the matrix inversion problem.

1.1.2 The Matrix Inversion Problem

Matrix operations are a fundamental building block of many computational tasks in diverse fields including machine learning, data mining, physics, bioinformatics, scientific computing, and many others. In most of these fields, there is a need to scale to large matrices to obtain higher fidelity and better results (e.g., running a simulation on a finer grid, or training a machine learning model with more data). To scale to large matrices, it is important to design efficient parallel algorithms for matrix operations, and using MapReduce is one way to achieve this goal. There has been prior work on implementing matrix operations in MapReduce (e.g., [31], and also refer to the brief introduction in the following sections), but that work does not handle matrix inversion even though matrix inversion is a very fundamental matrix operation. Matrix inversion is difficult to implement in MapReduce because each element in the inverse of a matrix depends on multiple elements in the input matrix, so the computation is not easily partitionable as required by the MapReduce programming model. In this thesis, we address this problem and design a novel partitionable matrix inversion technique that is suitable for MapReduce. We implement our technique in Hadoop, and develop several optimizations as part of this implementation.

Before we present our technique, we further motivate the importance of matrix inversion by presenting some of its applications in different fields. A key application of matrix inversion is solving systems of linear equations. Suppose that we want to solve the equation $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an $n \times n$ matrix, and $\mathbf{x}$ and $\mathbf{b}$ are both vectors with n elements. One method is to multiply both sides of the equation on the left by the the matrix inverse $\mathbf{A}^{-1}$, to get $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

Matrix inversion is also widely used in computed tomography (CT). In CT, the relationship between the original image of the material ($\mathbf{S}$) and the image ($\mathbf{T}$) detected by the detector can be written as: $\mathbf{T} = \mathbf{MS}$, where $\mathbf{M}$ is the projection matrix [76]. In order to reconstruct the original image, we can simply invert the projection matrix and calculate the product of $\mathbf{M}^{-1}$ and $\mathbf{T}$. As the accuracy of the detector increases, the number of image pixels increases and hence the order of the projection matrix ($\mathbf{M}$) also increases, motivating the need for scalable matrix inversion techniques. Image reconstruction using matrix inversion can also be found in other fields such as astrophysics [79].

In the field of numerical optimization, matrix inversion is used in the Newton method for high dimensional convex optimization. In this method, a sequence of approximate solutions $\mathbf{X}_n$ is computed as follows: $\mathbf{X}_{n+1} = \mathbf{X}_n - [Hf(\mathbf{X}_n)]^{-1}\Delta f(\mathbf{X}_n)$, where $Hf(\mathbf{X}_n)$ is the Hessian matrix [54]. As is clear from the equation, matrix inversion is a key component in this numerical method.

In bioinformatics, matrix inversion is used to solve the problem of protein structure prediction [52]. A scalable matrix inversion technique would enable novel insights into the evolutionary dynamics of sequence variation and protein folding.

These are but a few of the numerous applications that rely on matrix inversion. In some cases, it may be possible to avoid matrix inversion by using alternate numerical methods (e.g., pseudo-Newton methods for convex optimization), but it is clear that a scalable and efficient matrix inversion technique such as the one we present would be highly useful in many applications. The fact that our technique uses MapReduce and Hadoop means that it benefits from the scalability, fault tolerance, and wide adoption of MapReduce, in addition to being part of the rich software ecosystem of Hadoop (e.g., HDFS, Pig, and Mahout).

1.1.3 The Eigenvalue and Eigenvector Problem

Matrix inversion can also be used in solving the problem of finding the eigenvalues and eigenvectors of a matrix. An eigenvector of a square matrix is a non-zero vector that, when multiplied by the matrix, yields a vector that differs from the original vector at most by a multiplicative scalar. Finding eigenvectors is a very important mathematical computation that is used in many areas, including the Schrödinger equation in quantum mechanics, molecular orbitals, geology and glaciology, principal components analysis, vibration analysis, eigenfaces, and tensors of moments of inertia. Therefore, it is extremely useful to find an efficient solution to the eigenvector problem. Our matrix inversion technique can help to find the eigenvectors of a given large scale matrix very efficiently, using, for example the inverse power method [71] or preconditioning (a preconditioner matrix for $\mathbf{Ax} = \mathbf{b}$ can be almost $\mathbf{A}^{-1}$). In addition, matrix inversion can be used to find a preconditioner for partial differential equations (PDE), such as the Navier-Stokes equations [60].

1.2 Cloud Computing

Cloud computing, in which computing is provided as a service, has become quite popular in recent years. Cloud computing enables users of computing to elastically grow and shrink their computing infrastructure as the load increases and decreases. Cloud computing also enables unprecedented levels of scalability, facilitated by computing frameworks such as MapReduce, which is our focus in this thesis.

There are many companies that provide cloud computing services, such as the Elastic Compute Cloud (EC2) that is part of Amazon Web Services (AWS), the Azure cloud from Microsoft, and Google App Engine. In this thesis we use Amazon EC2 [25] for all our experiments, since it offers a high degree of scalability using a *pay-as-you-go* pricing model.

1.3 MapReduce

MapReduce is a parallel dataflow programming and execution framework introduced by Google in 2004 [18, 19]. It supports performing distributed computing on a large cloud. MapReduce separates the computation into two phases: (1) the *map* phase, where a mapper task runs on each node in a cluster and applies a map function to records read from *splits* of input files, then emits intermediate (*key, value*) pairs for each record processed, and (2) the *reduce* phase, where a reducer task runs each node and applies a reduce function to all intermediate (*key, value*) pairs that have the same key, emitting zero or more output (*key, value*) pairs. Between the map phase and the reduce phase, the (*key, value*) pairs emitted from map functions are shuffled to different reducer tasks nodes based on the hash value of the map output key. This shuffling guarantees that the (*key, value*) pairs with the same key will be sent to the same reducer task. The MapReduce framework is suitable for computations that have little or no communication between the compute nodes. The

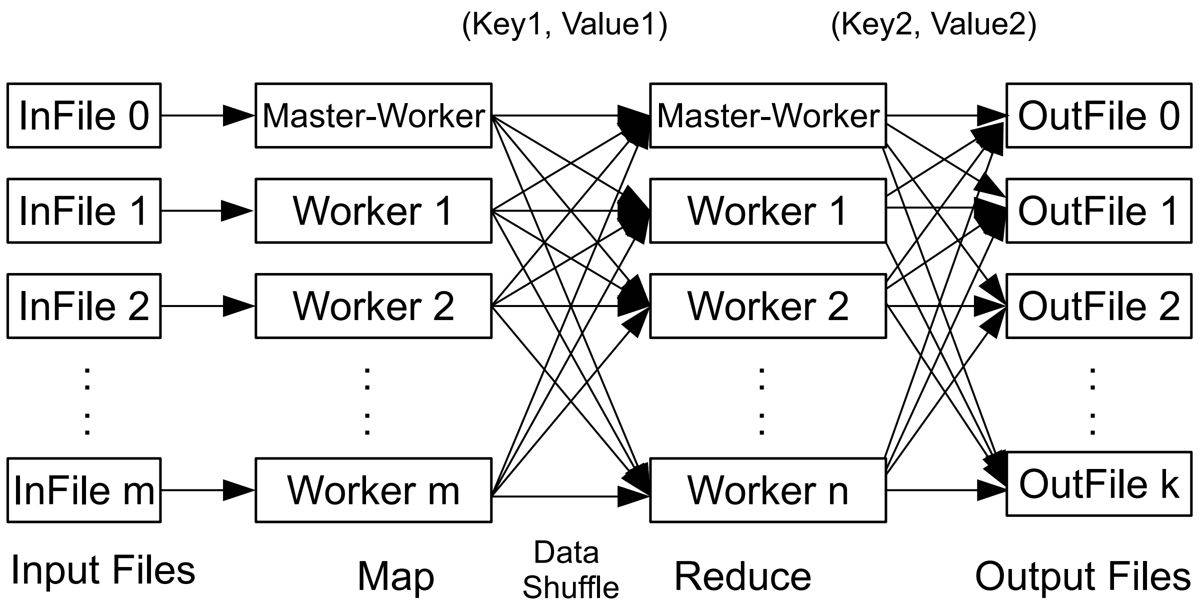


Figure 1.1: MapReduce framework.

framework provides good fault tolerance such that it can be easily used on a very large cluster, up to thousands of nodes. The MapReduce framework is shown in Figure 1.1, where $m+1$ splits of files are read into $m+1$ compute nodes in the map phase, and the results produced by these $m+1$ nodes are emitted to $n+1$ nodes (the reduce phase) based on the hash value of Key1, and finally the reduce function writes the result into $k+1$ files based on the hash value of Key2.

1.4 Hadoop

Hadoop is an open source implementation of Google's MapReduce framework. Hadoop supports programs written in Java and can be extended to support programs written in other languages such as C or Python. In this thesis, we use Hadoop and all implementation is done in the Java language.

In addition to the MapReduce computation framework, the Hadoop platform encompasses a file system, named *HDFS* (*Hadoop Distributed File System*), which provides scalability and fault tolerance for data storage. The HDFS file system is designed to run on low-cost commodity hardware and provides high throughput access to the data [37]. Hadoop is also the basis of a rich ecosystem that encompasses scripting languages like Pig Latin and Hive, storage systems like HBase and Cassandra, and other libraries and software systems like Mahout for machine learning and Zookeper for coordination. Hadoop is an easy-to-use, popular, well understood framework, which is part of the motivation for using it as the basis of our implementation in this thesis.

The rest of the thesis is organized as follows. In Chapter 2, we describe the details of our maximum clique algorithm. In Chapter 3, we describe our matrix inversion technique. Chapter 4 discusses extending our matrix inversion technique to the problem of finding eigenvalues and eigenvectors. Chapter 5 presents our conclusions.

Chapter 2

Scalable Maximum Clique Computation Using MapReduce

In this chapter, we present the details of our technique for finding maximum clique in a graph. This work appeared in [78].

A clique is a complete graph, in which any two vertices are connected by an edge. As mention in Chapter 1, the maximum clique problem is an important problem in scientific computing. The main idea of our solution is to recursively partition the input graph into smaller possibly overlapping subgraphs so that each compute node in the MapReduce cluster can independently compute the maximum clique for its partition. When two vertices connected by an edge are placed in two different partitions, one of the vertices has to be replicated in the other's partition so that the edge is preserved (hence the overlapping partitions). Graph partitioning has been studied in previous work, but partitioning for the maximum clique problem is challenging for two reasons: (1) Since any edge can be part of the maximum clique, there is no notion of a workload that dictates "important" edges that should not be split across partitions. Some partitioning methods require such a workload-driven measure of edge importance [80]. (2) Maximum clique is of interest on *dense graphs*, where a significant fraction of the edges that can be present in a graph are actually present. Most general purpose graph partitioning algorithms do not work well for dense graphs, since they end up replicating a large fraction of the vertices (as we will demonstrate in Section 2.7).

In this chapter, we develop a graph partitioning algorithm that addresses these two challenges and is targeted towards computing maximum cliques. Our partitioning algorithm removes one vertex at a time from the graph and puts the subgraph consisting of this vertex plus all its neighbors in one partition. By repeatedly partitioning the graph, we end up with multiple subgraphs on which the maximum clique can be computed independently. The maximum clique of the original graph is the largest of the maximum cliques computed on the partitions.

Partitioning alone does not result in a scalable maximum clique algorithm, since we still need to compute the maximum clique for all partitions. To avoid this brute-force effort, we use a *branch and bound* approach to avoid computing the maximum clique of a partition if it can be shown that its maximum clique will be smaller than the largest clique found so far. For this pruning to be effective, we need to find large cliques as early as possible. At the same time, we need the partitions to be of similar size for load balancing. Our partitioning method, which we call *Balanced Multi-depth Color-based (BMC)* partitioning, achieves these objectives. It uses graph coloring to compute an upper bound on the clique size and to heuristically order the partitions so that pruning can be effective. It partitions the graph recursively to multiple depths to achieve load balancing.

After the graph is partitioned, the maximum cliques for the different partitions can be computed independently on different nodes of a computing cluster, with some pruning based on the size of the largest clique found so far. In this thesis, we implement this parallel branch and bound computation using MapReduce, and in particular Hadoop. It is important to note that this approach to computing maximum cliques can be implemented using other cluster-based computing platforms, such as Pregel or even traditional Message Passing Interface (MPI) [35]. We use Hadoop because of its simple programming model, scalability, and fault tolerance, characteristics that are not available with, say, MPI. Moreover, Hadoop is popular and widely available, so a Hadoop-based maximum clique implementation would be easy for users to adopt and deploy on private clouds or public clouds such as Amazon EC2 [25]. Furthermore, such an implementation would work seamlessly with other tools and technologies in the Hadoop ecosystem.

To illustrate the power of our solution, we consider the DIMACS Maximum Clique Benchmark [20]. Maximum clique was part of the Second DIMACS Implementation Challenge [21], and the DIMACS Maximum Clique Benchmark consists of a set of graphs for which finding the maximum clique is difficult. Despite being well studied, the maximum clique for some of these graphs was still not known. We ran our algorithm on one such graph with 4000 vertices (C4000.5) on 128 Amazon EC2 high CPU medium instances, and completely solved the maximum clique problem for this graph for the first time in 39 hours. We estimate that a serial solution on a single CPU would have taken more than 1 year. More details about this experiment are presented in Section 2.7.

Note that while this is the largest graph that we used in our experiments, it is still only a few tens of megabytes in size. The complexity of maximum clique stems not from the size of the data but rather from the CPU time required to compute the maximum clique on individual partitions. The key factor determining complexity is the density of the graph not its size. Thus, our work represents an interesting study in using MapReduce for CPU-intensive parallel computation, in contrast to the data intensive computation for which MapReduce is typically used.

The rest of this chapter is organized as follows: in Section 2.1, we present preliminary information about the maximum clique problem. Section 2.2 presents related work, and Section 2.3 discusses single-node maximum clique algorithms. In Section 2.4, we present different graph partitioning techniques and introduce our proposed BMC partitioning. Section 2.5 describes our approach to running time estimation, which is required for BMC partitioning. The MapReduce implementation of our algorithm is discussed in Section 2.6. Section 2.7 presents an experimental evaluation and Section 2.8 concludes.

2.1 Preliminaries and Notation

In this thesis, we focus on undirected unweighted graphs. An undirected graph G is represented by an ordered pair (V, E), where V is a finite set of vertices and E is a set of two-element subsets of vertices called edges. For a graph G, we write $V(G)$ and $E(G)$ for the vertex set and the edge set, respectively. The number of vertices of a graph, i.e., the cardinality of $V(G)$, denoted by $|V(G)|$ or $|G|$ for simplicity, is called the *order* of the graph. The number of edges of a graph, denoted by $|E(G)|$, is called the *size* of the graph.

Two vertices $a, b \in V(G)$ are called adjacent if $(a, b) \in E(G)$ (which implies $(b, a) \in E(G)$ for undirected graph). A graph G is called *complete* if all its vertices are pairwise adjacent. For a set of vertices $S \subseteq V(G)$, $S(G) = (S, E(G) \cap (S \times S))$ is the subgraph induced by S. A clique C is a subset of V such that the induced subgraph $C(G)$ is complete. A k-clique of G is a clique in G of order k. A clique is called *maximal* if it cannot be extended by including one more adjacent vertex. In other words, a maximal clique does not exist exclusively within the vertex set of a larger clique. For a given graph G, the maximum cliques, denoted as $MaxClique(G)$, are cliques of the largest possible order in G. The order of a maximum clique in G is denoted by $\omega(G)$.

We now turn our attention to graph partitioning. The neighborhood of vertex v in G, denoted by $N(G, v)$, is a subgraph of G consisting of the adjacent vertices to vertex v. That is, $V(N(G, v)) = \{w | (v, w) \in E(G)\}$ and $E(N(G, v)) = \{(v, w) | v, w \in V(N(G, v)),\ (v, w) \in E(G)\}$. The degree of a vertex v in G, denoted by $D(G, v)$, is

Symbol	Definition
$G = (V, E)$	Graph with vertices V and edges E
$V(G)$	All vertices of G
$E(G)$	All edges of G
$\|G\|$ or $\|V(G)\|$	Number of vertices in G
$\|E(G)\|$	Number of edges in G
$G - \{v\}$	Subgraph of G obtained by removing vertex v
$G \cup \{v\}$	Adding the vertex $v \in G' - G$ to $G \subseteq G'$
$MaxClique(G)$	Maximum clique of G
$\omega(G)$	Number of vertices in the maximum clique of G, $\|MaxClique(G)\|$
$N(G, v)$	Neighborhood of vertex v in G, $V = \{w \| (v, w) \in E(G)\}$ $E = \{(v, w) \| v, w \in V,\ (v, w) \in E(G)\}$
$D(G, v)$	Degree of v in G, $\|N(G, v)\|$
$Ext\_D(G, v)$	Extension degree of v in G, $\sum\limits_{w \in N(G,v)} D(G, w)$
$I(G, G_1, G_2)$	Inter-graph induced by subgraphs G_1, G_2 of G
C_G	Coloring of G (each vertex has a color number)
$C_G[v]$	Color of vertex v
ρ, or $\rho(G)$	Density of G, $\frac{2\|E(G)\|}{\|G\|(\|G\|-1)}$

Table 2.1: Notation used in this chapter.

the number of adjacent vertices of v in G, i.e., $D(G, v) = |N(G, v)|$, while the extension degree of a vertex v in G, denoted by $Ext\_D(G, v)$, is the sum of the degree of its adjacent vertices in G, i.e., $Ext\_D(G, v) = \sum_{w \in N(G,v)} D(G, w)$.

Consider a graph G which is a subgraph of G'. For any $v \in V(G') - V(G)$, the union $G \cup \{v\}$ is a subgraph of G' induced by adding a vertex v to $V(G)$, i.e., $V(G \cup \{v\}) = V(G) \cup \{v\}$ and $\forall (v, w) \in E(G')$, $(v, w) \in E(G \cup \{v\})$ (i.e., $E(G \cup \{v\}) = \{(a, b) | (a, b) \in E(G'), \forall a \in V(G \cup \{v\}), \forall b \in V(G \cup \{v\}), a \neq b\}$).

For two subgraphs of G, G_1 and G_2, where $V(G_1) \cap V(G_2) = \{\}$, the *inter-graph* of G_1 and G_2 is a subgraph of G, denoted by $I(G, G_1, G_2)$. The vertex set of $I(G, G_1, G_2)$, $V(I)$, is the set of all vertices $v \in V(G_1)$ and $w \in V(G_2)$ such that $(v, w) \in G$. The edge set of $I(G, G_1, G_2)$, $E(I)$, is given by $E(I) = \{(v, w) | v, w \in V(I), (v, w) \in E(G)\}$.

Our partitioning algorithm relies on graph coloring. A coloring of a graph G is a list, denoted by C_G, containing for each vertex a number denoting the color of that vertex. The color of a vertex v is denoted by $C_G[v]$.

Our partitioning also relies on graph density. The density of a graph G is defined as $\rho(G) = \frac{2|E(G)|}{|G|(|G|-1)}$. For any graph G, $0 \leq \rho(G) \leq 1$, and for a complete graph $\rho(G) = 1$. We summarize the notation used in this chapter in Table 2.1.

2.2 Related Work

Ever since the term "clique" was introduced [46], the maximum clique problem has been investigated by many researchers. The k-clique decision problem asks whether there exists a k-clique in a given graph G, and is known to be NP-complete [30, 67]. Therefore, the maximum clique problem is NP-hard. If one can solve the maximum clique problem, one can also solve the k-clique decision problem by comparing the number of vertices in the maximum clique to k.

A common and effective method to find the maximum clique is to use branch and bound search. A classical algorithm proposed by Carraghan and Pardalos [10] is a straightforward heuristic and pruning algorithm. At every stage of the algorithm, there is a global control parameter ω, which is the largest known clique at this stage. For each subgraph G', the algorithm finds the set of vertices $\{w\}$ which are not in G' but are connected to all vertices in G'. In addition, any two vertices in $\{w\}$ are connected to each other. Denote by m the number of vertices in $\{w\}$. If $m + |G'| \leq \omega$, the subgraph G' is pruned. This algorithm is shown to be much more effective than a brute-force exhaustive search and is easy to parallelize.

Another maximum clique algorithm is proposed by Östergård [56]. That paper introduces an additional pruning strategy based on the following observation. Given a graph $G = (V, E)$ containing vertices $\{v_1, v_2, \ldots, v_{|V|}\}$, we can sort the vertices in some order and let subgraph $G_i = \{v_i, v_{i+1}, \ldots, v_{|V|}\}$. Observe that $\omega(G_i) = \omega(G_{i+1}) + 1$ if vertex v_i is included in the maximum clique of G_i, and $\omega(G_i) = \omega(G_{i+1})$ if vertex v_i is not included in the maximum clique of G_i. The algorithm uses a search strategy similar to [10], and adds a pruning rule based on the above observation.

An effective algorithm named *MCR* is proposed by Tomita and Kameda [69]. This algorithm uses graph coloring to obtain an upper bound on the size of the maximum clique, which achieves better branch pruning than previous algorithms. We use the *MCR* algorithm as our single-node maximum clique algorithm, and discuss it in more detail in the next section.

Wang and Cheng [75] recently suggested using the size of the maximum k-truss of a graph as an upper bound on the size of the maximum clique, so that some subgraphs can be more effectively pruned. The k-truss of a graph G is the largest subgraph of G in which every edge is contained in at least $(k-2)$ triangles within the subgraph. Finding the k-truss of a graph can be solved in polynomial time.

If a graph could be represented as a combination of a random graph of order n and a large clique of order larger than $\sqrt{n}$ (a "planted clique"), the maximum clique can be found with high probability using spectral methods [3]. In this thesis, we encounter maximum cliques whose orders are smaller than $\sqrt{n}$, so spectral methods cannot be used even if the planted clique assumption holds.

There have been previous attempts to solve the maximum clique problem on computer clusters. A parallel maximum clique algorithm based on MPI has been proposed in [58]. This algorithm uses a master-worker architecture where the master uses graph partitioning to assign a separate task to each worker. Once a worker has finished the task, the master will assign another subgraph to it. The partitioning strategy used by this algorithm is a simple version of our one-depth partitioning strategy. This strategy balances the load among workers in many cases, but it is not sufficient when the density of the graph is large. For dense graphs, the first few subgraphs generated by the partitioning are likely to be larger than others and will therefore take much longer to finish. The time taken for one of these large subgraphs can be larger than the total time of the smaller subgraphs. We show in our experiments that this limits the scalability of the approach in [58], even if we use an improved single-node maximum clique algorithm. In addition, MPI is a complex programming model, does not provide fault tolerance, and is not as scalable as MapReduce.

MapReduce solutions for the maximum clique problem have been proposed before. Lin et al. [45] proposed a MapReduce algorithm for maximum clique, but their partitioning strategy is naive random partitioning, and they do not employ good pruning. At best, their algorithm would perform similar to our one-depth partitioning strategy, which is much less scalable than our multi-depth partitioning strategy. In practice, it would be difficult for their algorithm to match our one-depth performance since they employ a naive random partitioning strategy with no partition optimization at all and also no pruning optimization. Wu et al. [77] also proposed a MapReduce algorithm for the clique problem, but they focus on enumerating all maximal cliques instead of the maximum clique problem. Moreover, their algorithm shows poor scalability.

It may be possible to use a system dedicated to graph processing for solving the maximum clique problem. One such system is Pregel [51], and its open source implementation based on Hadoop called Giraph [32]. Pregel implements a computational model called *bulk synchronous parallel computation* [72], which is targeted towards iterative computation on graphs. Our partitioning algorithm results in subgraphs that are completely independent, so iteration on the graph vertices is not required. What is needed is independently computing maximum clique on the individual partitions, for which MapReduce is suitable.

2.3 Maximum Clique Algorithm on a Single Node

Our approach partitions a graph into subgraphs, and can use any single-node algorithm to compute the maximum clique on one partition. We experimented with different single-node algorithms and found that the MCR algorithm [69] gives the best performance. Therefore, we use it as our single-node algorithm. Furthermore, the graph coloring that is used by MCR to order vertices is the basis for our color-based partitioning. Hence, we describe the MCR algorithm in some detail in this section.

The MCR algorithm is a branch and bound algorithm. Given a graph $G = (V, E)$, if $\omega(G)$ is the number of vertices in the maximum clique of G, then $\omega(G)$ can be obtained by independently computing the maximum cliques of different independent subgraphs as follows:

$$\begin{aligned}
&\omega(N(G, v_1)) + 1,\\
&\omega(N(G - \{v_1\}, v_2)) + 1,\\
&\omega(N(G - \{v_1, v_2\}, v_3)) + 1,\\
&\ldots,\\
&\omega(N(G - \{v_1, v_2, \ldots, v_k\}, v_{k+1})) + 1,\\
&\ldots,\\
&\omega(N(G - \{v_1, v_2, \ldots, v_{|V|-2}\}, v_{|V|-1})) + 1
\end{aligned}$$

To compute the maximum clique of $N(G, v_1)$, we can perform the same process on the subgraph $N(G, v_1)$. Thus, the maximum clique can be found in a recursive way. Without a pruning strategy, this algorithm is equivalent to brute-force exhaustive search. Therefore, it is important to prune the search based on bounds on the size of the maximum clique.

The MCR algorithm proposes a pruning strategy based on coloring the vertices of the graph. In this method, each vertex in the graph is given a color, and any two adjacent vertices cannot have the same color. Since no adjacent vertices can have the same color, a clique will always consist of vertices of different colors. Thus, the total number of colors used to color the graph is an upper bound on the size of the maximum clique. Using fewer colors results in a tighter upper bound, but the coloring procedure becomes more complex. At one extreme, we can use a different color for each vertex, in which case the coloring procedure is very simple but has no pruning power. At the other extreme, we can use the minimum possible number of colors, which increases the pruning power of coloring but makes the coloring procedure very complex.

A coloring procedure is given in [69] that strikes a balance between coloring complexity and pruning power. The coloring procedure starts by sorting the vertices of the graph in descending order according to their degree. The procedure for doing this is as follows: (1) Select a vertex v with the minimum degree from the graph G and append it to R (which is initially an empty list). If there are k (> 1) vertices with the same degree (denote this set of vertices as $G' = \{v_1, v_2, \ldots, v_k\}$), select the vertex with the minimum extension degree in G'. (2) Remove v from G. When removing v, we also remove all edges incident on v, which changes the degrees of the other vertices in G and hence may change the vertex with the minimum degree chosen in the next step. (3) Repeat Step (1) and (2) until there are no vertices in G. (4) Reverse the list R.

After sorting, the procedure shown in Algorithm 1 is applied. Each color is represented by a number, with 1 being the first color used. The last step in Algorithm 1 sorts the vertices by color in descending order. Sorting the vertices by degree before coloring and then sorting the colors in descending order is shown to improve the effectiveness of coloring. We return to this coloring algorithm in Section 2.4.6 when we discuss color-based partitioning.

The MCR algorithm processes vertices in descending order of color as shown in Algorithm 2. The variables Q_{max} and Q are global variables recording the maximum clique and the current maximal clique, respectively. The pruning based on the maximum color is in line 5 of the algorithm.

2.4 Graph Partitioning Strategies

In order to compute the maximum clique in parallel on a computer cluster, we need to partition the graph into several smaller subgraphs and compute the maximum clique for these subgraphs independently on the compute nodes. Graph partitioning has been studied before (e.g., [2, 38, 41]). In this section, we present some possible partitioning strategies from the literature, and our proposed BMC partitioning. Our experiments show that BMC partitioning outperforms all other partitioning strategies presented in this section.

Algorithm 1 Simple graph coloring.

```
1: function Color($G$)
2: $G_1, G_2, \ldots, G_n = \{\}$
3: $i = 1$
4: while $i < |G|$ do
5:     $j = 1$
6:     while $\exists v_k \in G_j$, s.t., $v_i \in N(G, v_k)$ do
7:         $j = j + 1$
8:     end while
9:     $C_G(v_i) = j$, append $v_i$ to $G_j$
10:    $i = i + 1$
11: end while
12: Sort $V(G)$ based on $C_G$ in descending order
13: return $C_G$
```

2.4.1 Bisection Partitioning

A straightforward partitioning strategy is bisection partitioning, which partitions the original graph G into two graphs G_A and G_B that have a similar number of vertices. In order to obtain the maximum clique of G, we need to compute the maximum cliques of G_A, G_B, and the inter-graph $I(G, G_A, G_B)$ (recall the definition of the inter-graph from Section 2.1). Both subgraphs G_A and G_B can be further partitioned into three more subgraphs, and so on.

In order to reduce the number of vertices of the inter-graph, we need to minimize the number of edges connecting G_A and G_B. This converts the partitioning problem into the bisection partitioning problem, which is a well-known NP-hard problem. Some approximate bisection partitioning algorithms exist, such as the classical Kernighan and Lin algorithm [41] as well as more recent algorithms [64, 65]. However, in our experiments we found that bisection partitioning (using the Kernighan and Lin algorithm) is not suitable for the maximum clique problem because it results in large inter-graphs $I(G, G_A, G_B)$. In general, we found that the largest inter-graphs typically have the same size as the original graph, so we gain no reduction in complexity through bisection partitioning.

Algorithm 2 *MCR* for single node maximum clique.

```
function MaxClique($G$)
while $|G| > 0$ do
    /* $C_G$ is the color list of $G$ */
    $v = \{p|C_G[p] = max(C_G)\}$
    if $|Q| + C_G[v] > |Q_{max}|$ then
        $Q = Q \cup v$
        $G_1 = N(G, v)$
        if $|G_1| > 0$ then
            MaxClique($G_1$)
        else if $|Q| > |Q_{max}|$ then
            $Q_{max} = Q$
        end if
        $Q = Q - \{v\}$
    end if
    $G = G - \{v\}$
end while
return $Q_{max}$
```

2.4.2 k-way Partitioning

Another possible partitioning strategy is k-way partitioning, which partitions the graph G into k subgraphs of similar size, denoted as $\{G_1, G_2, G_3, \ldots, G_k\}$, instead of two subgraphs as in bisection partitioning. A good partitioning also minimizes the number of edges running between separated subgraphs. There are many methods for k-way partitioning, such as spectral partitioning [36] and multi-level graph partitioning [38].

In order to compute the maximum clique of G, we need to separately compute the maximum clique of graphs $G_1, G_2, G_3, \ldots, G_k$, and the maximum clique of the inter-graphs that contain edges running between different subgraphs and the vertices connected by those edges. The inter-graphs can be obtained in various ways. The straightforward way is to compute the inter-graphs using multiple steps. The i-th step produces $k/2^i$ inter-graphs, each of which is computed from 2^i subgraphs. That is, in the first step,

$$\begin{aligned} &I_{1,1} = I(G, G_1, G_2), \\ &\ldots, \\ &I_{1,k/2} = I(G, G_{k-1}, G_k), \end{aligned} \tag{2.1}$$

and in the i-th step, the m-th inter-graph is

$$I_{i,m} = I(G, \cup_{n=1}^{2^{i-1}} G_{(m-1)2^{i-1}+n}, \cup_{n=1}^{2^{i-1}} G_{m2^{i-1}+n}). \tag{2.2}$$

The last inter-graph is

$$I_{log(k),1} = I(G, G_1 \cup G_2 \cup \ldots \cup G_{k/2}, G_{k/2+1} \cup G_{k/2+2} \cup \ldots \cup G_k), \tag{2.3}$$

which is very similar to the inter-graph obtained from bisection partitioning. As with bisection partitioning, such an inter-graph generally has the same size as the original graph G.

To avoid this problem, we use another method. Here we consider the inter-graphs one by one. That is, the i-th inter-graph is $I(G, G_1 \cup G_2 \cup \ldots \cup G_{i-1}, G_i)$ ($i = 2,\ 3, \ldots, n$). Using this method, we can get smaller inter-graphs for graphs with low densities, but we still face the problem of large inter-graphs for graphs with high densities.

2.4.3 PICS Partitioning

A recently proposed partitioning strategy that is fundamentally different from bisection partitioning or k-way partitioning is PICS partitioning [2]. Instead of partitioning the graph into subgraphs with a similar number of vertices, PICS partitioning tries to find clusters in a given graph. If we consider these clusters as subgraphs, we can use PICS to partition graph G into several subgraphs (*subgroups* to use the PICS terminology) $G_1, G_2, \ldots, G_k$. In contrast to bisection partitioning and k-way partitioning, it is not necessary that these subgraphs have a similar number of vertices.

In order to compute the maximum clique of the original graph G, we need to separately compute the maximum cliques of those subgraphs and the inter-graphs, as described above. The inter-graphs can be obtained using the same method as in Section 2.4.2. If we want to partition the subgraphs into smaller subgraphs, we can apply the PICS algorithm to those subgraphs. However, we find that this algorithm cannot guarantee to partition some subgraphs into two or more smaller subgraphs. Additionally, the algorithm suffers the same problem of large inter-graphs and is not suitable for the maximum clique problem.

2.4.4 One-Depth Partitioning

Motivated by the shortcomings of existing partitioning techniques, we present a simple partitioning that we call *one-depth partitioning*, which is similar to the partitioning used by the MCR algorithm in Section 2.3. Given a graph $G = (V, E)$, the subgraphs are obtained as follows:

$$\begin{aligned}
&G_1 = v_1 \cup N(G, v_1),\\
&G_2 = v_2 \cup N(G - \{v_1\}, v_2),\\
&G_3 = v_3 \cup N(G - \{v_1, v_2\}, v_3),\\
&\ldots,\\
&G_k = v_k \cup N(G - \{v_1, v_2, \ldots, v_{k-1}\}, v_k),\\
&\ldots,\\
&G_{|V|-1} = v_{|V|-1} \cup (G - \{v_1, v_2, \ldots, v_{|V|-2}\}, v_{|V|-1})
\end{aligned} \tag{2.4}$$

This method has been used previously in the parallel computation of maximum cliques based on MPI [58] and MapReduce [45]. However, this method does not work for dense graphs because it results in poor load balancing between the partitions, since some partitions are much larger than others.

2.4.5 Multi-Depth Partitioning

Due to the load balancing problem of one-depth partitioning, we propose a novel partitioning method suitable for parallel maximum clique computation using MapReduce. First, we partition the original graph into several smaller subgraphs as in Equation 2.4. Next, we estimate the running time of the maximum clique algorithm on each partition. Estimating the running time is based on a simple experiment-driven performance model described in Section 2.5. If the estimated running time on a partition is above a bound value, we further partition that subgraph into smaller subgraphs. The bound value in this thesis is chosen through experiments to be 60 seconds (see Section 2.7.2). We repeat this partitioning step until the estimated running time on each of the partitions is below the bound value.

If the estimated running time of any subgraph in G_1, G_2, …, $G_{|V|-1}$ is larger than the bound value (let us assume it is G_k), the subgraph G_k will be partitioned again as follows:

$$\begin{aligned}
&G_{k,1} = \{v_k, w_1\} \cup N(G_k - \{v_k\}, w_1),\\
&G_{k,2} = \{v_k, w_2\} \cup N(G_k - \{v_k, w_1\}, w_2),\\
&G_{k,3} = \{v_k, w_3\} \cup N(G_k - \{v_k, w_1, w_2\}, w_3),\\
&\dots,\\
&G_{k,l} = \{v_k, w_l\} \cup N(G_k - \{v_k, w_1, \dots, w_{l-1}\}, w_l),\\
&\dots,\\
&G_{k,m} = \{v_k, w_m\} \cup N(G_k - \{v_k, w_1, \dots, w_{m-1}\}, w_m),
\end{aligned} \tag{2.5}$$

where m is a cutoff value such that the estimated running time of the remaining subgraph $G_k - \{v_k, w_1, w_2, \dots, w_{m-1}\}$ is no larger than the bound value, while the estimated running time of $G_k - \{v_k, w_1, w_2, \dots, w_{m-2}\}$ is larger than the bound value.

Again, if there is a subgraph in $G_{k,1}, G_{k,2}, \dots, G_{k,m}$ such that its estimated running time is larger than the bound value (let us assume it is $G_{k,l}$), then $G_{k,l}$ needs to be partitioned again. The n-th partitioned subgraph from $G_{k,l}$ is defined as follows:

$$G_{k,l,n} = \{v_k, w_l, u_n\} \cup N(G_{k,l} - \{v_k, w_l, u_1, u_2, \dots, u_{n-1}\}, u_n). \tag{2.6}$$

At the end of the partitioning, the estimated running time of any subgraph is less than the bound value.

In our algorithm, the partitioning path is recorded. This path indicates how the subgraph is obtained from the original graph. For example, the subgraph $G_{k,l,n}$ is obtained by the partitioning algorithm via $v_k \to w_l \to u_n$. Therefore, the maximum clique of $G_{k,l,n}$ can be computed with the call

$$\text{MaxClique}(N(G_{k,l} - \{u_1, u_2, \dots, u_{n-1}\}, u_n)) \cup \{v_k, w_l, u_n\} \tag{2.7}$$

instead of

$$\text{MaxClique}(\{v_k, w_l, u_n\} \cup N(G_{k,l} - \{u_1, u_2, \dots, u_{n-1}\}, u_n)), \tag{2.8}$$

which improves performance because the subgraph $N(G_{k,l} - \{u_1, u_2, \dots, u_{n-1}\}, u_n)$ is smaller than the subgraph $\{v_k, w_l, u_n\} \cup N(G_{k,l} - \{u_1, u_2, \dots, u_{n-1}\}, u_n)$.

2.4.6 Sorting Vertices Based on Color

As described in Section 2.3, sorting the vertices based on graph coloring can significantly reduce the running time of solving the maximum clique problem on a single node. Our experiments based on random graphs also show that, in most cases, sorting the vertices based on color before partitioning leads to fewer partitions than other sorting methods such as sorting the vertices based on order. After partitioning, the running time to find the maximum clique in each subgraph of the final partitioning does not vary much, regardless of how the vertices were sorted. And since sorting based on color leads to fewer partitions, it reduces the running time of the overall maximum clique computation.

Therefore, we heuristically propose to sort the vertices based on color before each partitioning step in the multi-depth partitioning is performed. The coloring method is the same as the one used in the MCR algorithm [69], described in Section 2.3. The vertex with the largest color is first selected, then the one with the largest color in the remaining graph is selected, and so on.

In our implementation, we sort graphs based on color in both one-depth partitioning and multi-depth partitioning. However, our recommended partitioning method is multi-depth partitioning with sorting based on color. We call this partitioning method *Balanced Multi-depth Color-based (BMC)* partitioning.

Figure 1 presents an example to illustrate BMC partitioning. In this example, we assume that the running time of a graph with 6 vertices is greater than the bound value, i.e, the bound value corresponds to 5 vertices. First, the vertices are sorted based on degree in ascending order, resulting in the sorted list $\{3, 2, 7, 6, 8, 4, 9, 1, 5\}$. Next, we use Algorithm 1 to color these vertices. The colored vertices are shown in Figure 2.1 (a). The sorted vertices based on color in descending order are $\{5, 1, 8, 6, 9, 3, 2, 7, 4\}$. Therefore, vertex 5 is first selected for the partitioning (Figure 2.1 (b)) since it has the maximum color $= 4$. Then vertex 1 is selected because it has the highest color number after vertex 5 is removed, then vertex 8 is selected after vertices $\{5, 1\}$ are removed, and so on until the remaining graph has only 5 vertices and the estimated running time is less than the bound value. But we find that the subgraph from the neighborhood of vertex 5 has 7 vertices, so it needs to be partitioned again. Before partitioning, its vertices are again sorted based on the degree in this subgraph resulting in the list $\{3, 8, 7, 6, 9, 4, 1\}$. Then this list is colored and the sorted list of vertices based on color is $\{1, 4, 7, 9, 6, 8, 3\}$. Thus, vertex 1 is first selected since it has the maximum color number and then the next selected vertex is vertex 4. After that, we find that both the subgraphs partitioned via vertex 4 and the remaining subgraph have no more than 5 vertices (Figure 2.1 (c)). The BMC partitioning is then done.

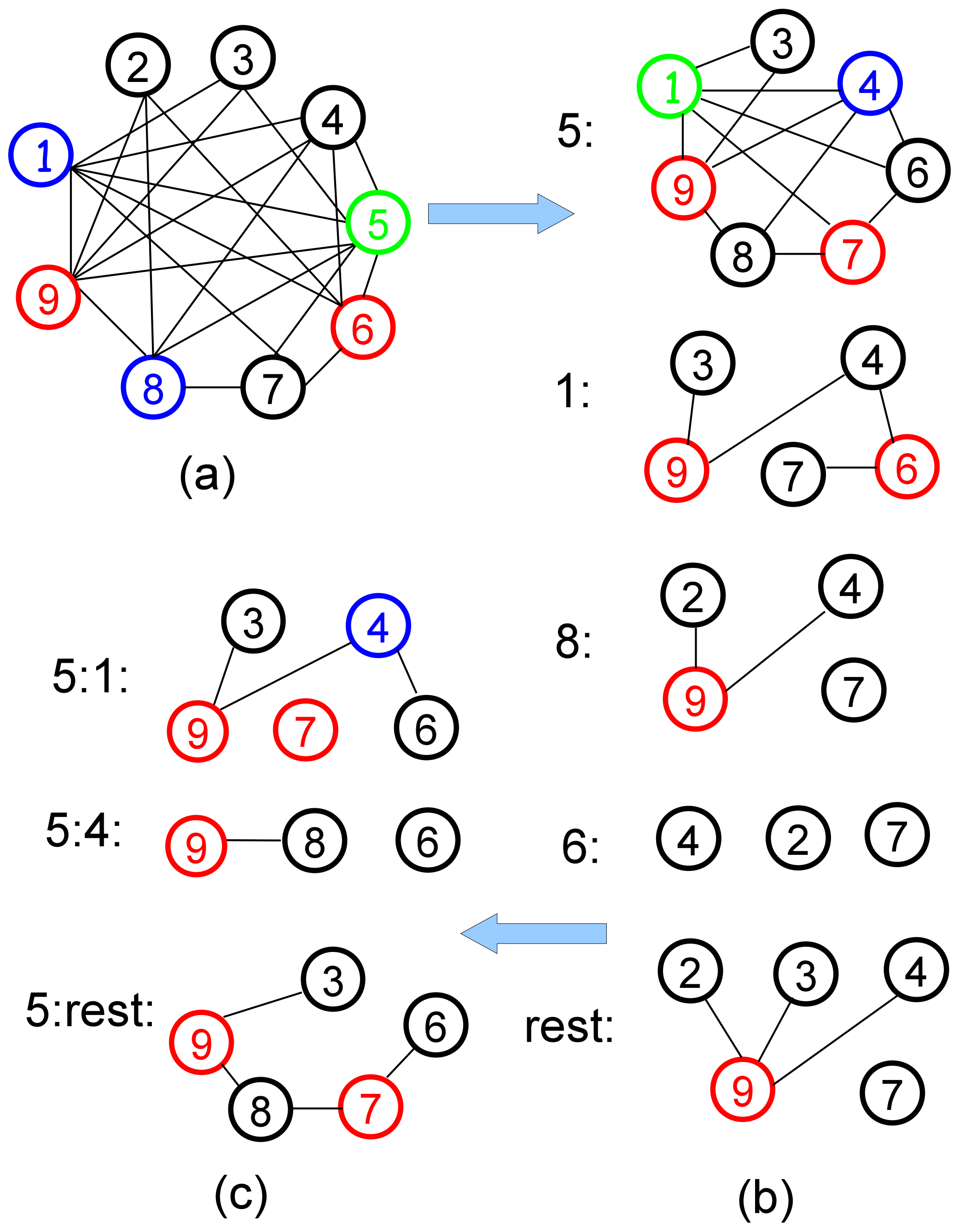


Figure 2.1: BMC partitioning. The numbers for the colors are: $black = 1$, $red = 2$, $blue = 3$, and $green = 4$.

2.5 Running Time Estimation

When we partition the graph into smaller subgraphs for better load balancing, the running time of each subgraph should be no greater than a bound value. Therefore, we need to model the relationship of the running time versus the number of vertices and the density of the graph. In this section, we present an experiment-driven model to estimate running time. The model can easily be calibrated for different computer clusters with different hardware and software configurations.

The exact relationship between running time and vertices and density is difficult to obtain. Therefore, we only aim to get an approximate relationship. First, we know that the worst case running time for a given graph G is $T(G) = O(a^{|G|})$, where a is a constant. Thus, an accurate model for running time would require an exponential function. For our model, we assume that $T(G) = f(|G|, \rho(G))^{|G|g(\rho(G))}$, where $f(|G|, \rho(G))$ and $g(\rho(G))$ are polynomial functions of the number of vertices $|G|$ and the density $\rho(G)$. We conducted experiments in which we observed that the function $g(\rho(G))$ can be modelled accurately as a quadratic function of $\rho(G)$. The Taylor series expansion of $f(|G|, \rho(G))$ gives us the following:

$$T(G) = (\sum_{i=0}^{k} |G|^i \sum_{j=0}^{k} a_{ij}\rho(G)^j)^{|G|(\rho(G)^2+b\rho(G)+c)} \tag{2.9}$$

where k is set to control the degrees of freedom in the model, and a_{ij}, b, and c are parameters determined by fitting training data obtained through experiments.

We run the model training experiments on a given machine A and obtain the model parameters for that machine. After that, we can estimate the running time on another machine B by multiplying a normalization factor $f = T_B(G_0)/T(G_0)$, where G_0 is a calibration graph, $T(G_0)$ is the estimated running time for graph G_0 on machine A based on Equation 2.9, and $T_B(G_0)$ is the actual running time for graph G_0 on machine B. Thus, for a given input graph G, the estimated running time on machine B is given by:

$$T_B(G) = T(G) \times \frac{T_B(G_0)}{T(G_0)} \tag{2.10}$$

We have found that this equation is not overly sensitive to the choice of calibration graph, G_0. In our experiments, we choose G_0 as a random graph with $|G_0| = 500$ and $\rho(G_0) = 0.5$.

2.6 Implementation and Optimizations

In this section, we show the implementation of our algorithm in MapReduce, in particular Hadoop, and optimizations of this implementation. We reiterate that other cluster computing frameworks such as MPI or Pregel can be used for this implementation, but they do not give us the flexibility, automatic scalability and fault tolerance, and wide adoption that we have with MapReduce.

2.6.1 Implementation Using MapReduce

In our implementation, we use two MapReduce jobs: the first job performs the partitioning and the second one computes the maximum clique of the subgraphs. There is only one input to the partitioning MapReduce job, namely the original graph. In order to parallelize the computation as soon as possible, we do only one-depth partitioning in the map phase of this job, and distribute the subgraphs to the reducers to continue the partitioning process. Each (*key, value*) pair from this map function contains $(v, N(G, v))$. Each reducer task will partition the subgraph into smaller subgraphs until the estimated running time of each smaller subgraph is less than the bound value. The output of the reducers is a set of $(key, value)$ pairs. Each *value* is a partitioned subgraph, e.g., G', and the *key* is the set of vertices $\{v_{d1}, v_{d2}, \ldots, v_{dm}\}$ that records how G' is partitioned from the original graph G. The pseudocode of the MapReduce implementation of BMC partitioning is shown in Algorithm 3. The global parameter *BOUND* is used to determine when partitioning is terminated. The function Running_Time(G) estimates the running time required to solve the maximum clique problem for graph G.

In the second MapReduce job, the maximum clique of each subgraph is computed using Algorithm 2. The map function reads strings containing the subgraphs and their expanding vertices (i.e., $\{v_k, w_l, u_n\}$ in Equation 2.8) from HDFS files, which are created by the reducers of the first MapReduce job. The map function then splits these strings into $(key, value)$ pairs, where the *value* is the subgraph G' and the *key* is the set of vertices that records how subgraph G' was partitioned from G. In the reduce function, we use Algorithm 2 to calculate the maximum clique for each subgraph. It should be noted that this algorithm can easily be replaced by other maximum clique algorithms such as *MCS* [70] or *cliquer* [56], if desired, without compromising the scalability of the solution. The pseudocode of the second MapReduce job is shown in Algorithm 4.

Algorithm 3 BMC partitioning using MapReduce.

```
 1: /* First map function ($key$, $value$) */
 2: function map_one($key$, $value$)
 3: $G$ = get_graph($value$)
 4: while Running_Time($G$) > $BOUND$ do
 5:     /* $C_G$ is the color list of $G$*/
 6:     $i = \{j|C_G[v_j] = max(C_G[v_k]|v_k \in G)\}$
 7:     Emit ($v_i$, $N(G, v_i)$)
 8:     $G = G - \{v_i\}$
 9: end while
10: Emit (-1, $G$)
11: return
12:
13: /* First reduce function ($key$, $value$) */
14: function reduce_one($key$, $value$)
15: $G$ = get_graph($value$)
16: $w$ = get_vertex($value$) /* $v_i$ in line 7 */
17: multi_partition($G$, $w$, $BOUND$)
18: return
19:
20: function multi_partition($G$, $w$, $BOUND$)
21: $C_G$ = Color($G$)
22: while Running_Time($G$) > $BOUND$ do
23:     $i = \{j|C_G[v_j] = max(C_G[v_k]|v_k \in G)\}$
24:     if Running_Time($N(G, v_i)$) > $BOUND$ then
25:         multi_partition($N(G, v_i)$, $w \cup v_i$, $BOUND$)
26:     end if
27:     Emit ($w \cup v_i$, $N(G, v_i)$)
28:     $G = G - v_i$
29: end while
30: Emit ($w$, $G$)
31: return
```

2.6.2 Optimizations in Hadoop

In order to implement an efficient branch and bound search, we use a global parameter MAX to store the number of vertices in the largest clique found so far. This global parameter is used to set the pruning parameter $|Q_{max}|$ in Algorithm 2 so that it can reduce the search space for most subgraphs. The global parameter MAX is communicated via a socket to the Hadoop job tracker. Before computing the maximum clique for a subgraph G' expanded from vertices $\{v_{d1}, v_{d2}, \ldots, v_{dm}\}$, the reduce function of the second MapReduce job first receives MAX from the socket, and then adds $\{v_{d1}, v_{d2}, \ldots, v_{dm}\}$ to Q and sets $|Q_{max}| = MAX$. The maximum clique of this subgraph is then obtained by running MaxClique(G'). After the algorithm terminates, we again request a current value of MAX over the socket and compare it with the $|Q_{max}|$ found for G'. If $|Q_{max}|$ is greater than MAX, a new maximum clique has been found from this subgraph and we send the value of $|Q_{max}|$ to the job tracker over the socket and update the parameter MAX. Otherwise, we do not update MAX. To obtain fault tolerance, we write MAX into an HDFS file once it is updated, so that if the socket connection fails, the parameter MAX can be read from this HDFS file.

Algorithm 4 Maximum clique on the partitioned subgraphs using MapReduce.

```
1:  /*Second map function (key, value) */
2:  function map_two(key, value)
3:  $G$ = get_graph(value) /* Read graph */
4:  /* key is $w \cup v_i$ in line 27 of Algorithm 3 */
5:  key = get_vertices(value)
6:  Emit (key, $G$)
7:  return
8:
9:  /* Second reduce function (key, value) */
10: function reduce_two(key, value)
11: $G$ = get_graph(value)
12: $Q$ = get_vertices(key)
13: $|Q_{max}| = MAX$ /* $MAX$ is a global parameter */
14: $Q$ = MaxClique($G$)
15: if $|Q| > MAX$ then
16:     Update global parameter $MAX$ with $|Q|$
17: end if
18: return
```

Graph	Order	Size (Bytes)	Density	ω	Comment
G_1	738	192243	0.71	19	Quantum Code
G_2	250	27980	0.9	44	DIMACS C250.9
G_3	2000	999836	0.5	16	DIMACS C2000.5
G_4	4000	4000268	0.5	18	DIMACS C4000.5

Table 2.2: Four graphs used in the experiments.

When deciding the *BOUND* parameter, there is a trade-off between load balancing and the amount of time taken to move graphs to worker nodes. To obtain good load balancing, we need to ensure that there are no large subgraphs that take a long time for maximum clique computation. Therefore, we should partition the graph into small subgraphs, i.e., set the bound value to be small. However, if the bound value is too small, the number of partitioned subgraphs is too large, and it takes a long time to shuffle the subgraphs to worker nodes. It should be noted that the number of partitioned subgraphs increases exponentially with decreasing the bound value. Therefore, an optimal choice of the bound value may improve the performance of our algorithm. We experimentally study the optimal value of this parameter in Section 2.7.2.

2.7 Experimental Evaluation

2.7.1 Experimental Environment

Our experiments are based on four graphs, which are summarized in Table 2.2. The first graph is a real-world coding graph for a CWS quantum error-correcting code, for given length $n = 10$ and $d = 3$. The other three graphs are DIMACS graphs [20, 21], which are synthetically generated graphs meant as benchmarks for maximum clique computation. All the graphs are undirected and unweighted.

We implemented our algorithm on Hadoop 1.0.0, and we perform all our experiments using Amazon's Elastic Compute Cloud (EC2) [25], except the investigation of the running time estimation (Section 2.7.2). Since we mainly want to measure the scalability of our algorithm, we use small Amazon EC2 instances to run our experiments. Each small instance has 1.7 GB of memory and 1 virtual core with 1 EC2 compute unit, which has a similar performance as one 2007-era 1.0–1.2 GHz AMD Opteron or Xeon processor.

2.7.2 Running Time Estimation and Bound Value

In order to evaluate the accuracy of Equation 2.9, we first generate many random graphs with the number of vertices $|G|$ from 100 to 1000 in a step of 50 and the density ρ from 0.1 to $0.93 - 0.0003|G|$. Second, we use the MCR algorithm to solve the maximum clique problem for those graphs and record the running time on a PC with one dual core processor (2.9 GHz) and 4 GB memory. Third, we use some graphs with $|G| = 100, 200, \ldots, 1000$ as training data to fit Equation 2.9, and the remaining graphs to test whether the model can predict the running time.

We find that when $k = 4$ in Equation 2.9, our model could fit the data very well ($\chi^2/d.o.f = 550/527$ with 27 parameters). Therefore, we use $k = 4$ in this paper. Some graphs over-plotted by the best-fit model are shown in Figure 2.2 (note that the y-axis of Figures 2.2, 2.4, and 2.5 is log scale). The predicted running time from the model is also shown in the figure, which shows that the model does indeed give a good estimation of the running time. The figure shows that the model is accurate even up to a running time of around 100 minutes. Since we set the bound value to 60 seconds (i.e., 1 minute), the subgraphs on which the model will be invoked will be well within the region in which it is accurate. Moreover, even if the model estimates are inaccurate, the end-to-end performance of the maximum clique algorithm will not be severely impacted since the algorithm is robust over a wide range of the bound value, as we show next.

To evaluate the robustness of the algorithm when changing the bound value on the estimated running time of the partitioned subgraphs, we first note that if the bound value is too large, some subgraphs will be too large and the load balancing will be bad. However, if the bound value is too small, the number of subgraphs will be too large and Hadoop will need more time to shuffle, read, and write the data. Recall that the number of subgraphs increases exponentially with decreasing the bound value.

We use 32 small EC2 instances for this experiment. The total running time versus the bound value for graphs G_1 and G_2 is shown in Figure 2.3. From this figure, we can see that there is a wide range in which the bound value results in good performance (approximately 5–500 seconds). Therefore, our algorithm is robust to changes or inaccuracy in setting the bound value, and our choice of 60 seconds is justified.

2.7.3 Scalability

In this section, we investigate the scalability of our proposed algorithm, namely, BMC partitioning with the optimization of the global parameter MAX. Comparisons between

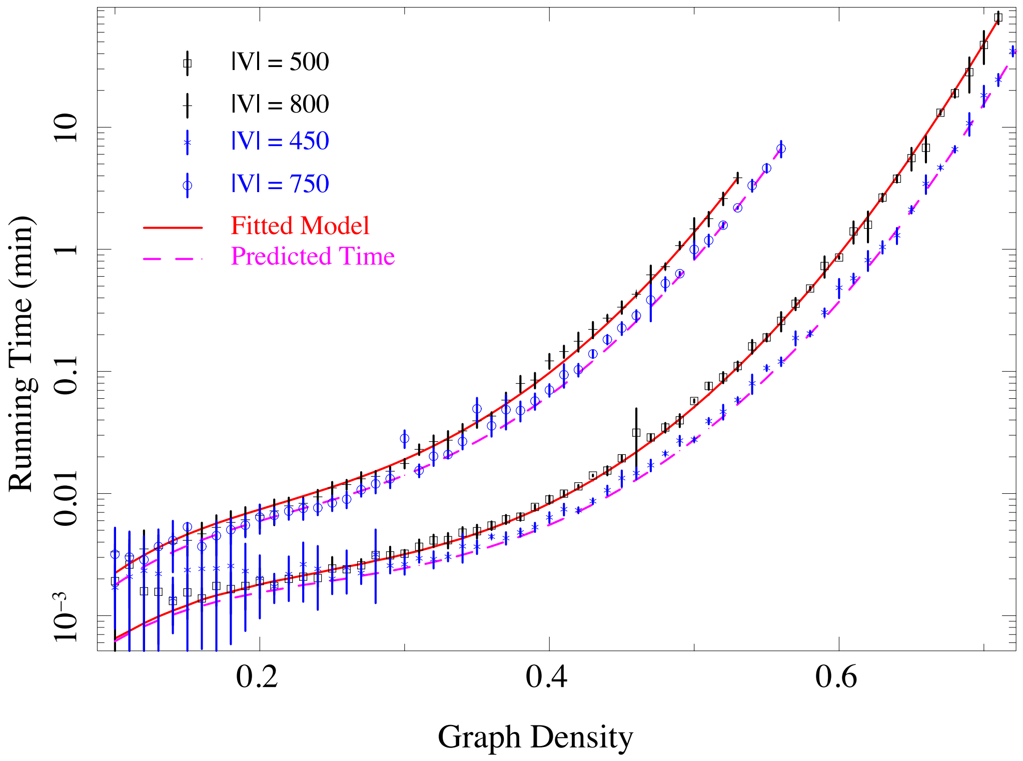


Figure 2.2: Running time for random graphs over-plotted by the best-fit model. The black data are training data used to fit the models (red solid lines), while the blue data are test data used to evaluate model predictions (purple dashed lines).

our algorithm and other alternatives, such as different partitioning strategies, removing the global parameter MAX, one-depth partitioning, and comparing to an MPI algorithm, are presented in the following sections.

We run our algorithm for first three graphs shown in Table 2.2, G_1, G_2, and G_3. We vary the number of EC2 nodes from 2 to 128. The total running time versus the number of EC2 nodes is shown in Figure 2.4. In order to illustrate the scalability of our algorithm, we over-plot a dashed line that indicates ideal scalability, i.e, $T(n) = T(1)/n$, where $T(n)$ is the running time on n EC2 nodes. This figure shows that our algorithm provides excellent scalability that is very close to ideal.

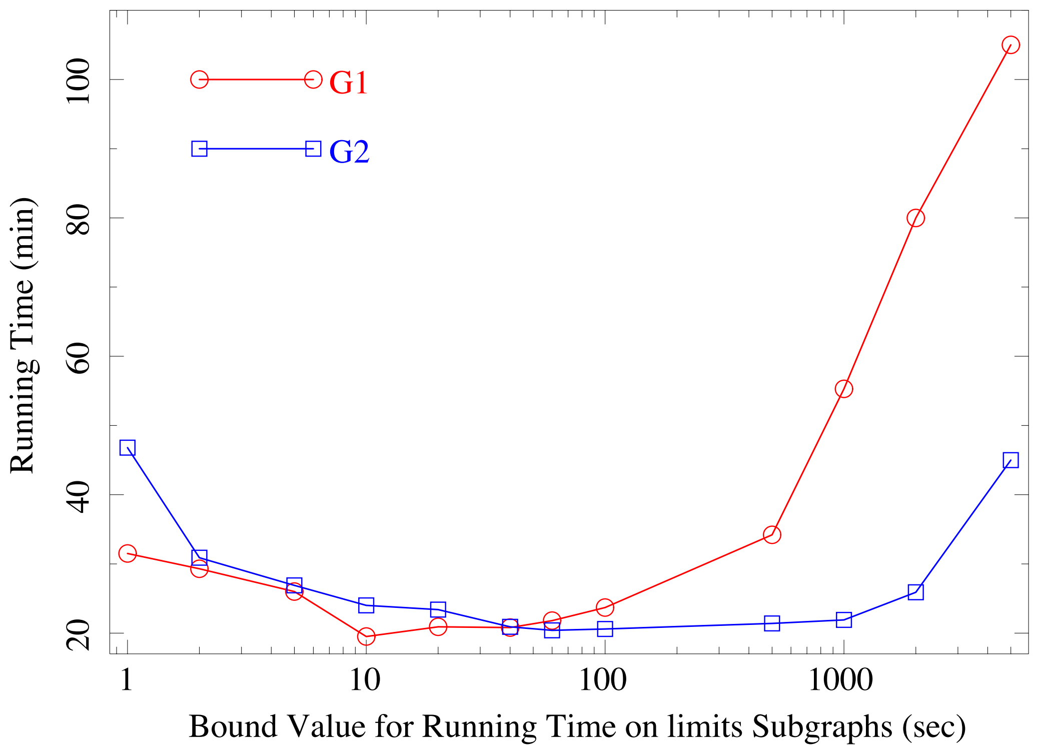


Figure 2.3: Running time of G_1 and G_2 versus the bound value on the running time of the partitioned subgraphs on 32 EC2 nodes.

The largest graph (G_4) is not used to investigate scalability because its running time is too large, particularly on a small number of nodes such as 2 or 4. However, we use this graph to test the effectiveness of our algorithm for solving the maximum clique problem on such large graphs by running it on 128 high CPU medium Amazon EC2 instances, each of which has two CPU cores equivalent to 5 compute units. The problem is completely solved in 39 hours. The size of the maximum clique is found to be 18, which is the same size as the largest clique previously found in this graph. However, we can now say with certainty that this is indeed the maximum clique, and we can enumerate several cliques that have this maximum size. This is the first time to completely solve the maximum clique problem for this DIMACS instance. We estimate that a solution on a single CPU would have required more than 1 year.

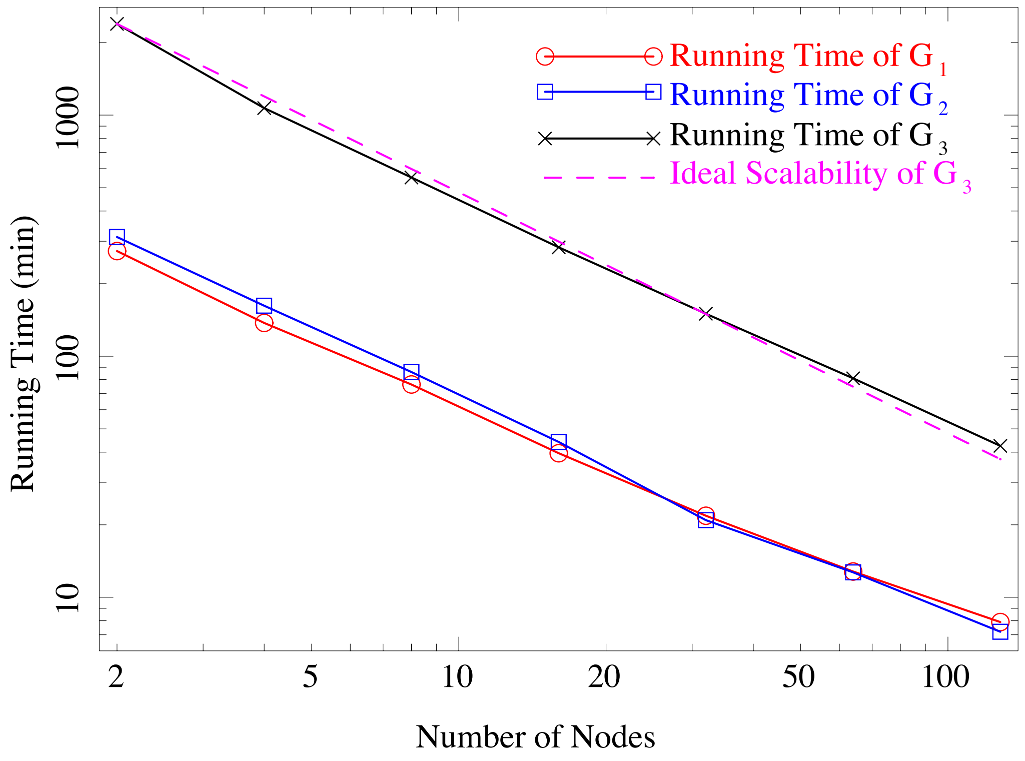


Figure 2.4: The scalability of our algorithm. The dashed purple line shows ideal scalability, i.e., $T(n) = T(1)/n$, where $T(n)$ is the running time on n EC2 nodes.

2.7.4 Comparison of Partitioning Strategies

In this section, we use graphs G_1 and G_2, plus 4 randomly generated graphs (which we call R_1–R_4), to investigate whether our BMC partitioning is better than other partitioning strategies.

In this experiment, bisection partitioning is performed using the Kernighan and Lin algorithm [41], as well as k-way partitioning based on spectral factorization [36] with k set to 16. We also evaluate PICS partitioning, which is implemented using the code in [2]. In each experiment, when the number of vertices of a subgraph is 150 or less, the subgraph is not partitioned any further.

In order to investigate the effectiveness of each partitioning strategy, Table 2.3 lists the final number of subgraphs resulting from the partitioning and the number of vertices in the largest subgraph. The table shows, for each partitioning strategy, partitioning with *depth* 1 to 6. The depth of a partitioning is the height of the partitioning tree.

From the table we can see that bisection partitioning and PICS cannot partition the original graph into smaller subgraphs for any graph. The largest subgraph has the same size as the original graph even for a graph with low density (R_4: $|G| = 500$, $\rho(G) = 0.05$). The k-way partitioning algorithm can partition graphs with low density into smaller subgraphs but it is not as effective as our BMC partitioning algorithm. Additionally, k-way partitioning also cannot partition graphs with medium or high density. In contrast, Table 2.3 shows that BMC partitioning can effectively partition any graph into smaller subgraphs.

2.7.5 Sorting By Color vs. Random Ordering

The sorting order of vertices (based on color or not) may affect the performance of our algorithm. Here, we use graphs G_1 and G_2 to investigate whether BMC partitioning, which does sort based on color, is better than a partitioning strategy based on a random order of vertices. The comparison of the running time of G_1 and G_2 based on these two strategies on 32 EC2 nodes is shown in Table 2.4. The experiment shows that BMC partitioning with vertices sorted based on color can significantly improve performance.

2.7.6 Global Parameter MAX

The running time of graph G_1 and G_2 on 32 EC2 nodes without the global MAX parameter is shown in Table 2.4. Without the MAX parameter, $|Q_{max}|$ in Algorithm 2 is set to zero for each subgraph. The table clearly shows that the global parameter MAX significantly improves performance.

2.7.7 One-depth Partitioning vs. Multi-depth Partitioning

In order to investigate whether BMC partitioning is better than one-depth partitioning, we implement a Hapdoop MapReduce program to calculate the maximum clique of a graph using one-depth partitioning. The graph is partitioned with one-depth partitioning until the estimated running time of the remaining graph is less than the bound value, i.e., 60 seconds, which is the same as the bound value in multi-depth partitioning. The relationship between the total running time and the number of EC2 nodes for graph G_2 is

Graph	Name	G_1		G_2		R_1		R_2		R_3		R_4	
	$\lvert V\rvert$	250		738		500		500		500		500	
	Density	0.9		0.73		0.3		0.2		0.1		0.05	
Name	Depth	n	v	n	v	n	v	n	v	n	v	n	v
Bisection Partitioning	1	3	250	3	738	3	500	3	500	3	500	3	500
	2	5	250	9	738	9	500	9	500	9	500	9	500
	3	7	250	27	738	19	500	19	500	19	500	19	500
	4	9	250	65	738	33	500	33	500	33	500	33	500
	5	11	250	131	738	51	500	51	500	51	500	51	500
	6	13	250	233	738	73	500	73	500	73	500	73	500
k-Partitioning	1	31	250	31	737	31	500	31	496	31	458	31	444
	2	211	250	421	737	421	500	421	495	361	426	271	350
	3	1051	250	4381	737	4321	500	4351	470	2341	389	781	241
	4	4021	250	34771	737	35191	500	31711	461	9241	341	1170	148
	5	12721	250	224851	737	224160	500	174300	457	24270	304	1168	148
	6	34831	250	1228131	737	1174463	500	764451	450	37949	269	1168	148
PICS Partitioning	1	3	250	7	737	3	500	3	500	3	500	3	500
	2	5	250	67	737	5	500	7	500	5	500	7	500
	3	7	250	353	737	7	500	13	500	7	500	11	500
	4	9	250	2171	737	9	500	21	500	9	500	15	500
	5	11	250	13419	737	11	500	31	500	11	500	19	500
	6	13	250	78496	737	13	500	43	500	13	500	23	500
BMC Partitioning	1	101	210	589	484	351	150	351	150	351	150	351	150
	2	2830	183	98062	350	351	150	351	150	351	150	351	150
	3	18703	160	4693076	258	351	150	351	150	351	150	351	150
	4	20951	150	40869924	184	351	150	351	150	351	150	351	150
	5	20951	150	42730190	150	351	150	351	150	351	150	351	150
	6	20951	150	42730190	150	351	150	351	150	351	150	351	150

Table 2.3: Different partitioning strategies. The value n is the final number of subgraphs, and v is the number of vertices in the largest subgraph.

shown in Figure 2.5. The figure also shows the result of BMC partitioning. It is clear that one-depth partitioning does not have good scalability for dense graphs because it does not effectively balance load.

2.7.8 Comparison with an MPI Algorithm

We developed an MPI-based maximum clique algorithm using the MPICH implementation of MPI. We employ the framework proposed in [58], in which one process acts as the master and collects results from other processes, which act as workers. The master first sends the original graph to all the workers, and then assigns one vertex to each worker node. If the master sends the vertex index k to worker i, worker i calculates the maximum clique of the subgraph $N(G - \{v_1, v_2, \ldots, v_{k-1}\}, v_k)$, and sends the result back to the master. If one worker has finished its work, the master will send another partition to it until the entire job has been finished. The algorithm in [58] calculates the maximum clique of subgraphs

Graph	Color Partitioning T_0	Random Partitioning T_r	$\frac{T_r - T_o}{T_o}$ (%)	Without MAX T_m	$\frac{T_m - T_o}{T_o}$ (%)
G_1	21.8	51.5	135	198	804
G_2	20.9	30.0	46	108	427

Table 2.4: Running time (in minutes) of G_1 and G_2 on 32 EC2 nodes based on color partitioning compared to the running time based on random partitioning. The running time without the global parameter MAX is also shown.

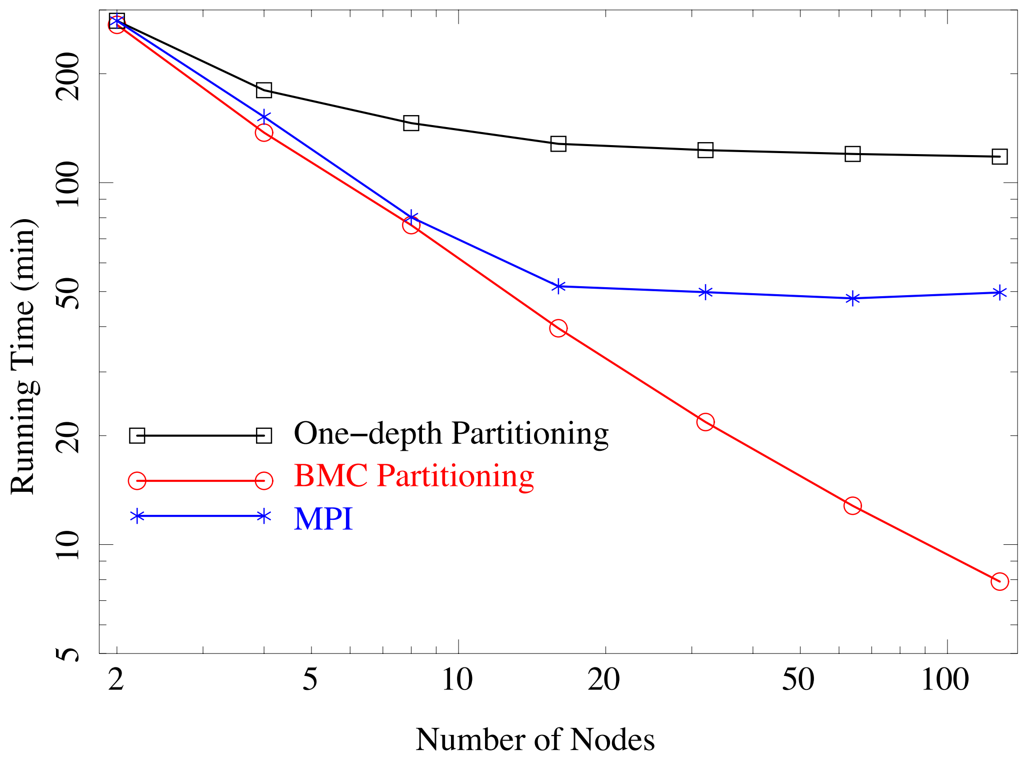


Figure 2.5: Running time of G_2 using a varying number of EC2 nodes based on one-depth partitioning and an MPI implementation, compared to BMC partitioning.

using the classical algorithm in [10]. We replace the algorithm in the workers with the MCR algorithm [69] to improve performance.

The MPI algorithm is similar to the one-depth partitioning algorithm and it also suffers from poor load balancing. In order to investigate the performance and scalability of the MPI algorithm, we use this algorithm to calculate the maximum clique of graph G_2. Figure 2.5 shows the total running time versus the number of EC2 nodes. The number of nodes in this figure refers to the number of worker nodes, since the master node in the MPI algorithm does not do any computation. Our one-depth algorithm is comparable in performance with the MPI algorithm even though the MPI implementation is in C and our one-depth implementation is in Java. Our multi-depth BMC partitioning significantly outperforms the MPI implementation because BMC has better scalability.

2.8 Summary

In this chapter, we presented a scalable and fault tolerant maximum clique algorithm based on MapReduce. Our maximum clique computation relies on a novel graph partitioning algorithm (BMC partitioning) that we implement in MapReduce. BMC partitioning provides good scalability as compared to alternative algorithms that we investigated. Another way that our maximum clique computation achieves scalability is by using a branch and bound search where the bound on the size of the largest clique found so far is updated by all workers (the MAX parameter). The partitioning algorithm and the single node maximum clique algorithm both rely on graph coloring to sort vertices in a way that further improves scalability. We were able to show that our proposed maximum clique computation technique can effectively scale to large cluster sizes and can find the maximum cliques for large dense graphs.

There are several directions for future work. For example, we have used the default hash value to shuffle different subgraphs from the map tasks to the reduce tasks. However, this random shuffling is not optimal. Therefore, we want to find a better method to combine these subgraphs such that we can improve the load balancing. We can also theoretically calculate the running time and compare it with that from random shuffling.

Chapter 3

Matrix Inversion Using MapReduce

After presenting maximum clique computation on MapReduce, we now turn our attention to another scientific computing task, namely matrix inversion. In addition to having numerous applications, matrix inversion is interesting for our research because it is both compute intensive and data intensive. It requires a lot of CPU, and in addition a lot of data is shuffled while the matrix inverse is computed. Thus, it is a different type of scientific computing task compared to computing the maximum clique.

Note that the crux of our technique is partitioning the computation required for matrix inversion into independent pieces. Such partitioning is required for any cluster computing framework. Thus, while this paper focuses on MapReduce, our technique can be used as a basis for implementing matrix inversion in other cluster computing systems such as Dryad or YARN. We leave this point as a direction for future work.

The contributions of this chapter are as follows:

- The choice of LU decomposition for matrix inversion in order to enable a MapReduce implementation.
- An algorithm for computing the LU decomposition that partitions the computation into independent pieces.
- An implementation of the proposed algorithms as a pipeline of MapReduce jobs.
- Optimizations of this implementation to improve numerical accuracy, I/O performance, and memory locality.
- An extensive experimental evaluation on Amazon EC2.

The rest of this chapter is organized as follows. In Section 3.1, we present some basic matrix inversion algorithms on a single node. In Section 3.2, we present related work. Our algorithm is introduced in Section 3.3, and our implementation in Section 3.4. We describe optimizations of the implementation in Section 3.5. We present an experimental evaluation in Section 3.6. Section 3.7 concludes.

3.1 Preliminaries

For any matrix $\mathbf{A}$, let $[\mathbf{A}]_{ij}$ denote its element of the i-th row and the j-th column, and denote by $[\mathbf{A}]_{[x_1 \ldots x_2][y_1 \ldots y_2]}$ the block defined by the beginning row x_1 (inclusive) and the ending row x_2 (exclusive), and by the beginning column y_1 (inclusive) and the ending column y_2 (exclusive).

A square matrix ($\mathbf{A}$: m-by-n) is a matrix with the same number of rows and columns, i.e, $m = n$. The m or n is called the order of matrix $\mathbf{A}$. If there is another square matrix $\mathbf{B}$ that satisfies the following equation:

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I}_n \tag{3.1}$$

where $\mathbf{I_n}$ is the identity matrix of order n, then $\mathbf{A}$ is called invertible. $\mathbf{B}$ is then the inverse of $\mathbf{A}$, denoted as $\mathbf{A}^{-1}$. Here the identity matrix $\mathbf{I}_n$ is an n-by-n square matrix with entries $a_{ij} = 1$ for $i = j$ and $a_{ij} = 0$ for $i \neq j$. A square matrix $\mathbf{A}$ is invertible if and only if $\mathbf{A}$ is non-singular, that is, $\mathbf{A}$ is of full rank n. The rank of a matrix is the number of rows (or columns) in the largest collection of linearly independent rows (or columns) of a matrix. Therefore, a square matrix is invertible if and only if there are no linearly dependent rows (or columns) in this matrix. For a collection of rows or columns to be linearly independent, none of these rows or columns can be obtained by a linear combination of other rows or columns in the collection.

The inverse of a matrix can be computed using many methods, such as Gauss-Jordan elimination, LU decomposition (also called LU factorization), Singular Value Decomposition (SVD), and QR decomposition [17, 47, 63, 74]. In order to clarify our choice for MapReduce implementation, we briefly discuss these methods.

Gauss-Jordan elimination is a classical and well-known method to calculate the inverse of a matrix [63]. This method has two different variants: row elimination and column elimination. They are quite similar so we only discuss the method using row elimination. The method first concatenates the matrix $\mathbf{A}$ and the identity matrix $\mathbf{I}_n$ into a new matrix $[\mathbf{A}|\mathbf{I}_n]$. Then, using elementary row operations which include row switching, row multiplication, and row addition, the method transforms the left side to the identity matrix, such that the right side is the inverse of matrix $\mathbf{A}$. That is:

$$[\mathbf{A}|\mathbf{I}_n] \xrightarrow{\text{row operations}} [\mathbf{U}|\mathbf{B}] \xrightarrow{\text{row operations}} [\mathbf{I_n}|\mathbf{A^{-1}}] \tag{3.2}$$

where $\mathbf{U}$ is an upper triangular matrix. The Gauss-Jordan elimination method first converts the matrix $\mathbf{A}$ into an upper triangular matrix in n steps using row operations as follows. In the first step, the first row is multiplied by a constant such that the first element in this row equals to 1, and the first row times a constant is subtracted from the i-th $(1 < i \leq n)$ row of $[\mathbf{A}|\mathbf{I}_n]$ such that the first element in the i-th $(1 < i \leq n)$ row is 0. In the k-th step, the k-th row is multiplied by a constant such that the k-th element in this row equals to 1, and the k-th row times a constant is subtracted from the i-th $(k < i \leq n)$ row of $[\mathbf{A}|\mathbf{I}_n]$ such that the k-th element in the i-th $(k < i \leq n)$ row is 0. If the k-th element of the k-th row is already 0 or close to 0 before the subtraction, we first switch the k-th row with any row below the k-th row where the k-th element is not 0. After $n-1$ steps, the left part of matrix $[\mathbf{A}|\mathbf{I}_n]$ is converted into an upper triangular matrix.

Next, the method converts the upper triangular matrix into an identity matrix also using row operations in n steps as follows. In the first step, the n-th row of $[\mathbf{U}|\mathbf{B}]$ times a constant is subtracted from the i-th $(1 \leq i < n)$ row of matrix $[\mathbf{U}|\mathbf{B}]$ such that the n-th element of the i-th $(1 \leq i < n)$ row of matrix $[\mathbf{U}|\mathbf{B}]$ is 0. In the k-th step, the $n+1-k$-th row of of $[\mathbf{U}|\mathbf{B}]$ times a constant is subtracted from i-th $(1 \leq i < n-k)$ row of matrix $[\mathbf{U}|\mathbf{B}]$ such that the n-th element of the i-th $(1 \leq i < n-k)$ row of matrix $[\mathbf{U}|\mathbf{B}]$ is 0. After $n-1$ steps, the upper triangular matrix in the left part is converted into an identity matrix, and the right part is the inverse of matrix $\mathbf{A}$. The Gauss-Jordan elimination method uses n^3 multiplication operations and n^3 addition operations to invert an $n \times n$ matrix, which is as efficient as, if not better than, any other method. However, this method is very difficult to parallelize due to the large number of steps that depend on each other in a sequential fashion, $O(n)$.

The LU decomposition method, also called LU factorization, first decomposes the original matrix into a product of two matrices $\mathbf{A} = \mathbf{LU}$, where $\mathbf{L}$ is a lower triangular matrix that has nonzero elements only on the diagonal and below, and $\mathbf{U}$ is an upper triangular matrix that has nonzero elements only on the diagonal and above. Since the inverse of a triangular matrix is easy to compute using back substitution [63] (also refer to Section 3.3), we can get the inverse of $\mathbf{A}$ as $\mathbf{U}^{-1}\mathbf{L}^{-1}$. The LU decomposition method uses the same number of multiplication and addition operations as the Gauss-Jordan method. However the LU decomposition method is much easier to parallelize because the LU decomposition can be done with a recursive block method. Therefore, in this thesis, we use the LU decomposition method to compute matrix inverse on MapReduce. The details of the LU decomposition method and the parallel algorithm will be discussed in Section 3.3.

Instead of decomposing matrix $\mathbf{A}$ into the product of two matrices, the SVD decomposition method [63] decomposes matrix $\mathbf{A}$ into the product of three matrices, $\mathbf{A} = \mathbf{U}\mathbf{W}\mathbf{V}^T$ ($\mathbf{V}^T$ is the transpose of matrix $\mathbf{V}$). $\mathbf{W}$ is a diagonal matrix with only positive or zero elements. $\mathbf{U}$ and $\mathbf{V}$ are both orthogonal matrices (i.e. $\mathbf{U}\mathbf{U}^T = \mathbf{V}\mathbf{V}^T = \mathbf{I}_n$), such that the inverse of $\mathbf{A}$ can be given by $\mathbf{A}^{-1} = \mathbf{V}\mathbf{W}^{-1}\mathbf{U}^T$, where the inverse of diagonal matrix $\mathbf{W}$ is easily obtained with $[\mathbf{W}^{\mathbf{-1}}]_{ii} = 1/[\mathbf{W}]_{ii}$ in running time $O(n)$. However, this method needs frequent row exchanges, which means that the computation cannot be partitioned into independent pieces. Hence, this method is not suitable for the MapReduce framework, which requires that the task can be divided into independent pieces to be worked on by separate nodes.

The QR decomposition first decomposes the original matrix $\mathbf{A}$ into a product of an orthogonal matrix $\mathbf{Q}$ and an upper triangular matrix $\mathbf{R}$, i.e., $\mathbf{A} = \mathbf{Q}\mathbf{R}$ and $\mathbf{A}^{-1} = \mathbf{R}^{-1}\mathbf{Q}^T$. One way of computing the QR decomposition is using the Gram-Schmidt process [63]. Let $\mathbf{a}_i$ be the vector corresponding to the i-th column of $\mathbf{A}$, $\mathrm{proj}_{\mathbf{e}}\mathbf{a} = \frac{\langle \mathbf{e},\mathbf{a}\rangle}{\langle \mathbf{e},\mathbf{e}\rangle}\mathbf{e}$ be the projection of a vector $\mathbf{a}$ onto another vector $\mathbf{e}$, and $\langle \mathbf{e}, \mathbf{a}\rangle$ be the scalar product (or dot product, or inner product) of the vector $\mathbf{e}$ and the vector $\mathbf{a}$. The Gram-Schmidt process returns a set of orthogonal vectors $\mathbf{e}_i$ which spans the same vector space as the column space of $\mathbf{A}$, where

$$\begin{aligned}
\mathbf{e}_1 &= \frac{\mathbf{a}_1}{||\mathbf{a}_1||}, \\
\mathbf{e}_2 &= \frac{\mathbf{a}_2 - \mathrm{proj}_{\mathbf{e}_1}\mathbf{a}_2}{||\mathbf{a}_2 - \mathrm{proj}_{\mathbf{e}_1}\mathbf{a}_2||}, \\
\cdots &, \\
\mathbf{e}_k &= \frac{\mathbf{a}_k - \sum_{j=1}^{k-1} \mathrm{proj}_{\mathbf{e}_j}\mathbf{a}_k}{||\mathbf{a}_k - \sum_{j=1}^{k-1} \mathrm{proj}_{\mathbf{e}_j}\mathbf{a}_k||}.
\end{aligned} \tag{3.3}$$

This will then give the matrices $\mathbf{Q}$ and $\mathbf{R}$.

$$\begin{aligned}
\mathbf{Q} &= [\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_n], \\
\mathbf{R}_{ij} &= \begin{cases} \langle \mathbf{e}_i, \mathbf{a}_j\rangle & \text{for } i \leq j \\ 0 & \text{for } i > j \end{cases}
\end{aligned} \tag{3.4}$$

However, as the vector of $\mathbf{e}_k$ depends on all vectors $\mathbf{e}_i$ ($i = 1, ..., k-1$), we can only compute the $\mathbf{e}_k$ vectors one-by-one, which means that we need at least n steps to compute all $\mathbf{e}_k$. Therefore, this QR decomposition method is not easy to parallelize.

In order to improve numerical accuracy, the Gauss-Jordan elimination method and LU decomposition could be applied with the idea of *pivoting* (since we do not want any pivots to be close to 0). For example, for the following 2×2 matrix

$$A = \begin{pmatrix} 10^{-5} & 1 \\ 1 & 10^4 \end{pmatrix}, \tag{3.5}$$

if we want to use the Gauss-Jordan elimination method to invert the matrix without pivoting, the matrix inversion process can be shown as

$$\begin{aligned} (\mathbf{A}|\mathbf{I}) &= \begin{pmatrix} 10^{-5} & 1 & 1 & 0 \\ 1 & 10^4 & 0 & 1 \end{pmatrix} \\ &\xrightarrow{\text{(row 1)}\times 10^5} \begin{pmatrix} 1 & 10^5 & 10^5 & 0 \\ 1 & 10^4 & 0 & 1 \end{pmatrix} \\ &\xrightarrow{\text{(row 2) - (row 1)}} \begin{pmatrix} 1 & 10^5 & 10^5 & 0 \\ 0 & -90000 & -10^5 & 1 \end{pmatrix} \\ &\xrightarrow{\text{(row 2)}/-90000} \begin{pmatrix} 1 & 10^5 & 10^5 & 0 \\ 0 & 1 & 1.11111 & -10^{-5} \end{pmatrix} \\ &\xrightarrow{\text{(row 1) - (row 2)}\times 10^5} \begin{pmatrix} 1 & 0 & -11111 & 1 \\ 0 & 1 & 1.11111 & -10^{-5} \end{pmatrix}, \end{aligned} \tag{3.6}$$

where we assume the data is accurate to 10^{-5}. The inverse of matrix $\mathbf{A}$ is therefore

$$\mathbf{A}^{-1} = \begin{pmatrix} -11111 & 1 \\ 1.11111 & -10^{-5} \end{pmatrix}. \tag{3.7}$$

However, if we invert matrix $\mathbf{A}$ with pivoting, the inversion process can be shown as

$$\begin{aligned} (\mathbf{A}|\mathbf{I}) &= \begin{pmatrix} 10^{-5} & 1 & 1 & 0 \\ 1 & 10^4 & 0 & 1 \end{pmatrix} \\ &\xrightarrow{\text{switch (row 1) with (row 2)}} \begin{pmatrix} 1 & 10^4 & 0 & 1 \\ 10^{-5} & 1 & 1 & 0 \end{pmatrix} \\ &\xrightarrow{\text{(row 2) - (row 1)}\times 10^{-5}} \begin{pmatrix} 1 & 10^4 & 0 & 1 \\ 0 & 0.9 & 1 & -10^{-5} \end{pmatrix} \\ &\xrightarrow{\text{(row 2)}/0.9} \begin{pmatrix} 1 & 10^4 & 0 & 1 \\ 0 & 1 & 1.11111 & -10^{-5} \end{pmatrix} \\ &\xrightarrow{\text{(row 1) - (row 2)}\times 10^4} \begin{pmatrix} 1 & 0 & -11111.1 & 1.1 \\ 0 & 1 & 1.11111 & -10^5 \end{pmatrix}, \end{aligned} \tag{3.8}$$

and the matrix inverse is

$$\mathbf{A}^{-1} = \begin{pmatrix} -11111.1 & 1.1 \\ 1.11111 & -10^{-5} \end{pmatrix}. \tag{3.9}$$

It is easy to check that the Gauss-Jordan elimination method with pivoting produces a more accurate matrix inverse than the method without pivoting. In general, the pivot in the k-th step of the Gauss-Jordan elimination is chosen such that the value of the k-th element is the maximum among the values of all the k-th elements from the k-th row to the n-th row. There is a similar pivoting in the LU decomposition (see Algorithm 5). With pivoting, the product of $\mathbf{L}$ and $\mathbf{U}$ equals $\mathbf{PA}$ where $\mathbf{P}$ is a permutation operation. Finally the inverse of $\mathbf{A}$ is $\mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}$.

To recap: There are multiple ways to compute the inverse of a matrix. Many of these ways use an optimal number of multiplication and addition operations, but are not easy to parallelize. In this thesis we use LU decomposition since it uses the optimal number of operations and is easy to parallelize. LU decomposition can also be used with pivoting to improve numerical accuracy.

3.2 Related Work

Several software packages have been developed that support matrix inversion, such as LINPACK [24], LAPACK [4] and ScaLAPACK [7]. The LINPACK package is written in Fortran and designed for supercomputers. The LAPACK package is developed from LINPACK and is designed to run efficiently on shared-memory vector supercomputers. ScaLAPACK is a software package that tries to provide a high-performance linear algebra library for parallel distributed memory machines instead of shared-memory in LAPACK. This package provides some routines for matrix inversion (see Section 3.6.5 for details). However, this package does not provide any fault tolerance, while our algorithm provides fault tolerance through the use of MapReduce. In addition, we show that the scalability of ScaLAPACK is not as good as our approach.

Parallel algorithms for inverting some special matrices also appear in the literature. Lau, Kumar, and Venkatesh [44] propose algorithms for parallel inversion of sparse symmetric positive matrices on SIMD and MIMD parallel computing platforms. It is not a surprise that these algorithms perform better than general algorithms that do not take into account any special properties of the input matrix. For symmetric positive definite matrices (not necessarily sparse), Bientinesi, Gunter, and Geijn [6] present a parallel matrix inversion algorithm based on the Cholesky factorization for symmetric matrices. Their implementation is based on the Formal Linear Algebra Methodology Environment (FLAME). The implementation shows good performance and scalability, but it does not work for general matrices and is not suitable for large clusters.

The most important step in our matrix inversion technique is LU decomposition. This decomposition has been investigated by many researchers. Agullo et al. [1] have shown that the LU decomposition in double precision arithmetic can reach a throughput of 500 Gflop/s. That work uses powerful CPUs, GPUs, and large memory. Although this method can solve the LU decomposition problem very efficiently, it is a centralized method that is not suitable for MapReduce. Moreover, it needs special hardware (GPUs) and large memory.

Zheng et al. [84] present an implementation of LU decomposition on a multi-core digital signal processor that does pre-fetching and pre-shuffling in MapReduce [66]. The algorithm is very simple and only runs row operations in reduce tasks, using the LU decomposition algorithm on a single node. That is, one reduce task computes one row as in lines 10–12 in Algorithm 5, so that the method needs n MapReduce tasks to decompose an $n \times n$ matrix, which represents very poor scalability.

Matrix inversion using LU decomposition has been investigated recently by Dongarra et al. [23], where a tile data layout [1] is used to compute the LU decomposition and the upper triangular matrix inversion. In that paper, a run time environment called QUARK is used to dynamically schedule numerical kernels on the available processing units in order to reduce the synchronization required between different CPU cores. The algorithm is suitable for multi-core architectures with shared memory and it achieves better performance than other numerical libraries, such as LAPACK and ScaLAPACK. However, this algorithm is not suitable for a distributed environment, since it relies on large shared memory. Hence, its scalability is limited.

Zhang and Yang [83] investigate I/O optimization for big array analytics. They improve the performance of a broad range of big array operations by increasing sharing opportunities. Our technique also optimizes I/O for big matrices (Section 3.5.2). However, our technique mainly focuses on reusing the data in memory as many times as possible to reduce the need for reading data from disk.

To the best of our knowledge, there are no matrix inversion algorithms using MapReduce, although there are several software packages for other matrix operations that use MapReduce. One of these packages is SystemML [31], which provides a high-level language for expressing some matrix operations such as matrix multiplication, division, and transpose, but not matrix inversion. SystemML provides a MapReduce implementation of these operations, and it achieves good scalability and fault tolerance.

3.3 Matrix Inversion Algorithm

As discussed in Section 3.1, the LU decomposition is the most important step toward our solution to the matrix inversion problem. In this section, we show how we compute the lower triangular matrix **L** and the upper triangular matrix **U** in both the single node and parallel setting. We also show how to use the **L** and **U** matrices to compute the matrix inverse.

The LU decomposition algorithm on a single node is widely studied and understood. It has two variants: with and without pivoting. Since pivoting can significantly improve numerical accuracy, we only discuss the algorithm with pivoting (in Algorithm 5, the row having the maximum j-th element among rows j to n is selected in the j-th loop). Although this algorithm can be found in many references (e.g., [33] and [63]), we present it here for completeness.

Let $a_{ij} = [\mathbf{A}]_{ij}$, $l_{ij} = [\mathbf{L}]_{ij}$, and $u_{ij} = [\mathbf{U}]_{ij}$. The LU decomposition $\mathbf{A} = \mathbf{LU}$ can be presented as follows (the blank elements in the **L** and **U** matrices are zeros):

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ a_{31} & a_{32} & \dots & a_{3n} \\ \dots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} l_{11} & & & \\ l_{12} & l_{22} & & \\ l_{13} & l_{23} & l_{33} & \\ \dots & & & \\ l_{n1} & l_{2n} & \dots & l_{nn} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ & u_{22} & \dots & u_{2n} \\ & & u_{33} & \dots \\ & & & \dots \\ & & & u_{nn} \end{pmatrix} \tag{3.10}$$

This matrix multiplication can be viewed as a system of linear equations. Since the difference between the number of unknown arguments (l_{ij} and u_{ij}) and the number of equations is n, there are n free arguments that can be set to any value. Generally, these n free arguments are chosen to be l_{ii} or u_{ii} ($i = 1, ..., n$) and they are all set to be 1.0. In our work we set all l_{ii} to 1.0. The other remaining unknown arguments can be derived by the following equations:

$$\begin{aligned} u_{ij} &= a_{ij} - \sum_{k=1}^{j-1} l_{jk} u_{kj}, \\ l_{ij} &= \frac{1}{u_{jj}}(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}), \end{aligned} \tag{3.11}$$

In order to improve numerical accuracy, the rows of the original matrix **A** are permuted, such that we decompose the pivoting matrix **PA** instead of the original one, i.e., $\mathbf{PA} = \mathbf{LU}$. It should be noted that pivoting in LU decomposition does not affect the final inverse of matrix **A** because we can apply the permutation matrix **P** to the product of the inverses of **L** and **U**. That is, we can compute $\mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}$ to obtain the inverse of original matrix **A** since $\mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{PA} = \mathbf{I}_n$. Computing $\mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}$ is equivalent to permuting the columns in the result of $\mathbf{U}^{-1}\mathbf{L}^{-1}$ according to **P**. The pseudocode of LU decomposition with pivoting is shown in Algorithm 5.

Algorithm 5 LU decomposition on a single node.

```
function LUDecomposition($\mathbf{A}$)
for i = 1 to n do
    j = {j | $[\mathbf{A}]_{ji} = \max([\mathbf{A}]_{ii}, [\mathbf{A}]_{i+1i}, ..., [\mathbf{A}]_{ni})$}
    Add j to $\mathbf{P}$
    Swap $i$-th row with $j$-th row if $i \neq j$
    for j = i + 1 to n do
        $[\mathbf{A}]_{ji} = [\mathbf{A}]_{ji}/[\mathbf{A}]_{ii}$
    end for
    for j = i + 1 to n do
        for k = i + 1 to n do
            $[\mathbf{A}]_{jk} = [\mathbf{A}]_{jk} - [\mathbf{A}]_{ji} \times [\mathbf{A}]_{ik}$
        end for
    end for
end for
return $(\mathbf{A}, \mathbf{P})$      /* i.e., return (L, U, P) */
```

After calling Algorithm 5, the decomposed lower triangular matrix and upper triangular matrix are stored in the original matrix, except for the diagonal elements of the lower triangular matrix which are all set to 1.0. Since there is only one nonzero element in each row or each column of the permutation matrix, the permutation of rows can be stored in an array **S**, where $[\mathbf{S}]_i$ indicates the permuted row number for the i-th row of matrix **A**.

After decomposing the matrix **A** into **L** and **U**, we need to invert **L** and **U** separately. The inverse of a lower triangular matrix is given by

$$[\mathbf{L}^{-1}]_{ij} = \begin{cases} 0 & \text{for } i < j \\ \frac{1}{[\mathbf{L}]_{ii}} & \text{for } i = j \\ -\frac{1}{[\mathbf{L}]_{ii}} \sum_{k=j}^{i-1} [\mathbf{L}]_{ik}[\mathbf{L}^{-1}]_{kj} & \text{for } i > j \end{cases}, \tag{3.12}$$

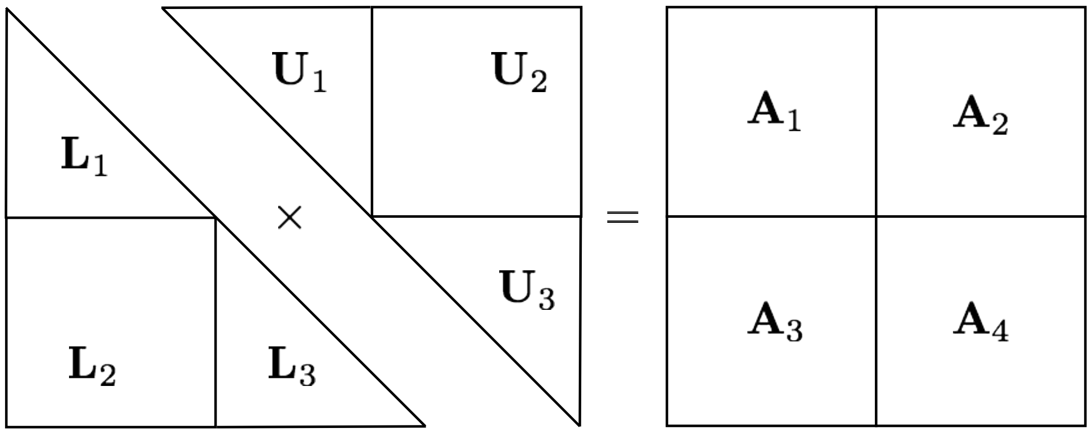


Figure 3.1: Block method for LU decomposition.

where $\mathbf{L}$ and $\mathbf{L}^{-1}$ are the original lower triangular matrix and its matrix inverse respectively.

The inverse of the upper triangular matrix can be computed similarly. In fact, in our implementation, for the upper triangular $\mathbf{U}$ matrix we instead compute the inverse of $\mathbf{U}^T$, which is a lower triangular matrix, as an optimization. The details can be found in Section 3.4.

The classical LU algorithm is not suitable for parallel computing, so we use a block method to compute the LU decomposition in parallel. Our block method is shown in Figure 3.1.

In this method, the lower triangular matrix $\mathbf{L}$ and the upper triangular matrix $\mathbf{U}$ are both split into three submatrices, while the original matrix $\mathbf{A}$ is split into four submatrices. These smaller matrices satisfy the following equations:

$$\begin{aligned} \mathbf{L}_1\mathbf{U}_1 &= \mathbf{P}_1\mathbf{A}_1, \\ \mathbf{L}_1\mathbf{U}_2 &= \mathbf{P}_1\mathbf{A}_2, \\ \mathbf{L}_2^{'}\mathbf{U}_1 &= \mathbf{A}_3, \\ \mathbf{L}_3\mathbf{U}_3 &= \mathbf{P}_2(\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2), \\ \mathbf{L}_2 &= \mathbf{P}_2\mathbf{L}_2^{'}, \end{aligned} \tag{3.13}$$

where both $\mathbf{P}_1$ and $\mathbf{P}_2$ are permutations of rows. The entire LU decomposition can be represented as

$$\mathbf{LU} = \begin{pmatrix} \mathbf{P}_1 & \\ & \mathbf{P}_2 \end{pmatrix} \mathbf{A} = \mathbf{PA},$$

where $\mathbf{P}$ is also a permutation of rows obtained by augmenting $\mathbf{P}_1$ and $\mathbf{P}_2$.

The method partitions an LU decomposition into two smaller LU decompositions and two systems of linear equations which can be easily computed in parallel (details in the following paragraphs). The logical order of computing the L and U blocks on the left-hand-side of Equation 3.13 is as follows: First, $\mathbf{L}_1$ and $\mathbf{U}_1$ are computed from $\mathbf{A}_1$ and $\mathbf{P}_1$. Then $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ are computed from $\mathbf{L}_1$, $\mathbf{U}_1$, $\mathbf{P}_1$, $\mathbf{A}_2$, and $\mathbf{A}_3$. Third, $\mathbf{L}_3$ and $\mathbf{U}_3$ are computed from $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ and $\mathbf{P}_2$. Finally, $\mathbf{L}_2$ is computed from $\mathbf{L}_2^{'}$ and $\mathbf{P}_2$.

First, let us examine the computation of $\mathbf{L}_1$ and $\mathbf{U}_1$. If submatrix $\mathbf{A}_1$ is small enough, e.g., order of 10^3 or less, it can be decomposed into $\mathbf{L}_1$ and $\mathbf{U}_1$ on a single node very efficiently (about 1 second on a general modern computer). In our MapReduce implementation, we decompose such small matrices in the MapReduce master node using Algorithm 5.

If submatrix $\mathbf{A}_1$ is not small enough, we can recursively partition it into smaller submatrices as in Figure 3.1 until the final submatrix is small enough to compute on a single node. Note that while this is conceptually a recursive computation, the number of partitioning steps (i.e., the depth of recursion) can be precomputed at the start of the matrix inversion process, so that the computation is implemented by a predefined pipeline of MapReduce jobs. In this pipeline, the input matrix is read only once and the partitioned matrix is written only once as described in Section 3.4.2.

After obtaining $\mathbf{L}_1$ and $\mathbf{U}_1$, the elements of $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ can be computed using the following two equations (for simplicity, we present the equations without pivoting since pivoting does not increase the computational complexity):

$$\begin{aligned}
[\mathbf{L}_2^{'}]_{ij} &= \frac{1}{[\mathbf{U}_1]_{ii}}\left([\mathbf{A}_3]_{ij} - \sum_{k=1}^{i-1}[\mathbf{L}_2^{'}]_{ik}[\mathbf{U}_1]_{kj}\right), \\
[\mathbf{U}_2]_{ij} &= \frac{1}{[\mathbf{L}_1]_{ii}}\left([\mathbf{A}_2]_{ij} - \sum_{k=1}^{i-1}[\mathbf{L}_1]_{ik}[\mathbf{U}_2]_{kj}\right). \qquad (3.14)
\end{aligned}$$

From these equations, it is clear that the elements in one row of $\mathbf{L}_2^{'}$ are independent of the elements in other rows. Similarly, the elements in one column of $\mathbf{U}_2$ are also independent of the elements in other columns. Therefore each row of $\mathbf{L}_2^{'}$ and each column of $\mathbf{U}_2$ can be computed independently, so we can parallelize the computation of $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$. In MapReduce, we can use one map function (multiple copies of which are executed in parallel on multiple map workers) to compute $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ in parallel. We use the reduce function of the same MapReduce job to compute $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ in parallel on the reduce workers.

After obtaining $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$, we decompose it into $\mathbf{L}_3$ and $\mathbf{U}_3$. If $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ is small enough, $\mathbf{L}_3$ and $\mathbf{U}_3$ are computed on the MapReduce master node using Algorithm 5. Otherwise, $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ is further partitioned and the computation proceeds recursively. As with $\mathbf{A}_1$, the number of partitioning steps can be precomputed and the recursive computation can be implemented by a predefined pipeline of MapReduce jobs. One difference between $\mathbf{A}_1$ and $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ is that $\mathbf{A}_1$ can be read from the input matrix and completely partitioned into as many pieces as necessary before the LU decomposition starts, while $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ can only be partitioned after $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ are computed. Note that while $\mathbf{A}_1$ and $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ may need additional partitioning, $\mathbf{A}_2$ and $\mathbf{A}_3$ never need additional partitioning due to the easily parallelizable nature of computing $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ using Equation 3.14.

The pseudocode of block LU decomposition is shown in Algorithm 6. In this algorithm, at the end of the block decomposition, the permutation matrix $\mathbf{P}$ is obtained by augmenting $\mathbf{P}_1$ and $\mathbf{P}_2$. The lower triangular matrix $\mathbf{L}$ is obtained by augmenting $\mathbf{L}_1$, $\mathbf{L}_2^{'}$, $\mathbf{L}_3$, and $\mathbf{P}_2$ and the upper triangular matrix $\mathbf{U}$ is obtained by augmenting $\mathbf{U}_1$, $\mathbf{U}_2$, and $\mathbf{U}_3$ (Figure 3.1).

Algorithm 6 Block LU decomposition.

```
1:  function BlockLUDecom($\mathbf{A}$)
2:  if A is small enough then
3:      $(\mathbf{L}, \mathbf{U}, \mathbf{P})$ = LUDecompoistion($\mathbf{A}$)
4:  else
5:      Partition $\mathbf{A}$ into $\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3, \mathbf{A}_4$
6:      $(\mathbf{L}_1, \mathbf{U}_1, \mathbf{P}_1)$ = BlockLUDecom($\mathbf{A_1}$)
7:      Compute $\mathbf{U}_2$ from $\mathbf{A}_2$, $\mathbf{U}_1$ and $\mathbf{P}_1$
8:      Compute $\mathbf{L}_2^{'}$ from $\mathbf{A}_3$ and $\mathbf{U}_1$
9:      Compute $\mathbf{B} = \mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$
10:     $(\mathbf{L}_3, \mathbf{U}_3, \mathbf{P}_2)$ = BlockLUDecom($\mathbf{B}$)
11:     $\mathbf{P}$ = Combination of $\mathbf{P}_1$ and $\mathbf{P}_2$
12:     $\mathbf{L}$ = Combination of $\mathbf{L}_1$, $\mathbf{L}_2^{'}$, $\mathbf{L}_3$, and $\mathbf{P}_2$
13:     $\mathbf{U}$ = Combination of $\mathbf{U}_1$, $\mathbf{U}_2$ and $\mathbf{U}_3$
14: end if
15: return $(\mathbf{L}, \mathbf{U}, \mathbf{P})$
```

After obtaining the lower triangular matrix $\mathbf{L}$ and the upper triangular matrix $\mathbf{U}$, we can compute the inverses of these two matrices using Equation 3.12. Inspecting this equation, we can see that a column of the matrix inverse is independent of other columns of the inverse. Therefore, the columns can be computed independently in parallel. After computing the inverses of $\mathbf{L}$ and $\mathbf{U}$, the inverse of the original matrix can be obtained by multiplying $\mathbf{U}^{-1}$ by $\mathbf{L}^{-1}$, which can also be done in parallel, and permuting the resulting matrix according to array $\mathbf{S}$. That is, $[\mathbf{A}^{-1}]_{[\mathbf{S}]_i j} = \sum_{k=1}^{n}[\mathbf{U}^{-1}]_{ik}[\mathbf{L}^{-1}]_{kj}$.

3.4 Implementation in MapReduce

In this section, we discuss the implementation of our algorithm in MapReduce. The implementation involves several steps: (1) We use the master compute node to create some control files in HDFS, which are used as input files for the mappers of all MapReduce jobs. (2) We launch a MapReduce job to recursively partition the input matrix $\mathbf{A}$. (3) We launch a series of MapReduce jobs to compute $\mathbf{L}_2^{'}$, $\mathbf{U}_2$, and $\mathbf{B} = \mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ for the different partitions of $\mathbf{A}$ as in Algorithm 6. Matrices $\mathbf{L}_1$ and $\mathbf{U}_1$ are computed in the master nodes of these jobs if $\mathbf{A}_1$ is small enough to be computed in a single node, otherwise they are computed in another MapReduce pipeline. Similarly, matrices $\mathbf{L}_3$ and $\mathbf{U}_3$ are computed in the master nodes if $\mathbf{B}$ is small, otherwise in another MapReduce pipeline. (4) We launch a final MapReduce job to produce the final output by computing $\mathbf{U}^{-1}$, $\mathbf{L}^{-1}$ and $\mathbf{A}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}$.

The number of MapReduce jobs required to compute the LU decomposition (Step 3) depends on the order (n) of matrix $\mathbf{A}$ and a bound value n_b. The bound value n_b is the maximum order of a matrix that can be LU decomposed on a single node (in our case the MapReduce master node). The number of MapReduce jobs is given by $2^{\lceil \log_2 \frac{n}{n_b} \rceil}$. Thus, we have a pipeline of MapReduce jobs as shown in Figure 3.2.

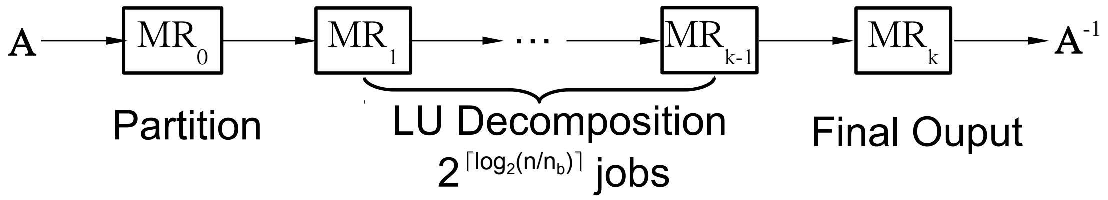


Figure 3.2: MapReduce pipeline for matrix inversion.

The bound value should be set so that the time to LU decompose a matrix of order n_b on the master node is approximately equal to the constant time required to launch a MapReduce job. If the running time to decompose a matrix of order n_b on the master node is significantly less than the launch time of a MapReduce job, there will be more MapReduce jobs than necessary and we can increase n_b to reduce the number of MapReduce jobs and also reduce the total running time of LU decomposition. On the other hand, if the running time on the master node is significantly larger than the launch time of a MapReduce job, the LU decomposition on the master node becomes the bottleneck and we can improve performance by reducing n_b so that we partition the matrix into smaller submatrices. In our experiments, we set n_b to 3200.

Next, we discuss the steps of the implementation. For the purpose of this discussion, Figure 3.3 shows an example of how the input matrix is partitioned, and Figure 3.4 shows the HDFS directory structure used by our implementation. The HDFS directory "Root" is the work directory of our implementation, and the file "a.txt" in "Root" is the input matrix.

3.4.1 Map Input Files on MapReduce

In the first step, we create m_0 files on the Master node, where m_0 is the number of compute nodes. These files are used as input files for the following MapReduce tasks and they are stored in "Root/MapInput/" (see the purple labels in Figure 3.4), where "Root/" is the root directory for our implementation. Each file only contains one integer, i.e., the first file A.0 contains 0, the second file A.1 contains 1, ..., and the last file A.$m_0 - 1$ contains $m_0 - 1$. The mappers use these files to control the computation, and they produce the output required for inverting the input matrix by writing directly to HDFS.

3.4.2 Data Partitioning for LU Decomposition

In this section we discuss how the input matrix is partitioned for LU decomposition and how the different partitions flow through the MapReduce pipeline. We also discuss how the input, intermediate, and output data files are stored in HDFS to improve I/O efficiency. In Figure 3.4, the green labels indicate which process produces each result or intermediate data file. For example "Map 0" means that the result is produced by the mappers in the first MapReduce job, and "Master 0" means that the result is produced by the master node of that job. Therefore, these labels also represent the computation process of the LU decomposition.

We launch a MapReduce job to partition matrix $\mathbf{A}$. This is a map-only job where the mappers do all the work and the reduce function does nothing. This is the only partitioning job, and it recursively partitions $\mathbf{A}$ into as many submatrices as needed, according to the depth of recursion implied by n and n_b. The mappers of this job read their input files from "Root/MapInput/". The integer read by each mapper tells that mapper which rows of the input matrix in the HDFS file "Root/a.txt" to read and partition. In order to improve I/O performance, each map function reads an equal number of consecutive rows

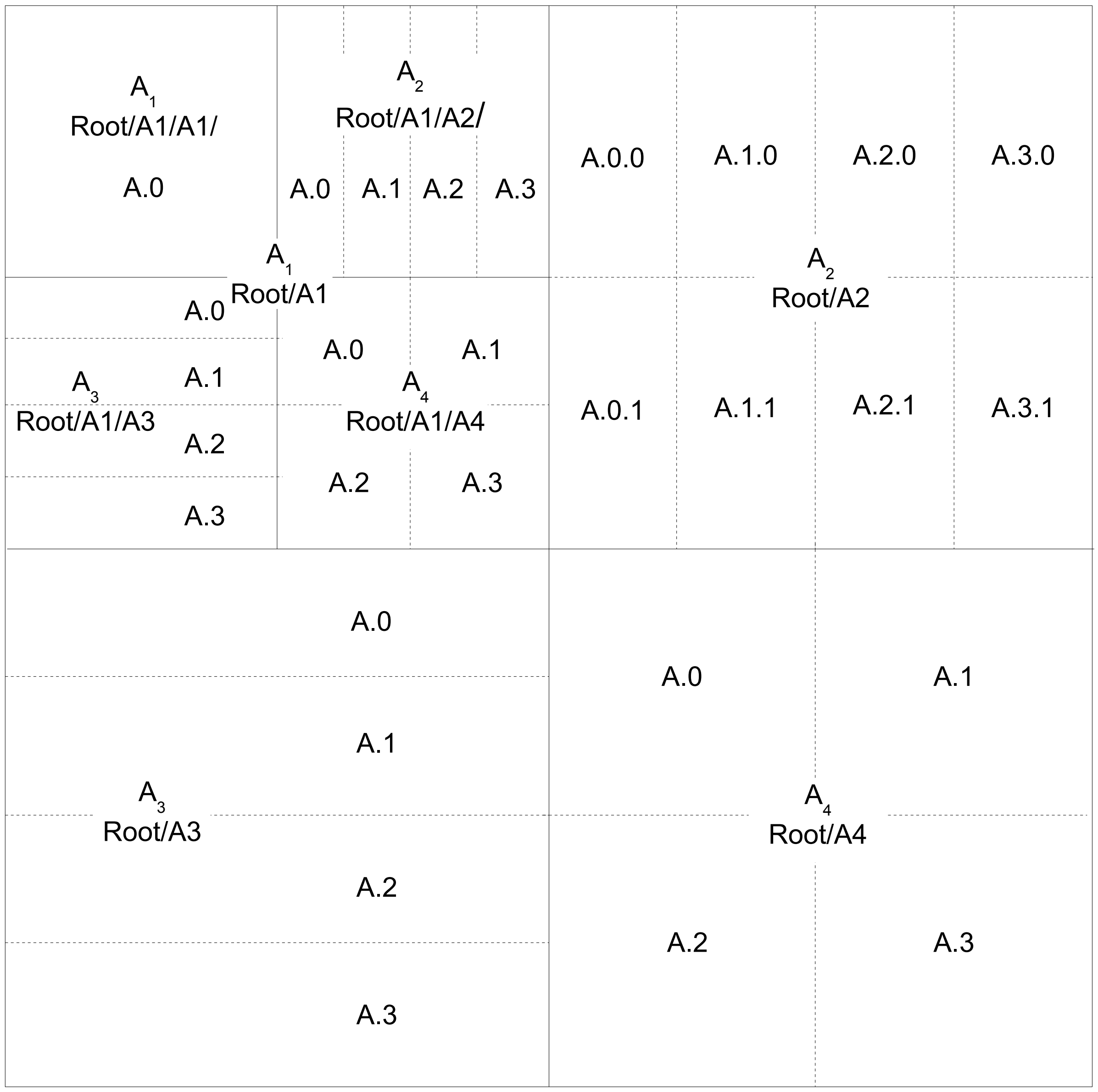


Figure 3.3: Matrix partitioning and data flow for LU decomposition.

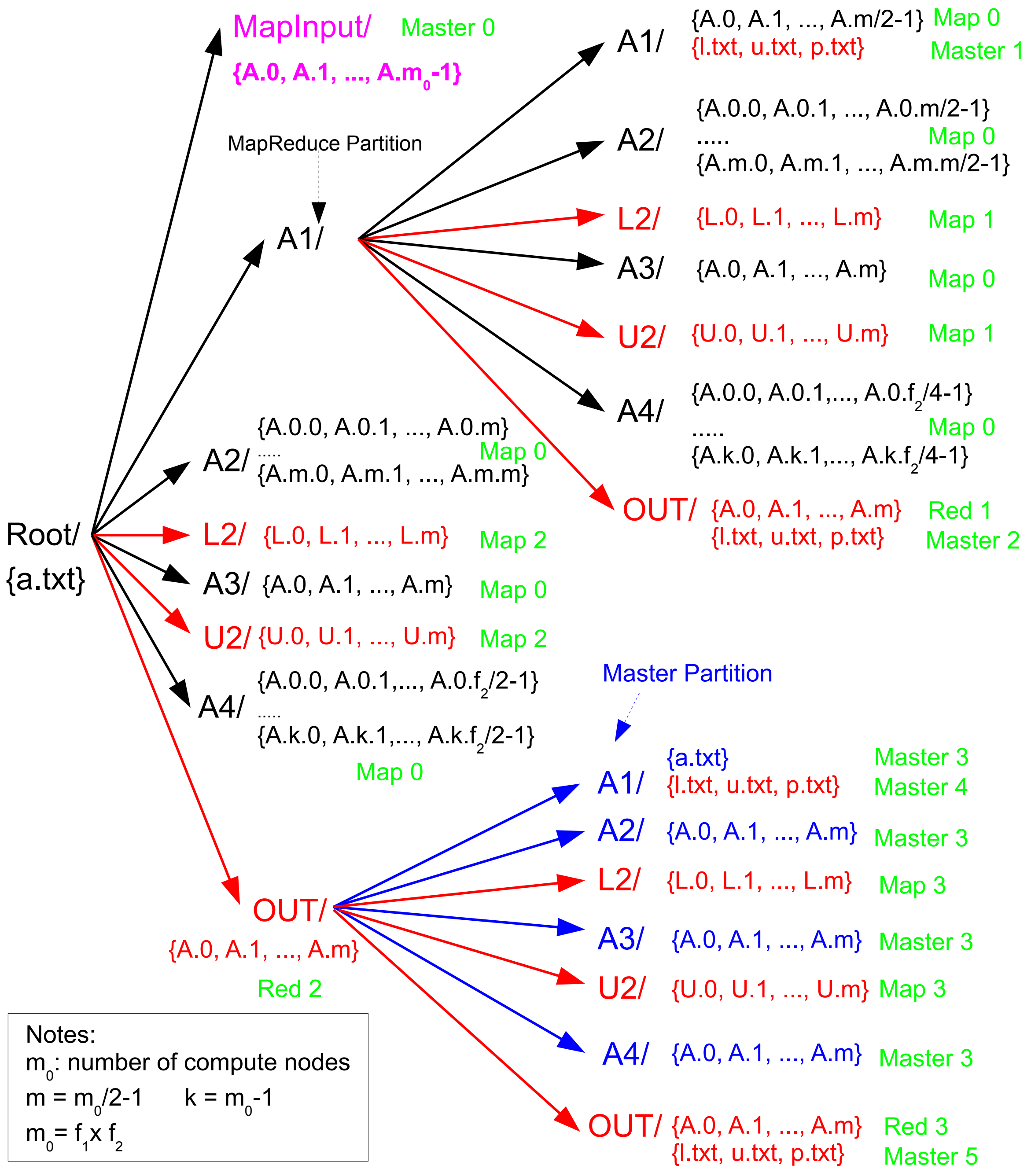


Figure 3.4: HDFS directory structure for LU decomposition.

from this file to increase I/O sequentiality. Worker j (the mapper assigned input file "Root/MapInput/A.j") reads rows $r_1 = \frac{nj}{m_0}$ to $r_2 = r_1 + \frac{n}{m_0}$ (exclusively). Each block is written to "Root/A1" and "Root/A2" if the block is in the first half of $\mathbf{A}$. Otherwise it is written to "Root/A3" and "Root/A4".

In order to improve I/O efficiency while reading submatrices from disk in subsequent MapReduce jobs, each submatrix, whether $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$, or $\mathbf{A}_4$, is split into multiple parts, each of which is stored in a separate file. This ensures that there will never be multiple mappers that simultaneously read the same file. For example, $\mathbf{A}_3$ is stored in $\frac{m_0}{2}$ files because we use only half the compute nodes to compute $\mathbf{L}_2^{'}$ using $\mathbf{A}_3$, while the other half are used to compute $\mathbf{U}_2$ using $\mathbf{A}_2$ (details later). Therefore, $m = \frac{m_0}{2} - 1$ in Figure 3.4. This approach also ensures that no two mappers write data into the same file, thereby eliminating the need for synchronization between mappers and improving I/O efficiency. Mappers and reducers in subsequent jobs also write their data into independent files, so synchronization on file writes is never required and I/O efficiency is maintained. The separate files written by worker nodes are shown in Figure 3.4, for example, "L2/L.1".

In Figure 3.3, which shows an example of matrix $\mathbf{A}$ partitioned by four mappers, the square blocks surrounded by solid lines are submatrices, while the rectangular blocks divided by dashed lines are separate files storing these submatrices. In Figure 3.4, the black labels within braces are the file names of partitioned submatrices. The value f_2 is the maximum factor of m_0 less than $\sqrt{m_0}$ (see the discussion in Section 3.5.2). The depth d of the directory structure equals the depth of data partitioning (2 in Figure 3.4) given by $\lceil \log_2 \frac{n}{n_b} \rceil$. The pseudocode of the data partitioning algorithm for LU decomposition is given in Algorithm 7. This listing shows one map function partitioning the block of data in rows r_1 to r_2.

Partitioning submatrix $\mathbf{B} = \mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ is handled differently from the input matrix. This submatrix is produced by m_0 reducers of a MapReduce job and stored in m_0 files. These files are not read simultaneously by many mappers or reducers, so instead of materializing the data partitions of $\mathbf{B}$ after this submatrix is produced, we only record the indices of the beginning and ending row, and the beginning and ending column, of each partition in this submatrix. We also record the names of the files storing this data. Using this approach, the files in "Root/OUT/A1", "Root/OUT/A2", "Root/OUT/A3", and "Root/OUT/A4" (indicated by blue labels in Figure 3.4) are very small (in general, less than 1 KB). The running time to partition $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ with such a method is quite short (less than 1 second). Therefore, it is not necessary to launch a MapReduce task to partition this matrix. In our implementation, we partition this matrix in the master node.

Algorithm 7 Data partitioning for LU decomposition.

```
function Partition($\mathbf{A}$, $r_1$, $r_2$, $n$, $n_b$, $m$, $f_1$, $f_2$, "path")
/* $\mathbf{A}$ is the original matrix. $r_1$ is the index of the beginning row to be saved by this function and $r_2$ is the index of the ending row. $n$ is the order of the matrix. $n_b$ is the bound value for data partitioning. $m$ is the number of map workers partitioning the current submatrix (e.g., the submatrix in the directory Root/A1 shown in Figure 3.3), $f_1$ and $f_2$ indicate that $\mathbf{A}_4$ is partitioned to $f_1 \times f_2$ blocks according to the optimization in Section 3.5.2. */
if $n < n_b$ then
    /* Save A1*/
    Save $[\mathbf{A}]_{[r_1 \dots r_2][0 \dots n]}$ to "path/A1/A.$\frac{r_1 m}{n}$"
else
    if $r_1 < \frac{n}{2}$ then
        Partition($\mathbf{A}$, $r_1$, $r_2$, $\frac{n}{2}$, $n_b$, $\frac{m}{2}$, $f_1$, $f_2$, "path/A1")
        /* Save A2 */
        for $i = 0$ to m - 1 do
            Save $[\mathbf{A}]_{[r_1 \dots r_2][\frac{n}{2} \dots n]}$ to "path/A2/A.$i$.$\frac{rm}{n}$"
        end for
    else
        /* Save A3 */
        for $i = 0$ to $\frac{(r_2 - r_1)m}{2n} - 1$ do
            $k = \frac{(2r_1 - n)m}{4n} + i$
            Save $[\mathbf{A}]_{[r_1 \dots r_1 + \frac{2mi}{n}][0 \dots \frac{n}{2}]}$ to "path/A3/A.$k$"
        end for
        /* Save A4 */
        for $j = 0$ to $f_2 - 1$ do
            $l = (\frac{2r_1 f_1}{n} - f_1) f_2 + j$
            for $i = 0$ to $\frac{2(r_2 - r_1) f_1}{n} - 1$ do
                Save $[\mathbf{A}]_{[r_1 \dots r_1 + \frac{(i+1)n}{2f_1}][\frac{n}{2} \dots \frac{n}{2} + \frac{(j+1)n}{2f_2}]}$ to "path/A4/A.$l$.$i$"
            end for
        end for
    end if
end if
```

3.4.3 LU Decomposition Using MapReduce

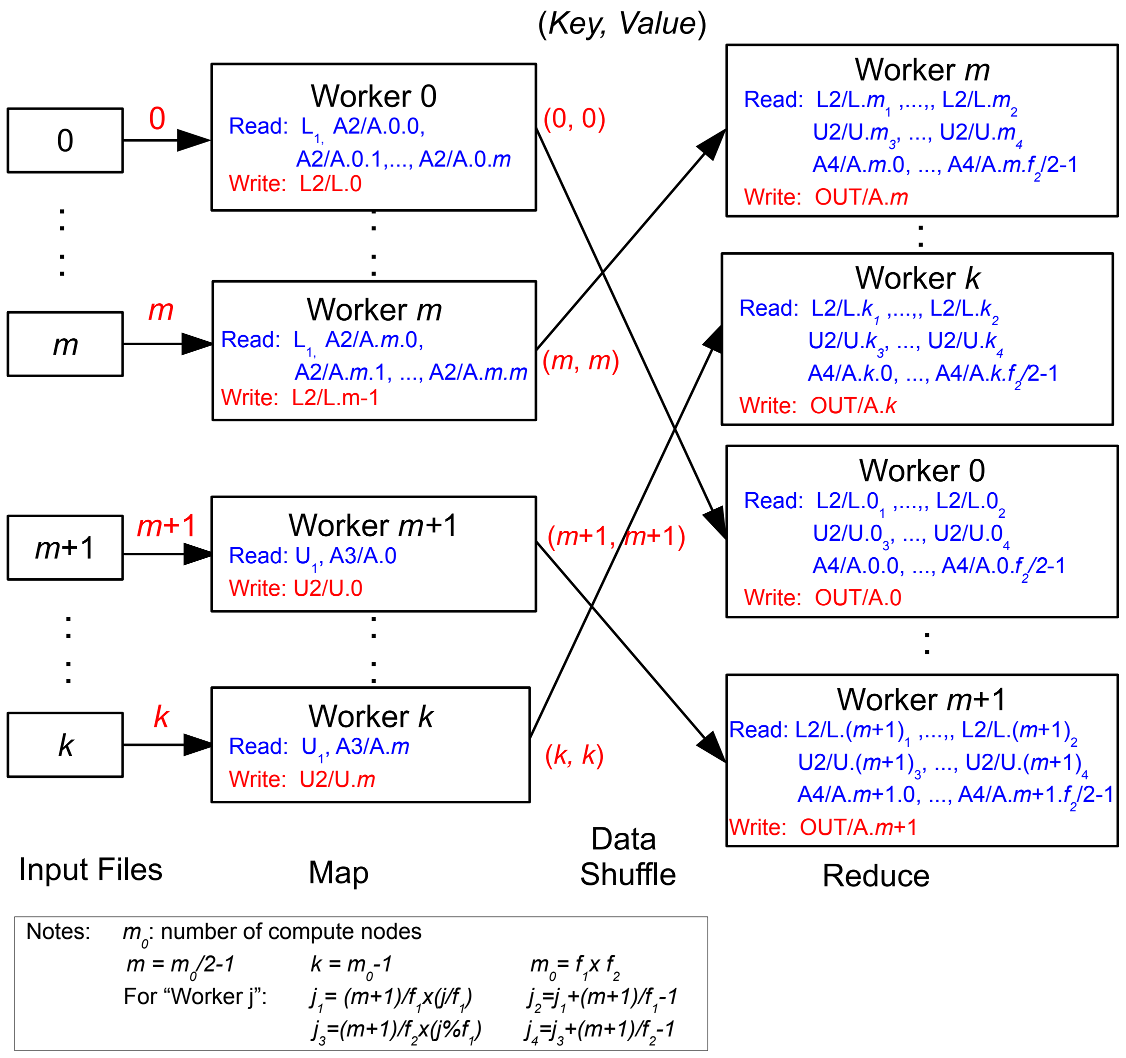
After partitioning $\mathbf{A}$, we use Algorithm 6 to compute the LU decomposition. Since $\mathbf{A}$ has been partitioned, line 5 is ignored, and $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$, and $\mathbf{A}_4$ are read from HDFS files. We launch one MapReduce job for lines 7–9 of this algorithm. One MapReduce job is sufficient regardless of the size of the input block. The map function of this job computes $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$, while the reduce function computes $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ (red labels in Figure 3.4). In our implementation, we use half of the mappers to compute $\mathbf{L}_2^{'}$ and the other half to compute $\mathbf{U}_2$, since computing $\mathbf{L}_2^{'}$ has the same computational complexity as computing $\mathbf{U}_2$. Each mapper reads an input file from the directory "Root/MapInput/". If the value in this file is not larger than $m = \frac{m_0}{2} - 1$, the mapper computes part of $\mathbf{L}_2^{'}$. Otherwise it computes part of $\mathbf{U}_2$. This is illustrated in Figure 3.5.

If worker j is computing part of $\mathbf{L}_2^{'}$, this worker is assigned the file "$\mathbf{A}.j$". The worker reads $\mathbf{L}_1$ from HDFS directory "Root/A1" and $\mathbf{A}_{2.j}$ from files "Root/A2/A.j.0, Root/A2/A.j.1, ..., Root/A.$j.m$", and computes one part of $\mathbf{L}_2^{'}$, which is written to "Root/L2/L.j".

Each mapper in this MapReduce job emits one (*key, value*) pair containing (j, j), where j is the value read by the mapper from its input file. These (*key, value*) pairs are used to control which part of $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$ each reducer should compute. In our implementation, we use block wrap for matrix multiplication (Section 3.5.2), so worker j computes the j-th block of $\mathbf{A}_4 - \mathbf{L}_2^{'}\mathbf{U}_2$. The detailed files read and written are shown in Figure 3.5. It should be noted that after obtaining $\mathbf{L}_2^{'}$ and $\mathbf{P}_2$, $\mathbf{L}_2$ can be easily obtained by permuting $\mathbf{L}_2^{'}$ based on the permutation matrix $\mathbf{P}_2$. Therefore in our implementation, $\mathbf{L}_2$ is constructed only as it is read from HDFS.

3.4.4 Time Complexity of LU Decomposition

The time complexity of our LU decomposition is shown in Table 3.1. We also present the time complexity of the algorithm used in ScaLAPACK (refer Section 3.6.5 for a detailed comparison of our algorithm with ScaLAPACK). In our algorithm, all data is written to HDFS, such that the amount of data read from HDFS is similar to the amount of data transferred between different compute nodes. In the MPI implementation using ScaLAPACK (Section 3.6.5), the data is only read once, but data transfer between the master and workers is very large. The data transfer in our algorithm is better than ScaLAPACK.



(Key, Value)
0
m
m+1
k
Worker 0
Read: L1, A2/A.0.0, A2/A.0.1,..., A2/A.0.m
Write: L2/L.0
Worker m
Read: L1, A2/A.m.0, A2/A.m.1, ..., A2/A.m.m
Write: L2/L.m-1
Worker m+1
Read: U1, A3/A.0
Write: U2/U.0
Worker k
Read: U1, A3/A.m
Write: U2/U.m
(0, 0)
(m, m)
(m+1, m+1)
(k, k)
Worker m
Read: L2/L.m1 ,..., L2/L.m2 U2/U.m3, ..., U2/U.m4 A4/A.m.0, ..., A4/A.m.f2/2-1
Write: OUT/A.m
Worker k
Read: L2/L.k1 ,..., L2/L.k2 U2/U.k3, ..., U2/U.k4 A4/A.k.0, ..., A4/A.k.f2/2-1
Write: OUT/A.k
Worker 0
Read: L2/L.01 ,..., L2/L.02 U2/U.03, ..., U2/U.04 A4/A.0.0, ..., A4/A.0.f2/2-1
Write: OUT/A.0
Worker m+1
Read: L2/L.(m+1)1 ,..., L2/L.(m+1)2 U2/U.(m+1)3, ..., U2/U.(m+1)4 A4/A.m+1.0, ..., A4/A.m+1.f2/2-1
Write: OUT/A.m+1
Input Files
Map
Data Shuffle
Reduce
Notes: m0: number of compute nodes
m = m0/2-1
k = m0-1
m0 = f1 x f2
For "Worker j":
j1 = (m+1)/f1 x (j/f1)
j2 = j1+(m+1)/f1-1
j3 = (m+1)/f2 x (j%f1)
j4 = j3+(m+1)/f2-1

Figure 3.5: MapReduce job to compute $\mathbf{L}_2'$, $\mathbf{U}_2$, and $\mathbf{A}_4 - \mathbf{L}_2'\mathbf{U}_2$.

Algorithm	Write	Read	Transfer	Mults	Adds
Our Algorithm	$\frac{3}{2}n^2$	$(l+3)n^2$	$(l+3)n^2$	$\frac{1}{3}n^3$	$\frac{1}{3}n^3$
ScaLAPCK	n^2	n^2	$\frac{2}{3}m_0n^2$	$\frac{1}{3}n^3$	$\frac{1}{3}n^3$

Table 3.1: Time complexity of our LU decomposition algorithm, in comparison with the ScaLAPACK algorithm, for $n \times n$ matrix on m_0 compute nodes, where $m_0 = f_1 \times f_2$ and $l = \frac{1}{4}(m_0 + 2f_1 + 2f_2)$.

3.4.5 Triangular Matrix Inversion and Final Output

One MapReduce job is used to compute the inverses of the triangular matrices **L** and **U** and the product of their inverses. In the map phase, the inverses of the triangular matrices are computed using Equation 3.12. Half of the mappers compute the inverse of **L** and the other half compute the inverse of **U**. In order to balance load, the i-th node is used to compute the $(k \times m_0 + i)$-th column of $\mathbf{L}^{-1}$ if i is less than $\frac{n}{2}$. If $i \geq \frac{n}{2}$, the node computes the $(k \times \frac{n}{m_0} + i - \frac{n}{2})$-th row of $\mathbf{U}^{-1}$, where k is an integer ($0 \leq k < \frac{n-i}{m_0}$ for $i < \frac{n}{2}$ or $0 \leq k < \frac{3n-2i}{2m_0}$ for $i \geq \frac{n}{2}$).

In the reduce phase, the product of these two inverses $\mathbf{U}^{-1}\mathbf{L}^{-1}$ is derived. Each reducer reads a number of columns of $\mathbf{L}^{-1}$ and a number of rows of $\mathbf{U}^{-1}$, and multiplies these two parts. In order to reduce read I/O, block wrap is used for matrix multiplication (Section 3.5.2). In order to balance load, instead of partitioning the final matrix into $f_1 \times f_2$ blocks (see Section 3.5.2), each of which contains consecutive rows and consecutive columns, the matrix is partitioned into grid blocks, each of which contains discrete rows and discrete columns. Worker j computes the product of row $\frac{m_0}{f_1}k_1 + j_1$ of $\mathbf{U}^{-1}$ and column $\frac{m_0}{f_2}k_2 + j_2$ of $\mathbf{L}^{-1}$, where $j_1 = \frac{j}{f_1}$, $j_2 = j \bmod f_1$. Here k_1 is any of the non-negative integers that satisfy $\frac{m_0}{f_1}k_1 + j < m_0$, and k_2 is any of the non-negative integers that satisfy $\frac{m_0}{f_2}k_2 + j < m_0$. As in the implementation of LU decomposition, all outputs, $\mathbf{L}^{-1}, \mathbf{U}^{-1}$ and $\mathbf{A}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}$, are written to HDFS. The map function only emits integers (j, j) to control the reduce tasks, and does not emit outputs. Table 3.2 shows the time complexity and data transfer of our matrix inversion algorithm, and the corresponding values in ScaLAPACK.

3.5 Optimizations of the MapReduce Implementation

In this section, we present optimizations that we use to speed up our implementation. Two of these optimizations are to reduce the read and write I/O, which improves the

Algorithm	Write	Read	Transfer	Mults	Adds
Our Algorithm	$2n^2$	ln^2	$(l+2)n^2$	$\frac{2}{3}n^3$	$\frac{2}{3}n^3$
ScaLAPCK	n^2	m_0n^2	m_0n^2	$\frac{2}{3}n^3$	$\frac{2}{3}n^3$

Table 3.2: Time complexity of our triangular matrix inversion and final matrix inversion, in comparison with the ScaLAPACK algorithm, for $n \times n$ matrix on m_0 compute nodes, where $m_0 = f_1 \times f_2$ and $l = \frac{1}{2}(m_0 + f_1 + f_2)$.

scalability of our algorithm. These optimizations are storing intermediate data in separate files (Section 3.5.1) and block wrap (Section 3.5.2). A third optimization improves memory access locality by storing the transpose of the upper triangular matrix in both HDFS and memory (Section 3.5.3).

3.5.1 Storing Intermediate Data in Separate Files

In order to reduce the amount of read and write I/O in different MapReduce jobs, we do not combine the results, such as $\mathbf{L}_1$, $\mathbf{L}_2$ and $\mathbf{L}_3$, in any stage. The results are located in many different files as shown in Section 3.4. Algorithm 6 writes all outputs into HDFS as separate files and skips lines 11–13. The total number of files for the final lower triangular or upper triangular matrix is $N(d) = 2^d + \frac{m_0}{2}(2^d - 1)$, where m_0 is the number of compute nodes, and d is the recursive dept which is constrained by $2^d < \frac{n}{n_b} < 2^{d+1}$, where n is the order of the matrix and n_b is the bound value. For example, given a square matrix $\mathbf{A}$ with $n = 2^{15}$, $n_b = 2^{11} = 2048$, and $m_0 = 64$, the recursive depth d is 4 and the final lower triangular matrix $\mathbf{L}$ is stored in $N(d) = 496$ files. In our implementation, these files are read into memory recursively.

Because combining intermediate files can only happen on one compute node, such as the master node, and other compute nodes have to wait until combination is completed, combining intermediate files significantly increases the running time. Therefore, storing intermediate results in separate files significantly improves performance, as we demonstrate in Section 3.6.3.

3.5.2 Block Wrap for Matrix Multiplication

Our algorithm requires multiplying two matrices at different stages, for example $\mathbf{L}_2'$ and $\mathbf{U}_2$, or $\mathbf{U}^{-1}$ and $\mathbf{L}^{-1}$. A simple and easy-to-implement way to multiply two matrices while reducing the amount of data read is to use the block method for matrix multiplication. In general, in order to compute $\mathbf{L}_2^{'}\mathbf{U}_2$, each compute node can read a number of rows, e.g., i-th to j-th rows, of $\mathbf{L}_2^{'}$ and the entire matrix $\mathbf{U}_2$. This compute node can then compute the i-th to j-th rows of $\mathbf{L}_2^{'}\mathbf{U}_2$. If the number of compute nodes is m_0, the amount of data read in each node is $(1 + \frac{1}{m_0})n^2$ and the total data read is $(m_0 + 1)n^2$.

There is a better method to multiply two matrices, called the *block wrap* method [14], which reduces the amount of data read. In this method, $\mathbf{L}_2^{'}$ is divided into f_1 blocks, each of which contains $\frac{n}{f_1}$ consecutive rows, while $\mathbf{U}_2$ is divided into f_2 blocks, each of which contains $\frac{n}{f_2}$ consecutive columns. Using this partitioning, every block of $\mathbf{L}_2^{'}$ will need to be multiplied by every block of $\mathbf{U}_2$, and the final matrix is partitioned into $f_1 \times f_2$ blocks. Each of these blocks is computed by one compute node. That is, each compute node reads $\frac{n}{f_1}$ rows of $\mathbf{L}_2^{'}$ and $\frac{n}{f_2}$ columns of $\mathbf{U}_2$ (one block from each matrix) and computes the product of these two block. f_1 and f_2 are chosen so that $m_0 = f_1 \times f_2$. The data read in each compute node is $(\frac{1}{f_1} + \frac{1}{f_2})n^2$, and the total data is $(f_1 + f_2)n^2$, which is significantly less than $(m_0 + 1)n^2$. In order to obtain the minimum data read, we compute f_1 and f_2 from n such that $|f_1 - f_2|$ is as small as possible. That is, we choose $f_2 \leq f_1$, and there is no other factor of m_0 between f_1 and f_2. For example, given 64 nodes, in the naive algorithm each node reads data of size $\frac{65}{64}n^2$, and the total data read for all 64 nodes is $65n^2$. Using the block wrap method and $f_1 = f_2 = 8$, each node reads data of size $\frac{1}{4}n^2$, and the total data read for all nodes is $16n^2$, much better than the naive algorithm.

3.5.3 Storing Transposed U Matrices

In general, matrices $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ are linearized in row-major order both in memory and in HDFS. The product of $\mathbf{L}_2^{'}$ and $\mathbf{U}_2$ is computed as follows:

$$[\mathbf{L}_2^{'}\mathbf{U}_2]_{ij} = \sum_{k=1}^{n}[\mathbf{L}_2^{'}]_{ik} \times [\mathbf{U}_2]_{kj} \tag{3.15}$$

However, when the order of the matrices n is large, each read of an element from $\mathbf{U}_2$ will access a separate memory page, potentially generating a TLB miss and a cache miss. If a page can hold k data items, this access pattern can generate up to $n^3 + \frac{k+1}{k}n^2$ misses for data read and matrix multiplication.

In our implementation, the upper triangular matrix is always stored in a transposed fashion, i.e., we store $\mathbf{U}^T$ instead of $\mathbf{U}$. The product of $\mathbf{L}_2^{'}$ and $\mathbf{U}_2^T$ can be computed as follows:

$$[\mathbf{L}_2^{'}\mathbf{U}_2]_{ij} = \sum_{k=1}^{n}[\mathbf{L}_2^{'}]_{ik} \times [\mathbf{U}_2^T]_{jk} \tag{3.16}$$

Matrix	Order	Elements (Billion)	Text (GB)	Binary (GB)	Number of MapReduce Jobs
$\mathbf{M}_1$	20480	0.42	8	3.2	9
$\mathbf{M}_2$	32768	1.07	20	8	17
$\mathbf{M}_3$	40960	1.68	40	16	17
$\mathbf{M}_4$	102400	10.49	200	80	33
$\mathbf{M}_5$	16384	0.26	5	2	9

Table 3.3: Five matrices used for the experiments.

The number of misses can be reduced to $\frac{n^3}{k} + \frac{2n^2}{k}$, which is significantly less than the previous implementation and can substantially improve performance.

3.6 Experimental Evaluation

In this section, we present the experimental evaluation of our algorithm, implementation, and optimizations. This section is organized as follows: in Section 3.6.1, we introduce the experimental environment and the data used for the experiments. The scalability of our algorithm is demonstrated in Section 3.6.2 and again in Section 3.6.4. The experimental evaluation of the optimizations of storing intermediate data in separate files and block wrap is given in Section 3.6.3. Finally, we show a performance comparison of our algorithm with the ScaLAPACK package in Section 3.6.5.

3.6.1 Experimental Environment

Our algorithm has been implemented on Hadoop 1.1.1, which was the latest stable version at the time the work was done. All experiments were performed on medium instances of Amazon's Elastic Compute Cloud (EC2) [25], except the largest matrix $\mathbf{M}_4$, for which large instances are used. Each medium instance has 3.7 GB of memory and 1 virtual core with 2 EC2 compute unit, where each EC2 compute unit has a similar performance as one 2007-era 1.0−1.2 GHz AMD Opteron or Xeon processor.

We use five matrices in our experiments. All of these matrices were randomly generated using the Random class in Java. Details about the matrices are shown in Table 3.3, which shows the order of each matrix, the number of elements (data type double), the size of the matrix in text format, and the size in binary format. Recall that the bound value n_b

used in our experiments is 3200. Table 3.3 shows, for this value of n_b, the total number of MapReduce jobs required for inverting each matrix.

In our implementation, the input file is stored in binary format (double data type, 64 bits), while the final output is stored in text format, i.e., the size in binary format and the size in text format listed in the table are the size of input data and the size of output data respectively. In our implementation, all intermediate data are stored in double data type (64 bits). For matrices $\mathbf{M}_1$, $\mathbf{M}_2$, and $\mathbf{M}_3$, we checked the numerical accuracy of the final inverse by computing the difference between $\mathbf{AA}^{-1}$ and $\mathbf{I}_n$. We found that the absolute value of any element in $\mathbf{AA}^{-1} - \mathbf{I}_n$ is less than 10^{-5}. Therefore, we can conclude that the double data type is accurate enough for our block LU decomposition based matrix inversion algorithm. We did not verify the numerical accuracy for $\mathbf{M}_4$ due to the long running time.

3.6.2 Algorithm Scalability

In this section, we investigate the scalability of our proposed algorithm. The running time versus the number of EC2 instances is shown in Figure 3.6. One ideal scalable line has been over-plotted on this figure in order to demonstrate the scalability of our algorithm. The deviation of our algorithm from ideal scalability when the number of nodes is high is due to the constant launch time of MapReduce jobs, since our algorithm uses multiple MapReduce jobs. However, we also note that the number of MapReduce jobs is proportional to n, where n is the order of the matrix, while the running time is proportional to n^3. Therefore, we can expect that the larger the matrix, the better the algorithm scalability. This is evident in Figure 3.6.

We investigated improving scalability by using systems that support iterative MapReduce computations, such as HaLoop [8]. However, we found that HaLoop and similar systems do not reduce the launch time of MapReduce jobs. HaLoop maintains intermediate state between MapReduce jobs, which is not useful for our algorithm.

The largest matrix $\mathbf{M}_4$ is used to further test the scalability limits of our algorithm (Section 3.6.4), and the smallest matrix $\mathbf{M}_5$ is used to evaluate our optimizations (next section).

3.6.3 Evaluating the Optimizations

Our first proposed optimization is storing intermediate data in separate files. Without this optimization, we combine all separate files of $\mathbf{L}$, such as $\mathbf{L}_1, \mathbf{L}_{2,1}, \mathbf{L}_{2,2}, \mathbf{L}_{2,3}, \mathbf{L}_{2,4}$ and

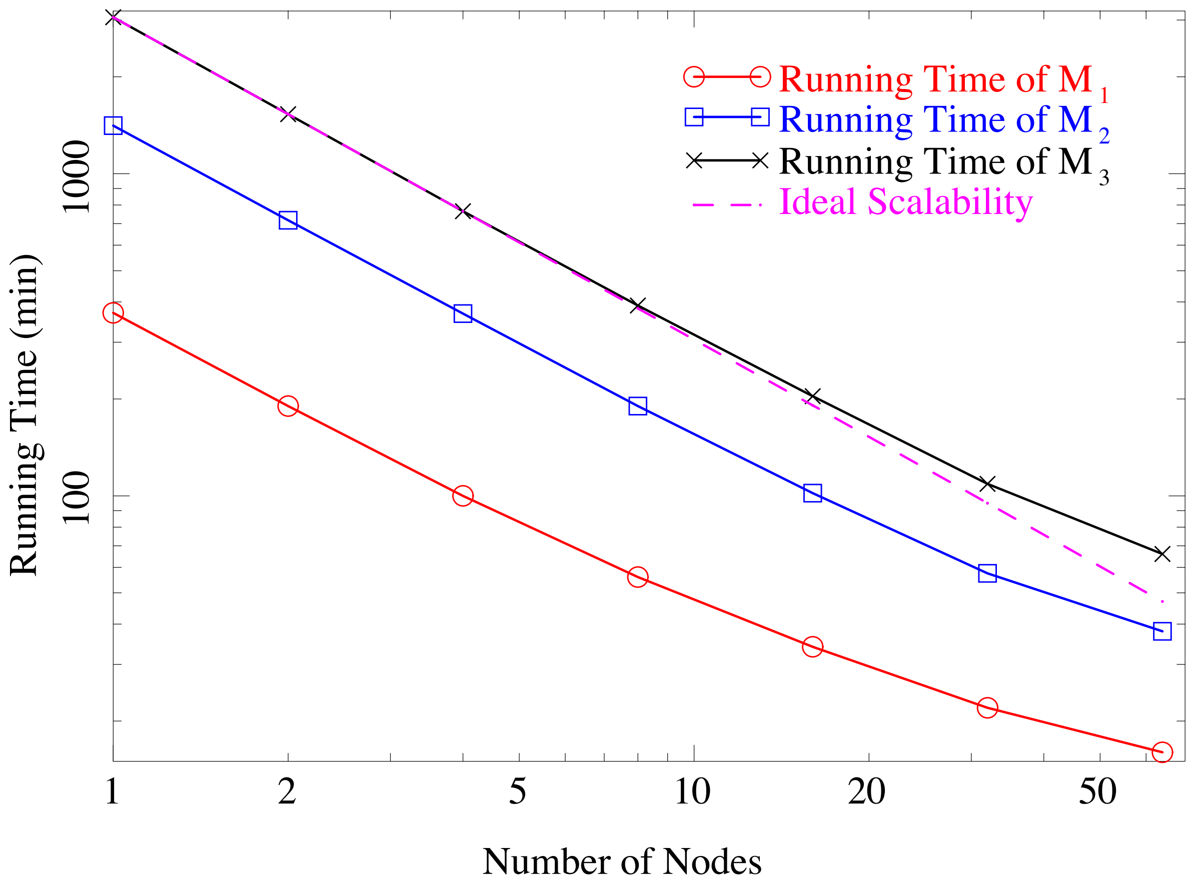


Figure 3.6: The scalability of our algorithm, in comparison with ideal scalability (purple line), which is defined as $T(n) = T(1)/n$, where $T(n)$ is the running time on n medium EC2 instances.

$\mathbf{L}_3$ in Figure 3.4, as well as all separate files of $\mathbf{U}$, in each MapReduce job in our iterative MapReduce process. The combination happens in the master node and the combined file is written by that node into HDFS. Since the combination is done in one node, it takes a constant time to combine the files independent on the number of the compute nodes. Therefore, we can expect that the benefit of this optimization increases as the number of compute nodes increases since the running time gets smaller.

To validate this, we conduct an experiment with matrix $\mathbf{M}_5$ in which we compare the time taken by our optimized algorithm to the time taken by the algorithm that combines $\mathbf{L}$ and $\mathbf{U}$ files. The ratio of the unoptimized running time to the optimized running time

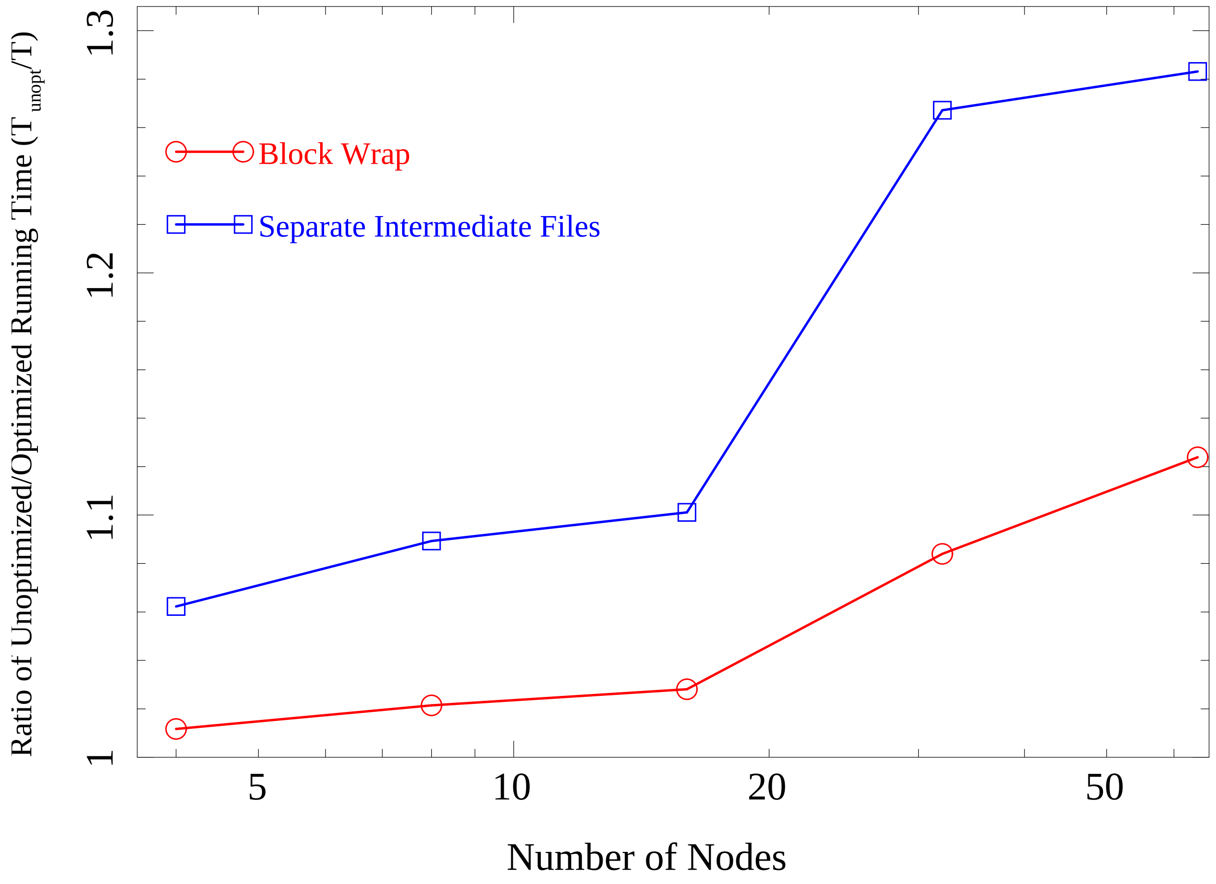


Figure 3.7: The running time of the optimized algorithm compare to the algorithm without the separate intermediate file optimization (blue), and without the block wrap optimization (red).

for 4–64 nodes is shown in Figure 3.7. The figure shows the unoptimized version to be close to 30% slower in some cases, demonstrating the importance of this optimization.

As mentioned in Section 3.5.2, the block wrap method can significantly reduce the amount of read I/O, which improves performance. In this section, we also use the smallest matrix $\mathbf{M}_5$ to validate this optimization. As before, we measure the running time without this optimization on 4–64 compute nodes and then compare this to the running time with the optimization. The improvement of this optimization is also shown in Figure 3.7. The figure shows that the larger the number of compute nodes, the larger the improvement in performance.

We did not evaluate the third optimization (transpose storing) in our experiments, but our experience is that this optimization greatly improves the performance of our algorithm, by a factor of 2–3.

3.6.4 Scaling to a Very Large Matrix

In this section, we study the ability of our algorithm to invert a very large matrix, namely $\mathbf{M}_4$, which is a matrix of order 102400. We measure the running time on 128 Amazon EC2 large instances, each of which has two medium CPU cores, for a total of 256 medium CPU cores. A medium CPU core has performance similar to two 2007-era 1.0–1.2 GHz AMD Opteron or Xeon processors.

We executed two runs of our algorithm to invert this large matrix. In the first run, it took about 8 hours to solve the problem. During this run, one mapper computing the inverse of a triangular matrix failed and this mapper did not restart until one of the other mappers finished. This increased the running time. However, this failure recovery is a good demonstration of the benefit of using a fault tolerant framework like MapReduce for large scale problems. In the second run, there were no failures and it took about 5 hours to invert the matrix.

The large matrix is about 80 GB in size in binary representation. Our algorithm on the EC2 large instances writes more than 500 GB of data and reads more than 20 TB of data in the 33 MapReduce jobs required to invert the matrix. In this experiment, there were more than 10^{15} double precision additions and 10^{15} double precision multiplications. This illustrates the scale at which our algorithm operates, and shows that the matrix inversion problem is both compute intensive and data intensive.

We also used 64 medium EC2 instances to invert this matrix. It took about 15 hours in this case to invert the matrix. Analyzing the scalability of the medium instances compared to the large instances, we see that the medium instances show better scalability. The reasoning is as follows. Assume for simplicity that each medium instance core has similar compute performance to a large instance core. When we used 128 large EC2 instances we were using 256 cores, whereas when we used 64 medium instances, we were using 64 cores. Thus, we have four times as many cores when using large instances. Therefore, if our algorithm has ideal scalability, the running time in large instances should be $15/4 = 3.8$ hours (four times the cores should result in $\frac{1}{4}$ the running time). However, the running time we observed on large instances (5 hours) is longer than this time assuming ideal scalability. There are two possible reasons related to EC2 for the running time being longer than expected. The first reason is that we found that the performance variance between different large EC2 instances is high, even though the instances are supposed to have similar performance. The second reason is that the data read speed on some large instances is less than the speed on the medium instances. We found that the speed of copying files between large instances is around 30–60 MB/s, while the speed of copying files between medium instances is around 60 MB/s.

Setting this difference in scalability between medium and large instances aside, the main conclusion of the experiment is that we are able to scale to such very large scales in terms of both data size and number of nodes, and that this scalability holds in different runs on different cluster sizes, even in the presence of failures.

3.6.5 Comparison with ScaLAPACK

ScaLAPACK is a popular library of high-performance linear algebra routines for distributed memory message passing computers and clusters. ScaLAPACK is an extension of LAPACK [4], and it has been shown to have good scalability and performance. More details about ScaLAPACK can be found in [7]. In this section, we compare our matrix inversion technique to ScaLAPACK, to see how we stack up against a state-of-the-art competitor.

In this experiment, the package libscalapack-mpi-dev in Ubuntu is used. The version of MPI used is MPICH [48]. The drive routines PDGETRF and PDGETRI in ScaLAPACK are used to compute the LU decomposition and the triangular matrix inverse respectively. In order to reduce the data transfer between compute nodes in ScaLAPACK, we use an optimization similar to our *block wrap* optimization. In particular, we set the process grid to $f_1 \times f_2$, where $m_0 = f_1 \times f_2$ is the number of compute nodes, $f_1 \leq f_2$, and there is no factor of m_0 between f_1 and f_2, which means that the matrix is partitioned into $f_1 \times f_2$. The matrix is first partitioned into blocks of dimension 128×128, since we found that this size provides the best performance in our experiment. Next, these blocks are assigned to the process grid. In order to improve load balancing, the blocks are assigned as follows: the block in row $f_1 \times m_1 + i$ and column $f_2 \times m_2 + j$ is assigned to the $(f_2 \times j + i)$-th compute node, where m, n, i and j are integers that are constrained by following inequalities: $f_1 \times m_1 + i < \frac{n}{128}$, $f_2 \times m_2 + j < \frac{n}{128}$, $i \leq f_1$, and $j \leq f_2$, where n is the order of the matrix. In our ScaLAPACK implementation, all intermediate data is stored in memory, such that the matrix is read only once and written only once.

The ratio of the running time of ScaLAPACK to the running time of our algorithm on medium EC2 nodes for matrices $\mathbf{M}_1$ to $\mathbf{M}_3$ is shown in Figure 3.8. This experiment shows that the running time of our algorithm is comparable to ScaLAPACK for these scales. We expect ScaLAPACK to perform well since it is written in Fortran, which has better

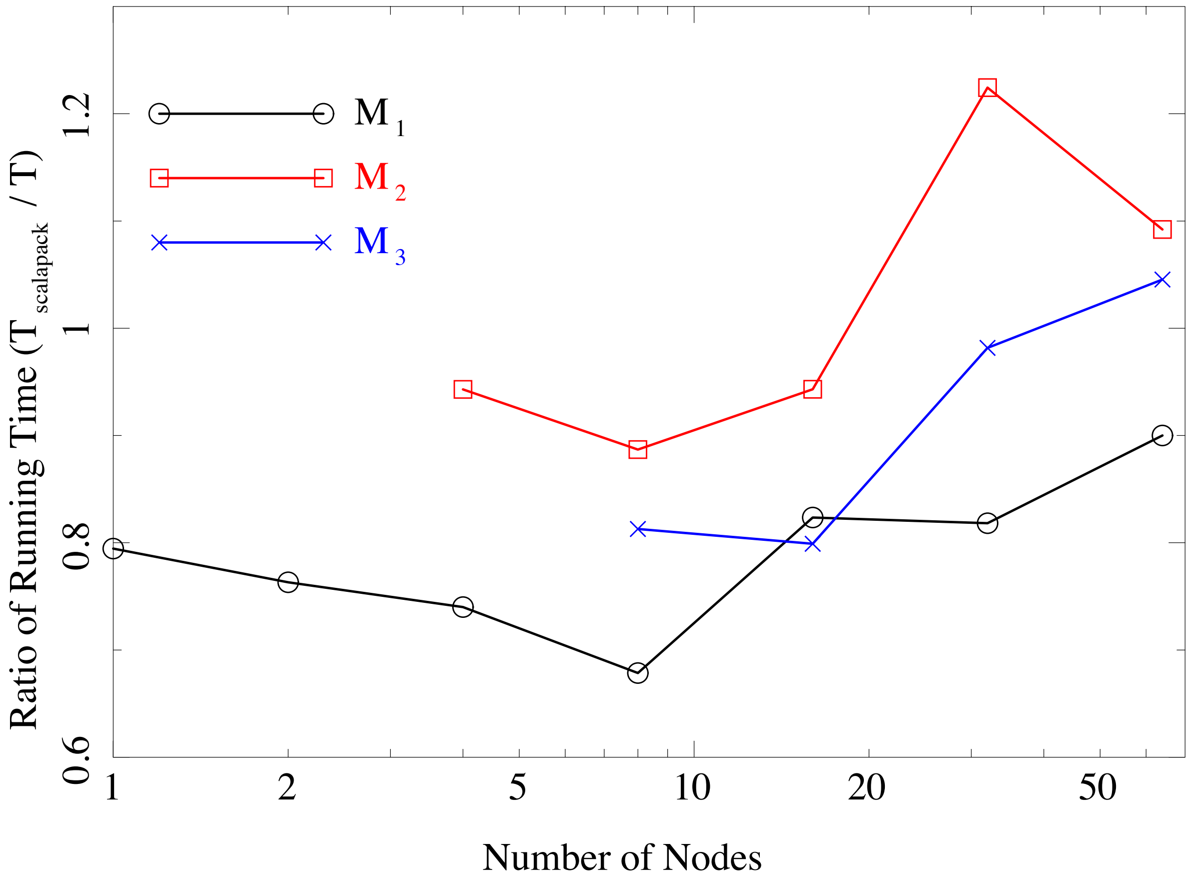


Figure 3.8: The ratio of the running time of ScaLAPACK to the running time of our algorithm.

performance than Java for numerical computation. Therefore, it is a positive result that our algorithm performs comparable to ScaLAPACK for these experiments. The advantage of our algorithm is that it uses the popular and widely deployed MapReduce framework, and it is scalable and fault tolerant.

To demonstrate the scalability of our algorithm as compared to ScaLAPACK, we run another experiment in which we use both 128 large EC2 instances (256 CPU cores) and 64 medium EC2 instances (64 CPU cores) to compute the inverse of the largest matrix, $\mathbf{M}_4$. These are the same cluster sizes that we used with our algorithm for this matrix. ScaLAPACK takes 8 hours on large instances and more than 48 hours on medium instances to invert this matrix, both of which are significantly longer than our results reported in

Section 3.6.4 (5 hours on large instances and 15 hours on medium instances). It should be noted that the running time of our algorithm on the large instances with a failure (approximately 8 hours) is similar to the running time of ScaLAPACK without failure. This experiment shows that our algorithm has better scalability for the large matrix than ScaLAPACK.

3.7 Summary and Future Work

In this chapter, we present a scalable and fault tolerant solution for the dense matrix inversion problem based on the MapReduce framework. We summarize our work as follows:

- We propose a recursive block LU decomposition algorithm that effectively decomposes the matrix into a lower triangular matrix and an upper triangular matrix using MapReduce.
- After calculating the lower triangular and upper triangular matrices, we use the MapReduce framework to compute their inverses and the inverse of original matrix, which is the product of inverses of lower triangular and upper triangular matrices.
- Our experiments show that our approach leads to good scalability, which for matrix inversion. To the best of our knowledge, ours is the first work to provide a scalable matrix inversion solution using MapReduce.
- Our optimizations, namely storing intermediate data in separate files, block wrap, and transpose storing, significantly improve the performance, further demonstrating the scalability of our algorithm. of our algorithm.
- Our algorithm finds the inverse of a large matrix with order 102400 on 128 Amazon EC2 large instances (256 CPU cores) in 5 hours. No other work has reported finding the inverse of such large matrix with comparable performance to our algorithm. It shows that our algorithm can scale to very large matrices.
- Our experiments also show that our algorithm has better scalability than the ScaLAPACK package. In addition, our algorithm is fault tolerant, while ScaLAPACK is not.

In our implementation using Hadoop, all intermediate data, such as $\mathbf{L}_1$ and $\mathbf{U}_1$, is written to HDFS files in each MapReduce job and this intermediate data is read from

HDFS files in the next MapReduce job. Therefore, the read I/O in our algorithm is much larger than other algorithms that persist intermediate data in memory, such as the MPI implementation of ScaLAPACK (see the comparison in the Table 3.1). In Hadoop, it is not easy to persist this data in memory and still keep the algorithm fault tolerant. However, the recently developed Spark system [82] provides parallel data structures that let users explicitly persist data in memory with fault tolerance. Therefore, one direction for future work is to implement our algorithm using Spark, which we expect would improve performance by reducing read I/O. Since Spark is based on MapReduce, the implementation of our algorithm in Spark is similar to the one in Hadoop, except that $\mathbf{L}_1$ and $\mathbf{U}_1$ in Figure 3.5 are loaded from distributed memory instead of being read from HDFS files.

Chapter 4

Finding Eigenvalues and Eigenvectors

Finding the eigenvalues and eigenvectors of a matrix is an important problem in many fields. In this chapter, we briefly motivate the eigenvalue and eigenvector problem. We then propose an algorithm to solve this problem using MapReduce. Unlike the previous two scientific computing problems, we do not implement the algorithms proposed in this chapter. Thus, this chapter can be viewed as an initial investigation of the problem, to be completed as future work.

4.1 Applications of the Eigenvalue and Eigenvector Problem

In physics and engineering, the eigenvalue and eigenvector problem is often the most crucial, or even the 'only' problem. That is because the dynamics of our world are typically governed by some linear differential equation, whose solution is essentially given by the eigenvalues and eigenvectors of some large matrix.

As an example, in classical mechanics, the dynamics of the positions of a system of particles, denoted by $\mathbf{r}(t)$, are usually governed by the coupled equation, which has motion of the form

$$\frac{d}{dt}\mathbf{r}(t) = \mathbf{A}\mathbf{r}(t), \tag{4.1}$$

where $\mathbf{A}$ is a square matrix [68]. The solution to this systems of equations is to diagonalize $\mathbf{A}$ to find all the eigenvalues and eigenvectors. Denote the set of eigenvalues and the corresponding eigenvectors by λ_j, and $\mathbf{r}_j$. In the basis of $\{\mathbf{r}_j\}$ which diagonalizes $\mathbf{A}$, Equation 4.1 is rewritten as, for each j,

$$\frac{d}{dt}\mathbf{r}_j(t) = \mathbf{A}_n \mathbf{r}_j(t), \tag{4.2}$$

which are decoupled harmonic oscillator equations that can be solved independently.

Another example is from quantum physics, where a central goal is to solve the Shrödinger equation

$$i\frac{\partial}{\partial t}\Psi(\mathbf{r},t) = \mathbf{H}(\mathbf{r},t)\Psi(\mathbf{r},t). \tag{4.3}$$

Here, $\Psi(\mathbf{r},t)$, as a function of position $\mathbf{r}$ and time t, is the wave function characterizing the state of the system, and i is the imaginary unit, i.e., $i^2 = -1$. $\mathbf{H}(\mathbf{r},t)$, called the Hamiltonian of the system, is a matrix representing the Hermitian operator in terms of the position $\mathbf{r}$ and time t.

In most practical cases, the Hamiltonian $\mathbf{H}$ is time independent, i.e., independent of t. Hence, the solution to the Shrödinger equation can be given by

$$\Psi(\mathbf{r},t) = e^{-i\mathbf{H}t}\Psi(\mathbf{r},t=0), \tag{4.4}$$

where $\Psi(\mathbf{r},t=0)$ is the initial state of the system, and the operator $e^{-i\mathbf{Ht}}$ is a unitary operator that governs the dynamics of the system.

The form of the operator $e^{-i\mathbf{Ht}}$ is given by diagonalizing $\mathbf{H}$, i.e., solving the 'stationary' Shrödinger equation

$$\mathbf{H}\Psi_j(\mathbf{r}) = \mathbf{E}_j\Psi_j(\mathbf{r}), \tag{4.5}$$

to find the eigenvalues $\mathbf{E}_j$ and the eigenvectors $\Psi_j(\mathbf{r})$ of $\mathbf{H}$.

One can then expand the initial state $\Psi(\mathbf{r},t=0)$ in terms of the complete basis of the eigenvectors $\Psi_j(\mathbf{r})$ of $\mathbf{H}$, i.e.,

$$\Psi(\mathbf{r},t=0) = \sum_j c_j\Psi_j(\mathbf{r}), \tag{4.6}$$

then the solution to the Shrödinger equation is found, which is given by

$$\Psi(\mathbf{r},t) = \sum_j e^{-iE_jt}\Psi_j(\mathbf{r}). \tag{4.7}$$

In these two examples, the key step toward the solution to the equation of motion is to diagonalize some large square matrix. In the quantum mechanics case, the matrix is of size $2^n \times 2^n$, where n is the number of particles in the system. In many cases, the most important thing one needs to know is the smallest eigenvalue and the corresponding eigenvector, which are called the ground state energy and ground state wave function of the systems, respectively. In those systems where each particle has interaction with most other particles, the matrix $\mathbf{H}$ is a dense matrix.

Many other applications of the eigenvalues and eigenvector problems exist, and we now turn our attention to solving this problem on MapReduce.

4.2 Finding Eigenvalues and Eigenvectors on MapRedcue

The eigenvalue and eigenvector problem can be written as

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \tag{4.8}$$

where $\mathbf{A}$ is a square matrix of order n, and $\mathbf{v}$ is a vector of dimension n. λ is the eigenvalue of $\mathbf{A}$ associated with the eigenvector $\mathbf{v}$.

There are many methods to solve the eigenvalue and eigenvector problems. They, however, are designed to run on a single core. When the order of the matrix $\mathbf{A}$ becomes very large, we need to extend these algorithms so that they can efficiently run on a large cluster. MapReduce can give us the required scalability and fault tolerance to run on such clusters.

In this chapter, we do not introduce all possible algorithms that are suitable for parallel computing. Instead, we briefly present one algorithm, named *inverse iteration* [61], and propose an approach to use it to solve the eigenvalue and eigenvector problem for large dense matrices on MapReduce.

The inverse iteration method first assumes that there is an approximate eigenvalue μ, and an approximate eigenvector $\mathbf{v}_0$. The method then uses an iteration step to get an increasingly accurate eigenvector. The iteration step is given by

$$\mathbf{v}_{k+1} = \frac{(\mathbf{A} - \mu\mathbf{I}_n)^{-1}\mathbf{v}_k}{||(\mathbf{A} - \mu\mathbf{I}_n)^{-1}\mathbf{v}_k||} \tag{4.9}$$

where $\mathbf{I}_n$ is the identity matrix of order n. The eigenvalue corresponding to this eigenvector is

$$\lambda = \frac{\mathbf{v}^*\mathbf{A}\mathbf{v}}{\mathbf{v}^*\mathbf{v}} \tag{4.10}$$

Algorithm 8 Bisection binary search algorithm to find all eigenvalues.

```
1:  function BinarySearch($\mathbf{A}$, $\lambda_1$, $\lambda_2$)
2:  $\lambda = (\lambda_1 + \lambda_2)/2$
3:  $\lambda_0$ = getEigenvalue($\mathbf{A}$, $\lambda$)
4:  if $\lambda_0 <= \lambda_1$ OR $\lambda_0 >= \lambda_2$ then
5:      return
6:  end if
7:  Add $\lambda_0$ into result set. /* Get new eigenvalue */
8:  if $\lambda_0 <= \lambda$ then
9:      BinarySearch($\mathbf{A}$, $\lambda_1$, $\lambda_0$)
10:     BinarySearch($\mathbf{A}$, $\lambda$, $(\lambda + \lambda_1)$ / 2)
11:     BinarySearch($\mathbf{A}$, $(\lambda + \lambda_1)$ / 2, $\lambda_1$)
12: else
13:     BinarySearch($\mathbf{A}$, $\lambda_1$, $(\lambda_1 + \lambda)$ / 2)
14:     BinarySearch($\mathbf{A}$, $(\lambda_1 + \lambda)$ / 2, $\lambda$)
15:     BinarySearch($\mathbf{A}$, $\lambda_0$, $\lambda_1$)
16: end if
17: return
```

where $\mathbf{v}^*$ is the conjugate transpose of $\mathbf{v}$. The conjugate transpose of $\mathbf{v}$ is defined as $[\mathbf{v}^*]_{ij} = \overline{[\mathbf{v}]_{ji}}$, where the overbar is a scalar complex conjugate, i.e., $\overline{a+bi} = a - bi$, where a and b are both real. If we set $\mu = 0$, then this iteration method would get the ground (smallest) eigenvalue and the ground eigenvector, while with $\mu = \infty$, it gets the largest eigenvalue and the corresponding eigenvector.

The complexity of this method is $O(n^3) + kO(n^2)$, i.e., the most time consuming part is computing $(\mathbf{A} - \mu\mathbf{I}_n)^{-1}$. In general, k is much less than n. Therefore if we can efficiently compute $(\mathbf{A} - \mu\mathbf{I}_n)^{-1}$, we can efficiently solve the eigenvalue and eigenvector problem using the inverse iteration method.

We propose that we first compute the inverse of matrix $(\mathbf{A} - \mu\mathbf{I}_n)$ using MapReduce as shown in Chapter 3, and then compute the eigenvector $\mathbf{v}$ on one CPU. Finally, we can derive the eigenvalue λ on one CPU. Since we have shown that the matrix inversion algorithm in Chapter 3 is scalable and fault tolerant, the proposed inverse iteration algorithm to determine the eigenvalue and eigenvector is expected to be scalable and fault tolerant as well.

In order to calculate other eigenvalues and eigenvectors of the matrix $\mathbf{A}$, we need to choose an appropriate starting eigenvalue μ. For this, we can use the bisection binary search method. Assuming that we have found the smallest (ground) eigenvalue λ_0 and the largest eigenvalue λ_{n-1}, then we can set the next initial eigenvalue to $(\lambda_0 + \lambda_m)/2$ and we get another eigenvalue λ_k. After that we can use this new eigenvalue to find the next eigenvalue in other intervals, and so on. The pseudocode of the bisection binary search method is given in Algorithm 8, where the function getEigenvalue($\mathbf{A}$, λ) is the MapReduce based algorithm using Equations 4.9 and 4.10.

Using this approach, all eigenvalues and eigenvectors of a large matrix $\mathbf{A}$ can be efficiently obtained using MapReduce. The detailed implementation and evaluation of this approach are left to future work.

Chapter 5

Conclusion

In this thesis, we presented algorithms to solve two important scientific computing problems using MapReduce, the maximum clique problem and the matrix inversion problem. Our results illustrate that we can use cloud computing, in particular MapReduce, to solve some scientific computing problems with good scalability and fault tolerance. We also outline an algorithm to solve the eigenvalue and eigenvector problem using MapReduce. Next, we summarize the main contributions of the thesis.

We presented a scalable and fault tolerant maximum clique algorithm based on MapReduce. The main contribution underlying our algorithm is the BMC partitioning method, which enables us to partition a graph into many subgraphs in a way that maintains load balancing. Good load balancing cannot be achieved with alternative partitioning algorithms that we investigated such as one-depth partitioning. We also investigated the relationship between the total running time of our maximum clique algorithm and the graph size and density, and provided a cost model to accurately estimate this running time, which is required by our algorithm.

Our experiments show that our approach outperforms other approaches on real and synthetic graphs, and is robust to tuning parameter settings. Some real maximum clique problems have been solved using our scalable and fault tolerant algorithm with orders of magnitude improvement in performance.

We also presented a scalable and fault tolerant algorithm for matrix inversion based on the MapReduce framework. Our main contribution is to design a parallel algorithm suitable for MapReduce to LU decompose the matrix, i.e., to compute the L and U matrices that are used to invert the matrix. The algorithm uses a pipeline of MapReduce jobs, and depends on a recursive block partitioning of the matrix to optimize data flow through this pipeline of jobs. Our experiments show that our approach leads to good scalability, which is a first using the MapReduce framework. Our method is simpler and more fault tolerant than MPI implementations such as ScaLAPACK, as well as solutions that require special hardware such as GPUs.

We also propose an algorithm to find the eigenvalues and eigenvectors of a large dense matrix based on our scalable and fault tolerant matrix inversion algorithm. Implementing and evaluating this algorithm is an interesting direction for future work.

References

[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov. LU factorization for accelerator-based systems. In *Proc. ACS/IEEE Int. Conf. on Comp. Systems and Applications (AICCSA)*, 2011.

[2] L. Akoglu, H. Tong, B. Meeder, and C. Faloutsos. PICS: Parameter-free identification of cohesive subgroups in large attributed graphs. In *Proc. SIAM Data Mining Conf. (SDM)*, 2012.

[3] N. Alon, M. Krivelevich, and B. Sudakov. Finding a large hidden clique in a random graph. In *Proc. Symp. on Discrete Algorithms (SODA)*, 1998.

[4] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In *Proc. Conf. on Supercomputing (SC)*, 1990.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Comm. ACM*, 53(4):50–58, 2010.

[6] P. Bientinesi, B. Gunter, and Geijn, Robert A. van de. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Mathematics Software*, 35(1):3:1–3:22, 2008.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics, 1997.

[8] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow. (PVLDB)*, 3(1-2):285–296, 2010.

[9] A. Calderbank, E. Rains, P. Shor, and N. J. A. Sloane. Quantum error correction via codes over GF(4). *IEEE Trans. on Information Theory*, 44(4):1369–1387, 1998.

[10] R. Carraghan and P. M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9(6):375 – 382, 1990.

[11] I. Chuang, A. Cross, G. Smith, J. Smolin, and B. Zeng. Codeword stabilized quantum codes: Algorithm and structure. *J. Mathematical Physics*, 50(4):042109 – 042125, 2009.

[12] A. W. Cross, G. Smith, J. A. Smolin, and B. Zeng. Codeword stabilized quantum codes. *IEEE Trans. on Information Theory*, 55(1):433 – 438, 2009.

[13] L. Csanky. Fast parallel matrix inversion algorithms. In *Proc. Annual Symp. on Foundations of Comp. Science (FOCS)*, 1975.

[14] K. Dackland, E. Elmroth, B. Kågström, and C. V. Loan. Design and evaluation of parallel block algorithms: LU factorization on an IBM 3090 VF/600J. In *Proc. SIAM Conf. on Parallel Proc. for Scientific Computing*, 1991.

[15] L. E. Danielsen. Database of self-dual quantum codes. http://www.ii.uib.no/ larsed/vncorbits/.

[16] L. E. Danielsen. On Self-Dual Quantum Codes, Graphs, and Boolean Functions. *eprint arXiv:quant-ph/0503236*, 2005.

[17] B. De Schutter and B. De Moor. The QR decomposition and the singular value decomposition in the symmetrized max-plus algebra. In *Proc. European Control Conf. (ECC)*, 1997.

[18] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. Conf. on Symp. on Operating Systems Design and Implementation (OSDI)*, 2004.

[19] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Comm. ACM*, 51(1):107–113, 2008.

[20] DIMACS Benchmark Set. http://iridia.ulb.ac.be/∼fmascia/maximum_clique/.

[21] DIMACS Implementation Challenges. http://dimacs.rutgers.edu/Challenges/.

[22] X. Dimitropoulos, D. Krioukov, M. Fomenkov, B. Huffaker, Y. Hyun, K. Claffy, and G. Riley. AS relationships: inference and validation. *SIGCOMM Comp. Comm. Rev.*, 37(1):33 – 40, 2007.

[23] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. High performance matrix inversion based on LU factorization for multicore architectures. In *Proc. ACM Intl. Workshop on Many Task Computing on Grids and Supercomputers (MTAGS)*, 2011.

[24] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:2003 – 2016, 2003.

[25] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[26] M. Falchi and C. Fuchsberger. Jenti: an efficient tool for mining complex inbred genealogies. *Bioinformatics*, 24(5):724–726, 2008.

[27] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proc. European PVM/MPI Users' Group Meeting*, 2004.

[28] L. Gao. On inferring autonomous system relationships in the Internet. *IEEE/ACM Trans. Network*, 9(6):734–745, 2001.

[29] E. Gardiner, P. Artymiuk, and P. Willett. Clique-detection algorithms for matching three-dimensional molecular structures. *J. Molecular Graphics and Modelling*, 15(4):245 – 253, 1997.

[30] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.

[31] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2011.

[32] Giraph. http://giraph.apache.org/.

[33] G. H. Golub and C. F. Van Loan. *Matrix computations.* Johns Hopkins University Press, third edition, 1996.

[34] D. Gottesman. *Stabilizer Codes and Quantum Error Correction.* PhD thesis, California Institute of Technology, 1997.

[35] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[36] L. W. Hagen and A. B. Kahng. New spectral methods for ratio cut partitioning and clustering. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11(9):1074–1085, 1992.

[37] Hadoop Distributed File System (HDFS). http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html.

[38] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Conf. on Supercomputing (SC)*, 1995.

[39] C. W. Huffman and V. Pless. *Fundamentals of Error-Correcting Codes.* Cambridge University Press, 2003.

[40] E. Kaltofen and V. Pan. Processor efficient parallel solution of linear systems over an abstract field. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, 1991.

[41] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical J.*, 49:291–307, 1970.

[42] A. Kloczkowski, R. Jernigan, Z. Wu, G. Song, L. Yang, A. Kolinski, and P. Pokarowski. Distance matrix-based approach to protein structure prediction. *J. Structural and Functional Genomics*, 10(1):67–81, 2009.

[43] S. Krebs. Dense networks spread bad behavior in baseball. http://orgnet.com/steroids.html.

[44] K. K. Lau, M. Kumar, and S. Venkatesh. Parallel matrix inversion techniques. In *Proc. IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing*, 1996.

[45] P. Lin, Z. Wang, and M. Guo. A maximum clique algorithm based on MapReduce. In *Proc. Int. Conf. on Advanced Comp. Theory and Engineering*, 2010.

[46] R. D. Luce and A. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14:95–116, 1949.

[47] S. Lupke. LU-decomposition on a massively parallel transputer system. In *Proc. Int. Parle Conf. on Parallel Architectures and Languages*, 1993.

[48] E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[49] F. MacWilliams and N. Sloane. *The Theory of Error-Correcting Codes.* North-Holland, second edition, 1978.

[50] H. V. Madhyastha, T. Anderson, A. Krishnamurthy, N. Spring, and A. Venkataramani. A structural approach to latency prediction. In *Proc. ACM SIGCOMM Conf. on Internet Measurement*, 2006.

[51] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2010.

[52] D. Marks, L. Colwell, R. Sheridan, T. Hopf, A. Pagnani, R. Zecchina, and C. Sander. Protein 3d structure computed from evolutionary sequence variation. *PLoS One*, 6(12):e28766–e28766, 2011.

[53] M. Nielsen and I. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2000.

[54] J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer, 2000.

[55] P. Östergård, T. Baicheva, and E. Kolev. Optimal binary one-error-correcting codes of length 10 have 72 codewords. *IEEE Trans. on Information Theory*, 45(4):1229–1231, 1999.

[56] P. R. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1–3), 2002.

[57] P. R. J. Östergård. Cliquer. http://users.tkk.fi/pat/cliquer.html.

[58] P. M. Pardalos, J. Rappe, and M. G. Resende. An exact parallel algorithm for the maximum clique problem. *High Performance Algorithms and Software in Nonlinear Optimization*, 38:279–300, 1998.

[59] P. M. Pardalos and J. Xue. The maximum clique problem. *J. Global Optimization*, 4(3):301–328, 1994.

[60] S. P. Parker. *MacGraw-Hill Encyclopedia of Physics.* McGraw-Hill Ryerson, second edition, 1993.

[61] B. N. Parlett. *The symmetric eigenvalue problem.* Prentice-Hall, Inc., 1998.

[62] M. C. Pease. Matrix inversion using parallel processing. *J. ACM*, 14(4):757–764, 1967.

[63] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, third edition, 2007.

[64] M. Sellmann, N. Sensen, and L. Timajev. Multicommodity flow approximation used for exact graph partitioning. In *Proc. Annual European Symp. on Algorithms (ESA)*, 2003.

[65] N. Sensen. Lower bounds and exact algorithms for the graph partitioning problem using multicommodity flows. In *Proc. Annual European Symp. on Algorithms (ESA)*, 2001.

[66] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng. HPMR: Prefetching and pre-shuffling in shared mapreduce computation environment. In *Proc. IEEE Int. Conf. on Cluster Computing and Workshops*, 2009.

[67] M. Sipser. *Introduction to the Theory of Computation.* Course Technology, third edition, 2012.

[68] J. Taylor. *Classical Mechanics.* University Science Books, 2005.

[69] E. Tomita and T. Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *J. Global Optimization*, 37(1):95–111, 2007.

[70] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *WALCOM: Algorithms and Computation.* Springer Berlin Heidelberg, 2010.

[71] L. N. Trefethen and D. Bau. *Numerical Linear Algebra.* Society for Industrial and Applied Mathematics, 1997.

[72] L. G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, 1990.

[73] M. Vingron and P. Pevzner. Motif recognition and alignment for many sequences by comparison of dot matrices. *J. Molecular Biology*, 218(1):33–43, 1991.

[74] M. Wall, A. Rechtsteiner, and L. Rocha. Singular value decomposition and principal component analysis. In *A Practical Approach to Microarray Data Analysis*. Springer US, 2003.

[75] J. Wang and J. Cheng. Truss decomposition in massive networks. *Proc. VLDB Endow. (PVLDB)*, 5(9):812–823, 2012.

[76] R. J. Warp, D. J. Godfrey, and J. T. Dobbins III. Applications of matrix inversion tomosynthesis. *Physics of Medical Imaging*, 3977:376–383, 2000.

[77] B. Wu, S. Yang, H. Zhao, and B. Wang. A distributed algorithm to enumerate all maximal cliques in MapReduce. In *Proc. Int. Conf. on Frontiers of Comp. Science and Technology*, 2009.

[78] J. Xiang, C. Guo, and A. Aboulnaga. Scalable maximum clique computation using MapReduce. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2013.

[79] J. Xiang, S. N. Zhang, and Y. Yao. Probing the spatial distribution of the interstellar dust medium by high angular resolution X-ray halos of point sources. *The Astrophysical J.*, 628:769–779, 2005.

[80] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2012.

[81] Y. Yao, S. N. Zhang, X. Zhang, and Y. Feng. A new method to resolve X-ray halos around point sources with Chandra data and its application to Cygnus X-1. *The Astrophysical J. Letters*, 594:L43–L46, 2003.

[82] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX Conf. on Networked Systems Design and Implementation (NSDI)*, 2012.

[83] Y. Zhang and J. Yang. Optimizing I/O for big array analytics. *Proc. VLDB Endow. (PVLDB)*, 5(8):764–775, 2012.

[84] Q. Zheng, X. Wu, M. Fang, H. Wang, S. Wang, and X. Wang. Application of HPMR in parallel matrix computation. *Comp. Engineering*, 36(08):49–52, 2010.