MAPREDUCE IMPLEMENTATION OF STRASSEN’S ALGORITHM FOR MATRIX MULTIPLICATION

A Thesis by

Minhao Deng

Bachelor of Management, Jiangxi University of Finance and Economics, 2011

Submitted to the Department of Electrical Engineering and Computer Science and the faculty of the Graduate School of Wichita State University in partial fulfillment of the requirements for the degree of Master of Science

May 2017
MAPREDUCE IMPLEMENTATION OF STRASSEN'S ALGORITHM FOR MATRIX MULTIPLICATION

The following faculty members have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science with a major in Computer Science.

Prakash Ramanan, Committee Chair

Kaushik Sinha, Committee Member

Krishna Krishnan, Committee Member
DEDICATION

To my parents, and my dear friends.
Get wisdom . . . though it costs all your possessions,
      get understanding.
I would like to thank my advisor, Prof. Ramanan. He gave me generous help during the hardest time I was going through. Without him, I could have never kept studying at a high level. Under his direction, I learnt a lot about distributed algorithms, and during the period, I finally found something I was interested in. He was more than an instructor, also the very best friend, and a family.
ABSTRACT

Consider the multiplication $C = A \times B$ of two $n \times n$ matrices. A straight-forward sequential algorithm for computing the product takes $\Theta(n^3)$ time. Strassen [17] presented an algorithm that takes $\Theta(n^{\lg 7})$ time; $\lg$ denotes logarithm to the base 2; $\lg 7$ is about 2.81.

Now, consider the implementation of these two algorithms (straight-forward and Strassen) in the mapReduce framework. Several papers have studied mapReduce implementations of the straight-forward algorithm; this algorithm can be implemented using a constant number (typically, one or two) mapReduce passes. In this thesis, we study the mapReduce implementation of Strassen’s algorithm.

If we unwind the recursion, Strassen’s algorithm consists of three parts, Parts I-III. Direct mapReduce implementations of the three parts take $\lg n$, 1 and $\lg n$ passes, for a total of $2 \lg n + 1$ passes. We show that Part I can be implemented in 2 passes, with total work $\Theta(n^{\lg 7})$, and reducer size and reducer workload $o(n)$. We believe that Part III can also be implemented, with small reducer size and workload, in a constant number of passes (possibly 4 passes); this is future work.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2.1</td>
</tr>
<tr>
<td>2.1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2.2</td>
</tr>
<tr>
<td>2.2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>6.1</td>
</tr>
<tr>
<td>6.1</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>6.2</td>
</tr>
<tr>
<td>6.2</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>7.1</td>
</tr>
<tr>
<td>7.1</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>7.2</td>
</tr>
<tr>
<td>7.2</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>REFERENCES</td>
</tr>
</tbody>
</table>

viii
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Performance Measures</td>
<td>5</td>
</tr>
<tr>
<td>4.1 Part II: $S_i(A)$ is paired with $S_{T[i]}(B)$</td>
<td>9</td>
</tr>
<tr>
<td>4.2 Example 4.2: For $n = 2^2$: $f_A = S_{i_1,i_2}(A) = S_{i_2}(S_{i_1}(A))$</td>
<td>10</td>
</tr>
<tr>
<td>4.3 Notations from Chapter 4</td>
<td>11</td>
</tr>
<tr>
<td>6.1 Notations from Chapter 6</td>
<td>17</td>
</tr>
<tr>
<td>6.2 Example 6.3: For $n = 2^2$: $(i_1, i_2)$th entry gives the NFFVs $S_{i_1,i_2}(A)$ computed by reducer $r_{i_1,i_2}$</td>
<td>18</td>
</tr>
<tr>
<td>6.3 Mapper for $E1$: $j$th component $i_j$ of reducer signature $(i_1, \ldots, i_j, \ldots, i_k)$ based on $((x - 1)_j, (y - 1)_j)$</td>
<td>19</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Dividing a matrix into submatrices</td>
<td>6</td>
</tr>
<tr>
<td>4.1</td>
<td>Unwind the recursion</td>
<td>8</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Consider the multiplication $C = A \times B$ of two $n \times n$ matrices. A straight-forward sequential algorithm for computing the product takes $\Theta(n^3)$ time. For large values of $n$, this is very slow. So, several papers have studied the problem of multiplying matrices using a large number of processors (CPUs) in parallel.

There are many parallel computation models studied in the literature, such as the shared memory model [9] and the distributed memory model; they involve special-purpose architectures, and are expensive. In contrast, the mapReduce framework (introduced by Google [4, 12]) consists of a large number of commodity compute nodes that are connected together by Ethernet or switches. Apache’s Hadoop [16, 19] is an open source implementation of this framework.

The straight-forward matrix multiplication algorithm can be implemented on mapReduce using one or two passes. For these implementations, [1, 15] presented tight bounds on the tradeoff between communication cost and parallelism. [14] presented upper and lower bounds on the number of mapReduce passes required as a function of memory sizes of individual computing nodes, and the total memory in the computing cluster.

Algorithms for matrix multiplication have been studied in other parallel computation models. In the distributed memory model, [8] presented a tradeoff between communication among the processors, and memory size of the individual processors. [6] studied communication-parallelism tradeoff in pipelined parallelism. [11] presented a parallel algorithm that uses shared memory and remote memory access communication.

In contrast to the $\Theta(n^3)$ time algorithm mentioned above, Strassen [17, 3] presented a sequential algorithm that runs in $\Theta(n^{\log_27})$ time. Our thesis deals with implementing Strassen’s algorithm on mapReduce. Strassen’s algorithm is recursive. If we unwind the recursion, the algorithm consists of three parts, Parts I-III. We do the following:
• We first present iterative versions of the three parts.

• We then present a straight-forward mapReduce implementation of the three parts. The three parts take $\lg n$, 1 and $\lg n$ mapReduce passes, for a total of $2\lg n + 1$ passes.

• We consider some one-pass algorithms for Part I; their performance measures are not good.

• We consider some two-pass algorithms for Part I. We show that one such algorithm has total work $\Theta(n^{\lg 7})$, and reducer size and reducer workload $o(n)$.

We believe that Part III can also be implemented in a constant number of passes (possibly 4 passes), with good performance measures; this is future work.

The outline of our thesis is as follows. In Chapter 2, we describe the mapReduce framework, and its complexity factors. In Chapter 3, we describe Strassen’s algorithm. In Chapter 4, we present an iterative version of this algorithm. In Chapter 5, we describe a straight-forward mapReduce implementation. In Chapter 6, we present our one-pass algorithms for Part I. In Chapter 7, we present our two-pass algorithms for Part I. Chapter 8 concludes.
CHAPTER 2
The MapReduce Framework

In this chapter, we first present an overview of the mapReduce framework [13], and then introduce its complexity factors. This chapter is taken from [15].

2.1 Overview

In the mapReduce framework, large number of compute nodes are connected together by Ethernet cables or switches. Compute nodes operate on files of enormous size, using a distributed file system [5]. Files are divided into chunks and these chunks are processed in parallel.

A mapReduce computation consists of two phases: map phase and reduce phase. In between these two phases, an important role is played by the master controller.

Map Phase: The map phase consists of several independent map tasks. Each map task gets as input a chunk of the input file. All the map tasks execute the same map function. For each input element, the map function outputs zero or more (key, value) pairs. The keys of the generated (key, value) pairs need not be unique. The code for the map function is written by the user, tailored to the specific problem to be solved.

Master Controller: The master controller collects the (key, value) pairs output by all the map tasks. It then groups them and, for each distinct key, generates a pair consisting of the key along with a list of all the values associated with that key.

Reduce Phase: The reduce phase consists of several independent reducers. Each reducer receives a (key, list of values) pair from the master controller. All the reducers execute the same reduce function. The function combines all the values associated with that key in some specific way. The code for combining the values is written by the user, tailored to the specific problem to be solved. The output of all the reducers is the output (i.e., result) for the entire algorithm.
A mapReduce computation could consist of several passes. In this case, the output of the reduce phase in one pass is input to the map phase of the next pass. The output of the reduce phase in the last pass is the output for the entire algorithm.

2.2 Complexity Factors

Communication cost and the degree of parallelism are the two most important factors for a mapReduce algorithm. Among the following three measures, the first measures communication cost; the latter two measure the degree of parallelism.

Communication Cost $CC$ [1]: Communication cost $CC$ is the cost of sending data from the map phase to the reduce phase.

$$CC = \sum_{\text{map-tasks}} (\text{Output size of the map-task}) = \sum_{\text{reduce-tasks}} (\text{Input size of the reduce-task}).$$

Reducer Size $q$ [1]: This is one measure of the degree of parallelism. It is the largest size of the input (i.e., list of values) sent to any one reducer. It measures the largest amount of memory any reducer needs. Smaller the reducer size, higher is the degree of parallelism.

Reducer Workload $w$ [15]: This is another measure of the degree of parallelism. It is the largest amount of work done by any reducer. It measures the largest amount of time any reducer needs. Smaller the reducer workload, higher is the degree of parallelism. $W$ denotes the total work of all the reducers.

Having more reducers increases the communication cost, because the same input item has to be sent to more reducers. But having more reducers decreases the reducer size and workload. If the reducer size is small, then the reduce function can be executed in main memory. So, having a smaller reducer size or workload lowers the wall-clock time for each reducer; this increases parallelism.
Table 2.1: Performance Measures

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>number of reducers</td>
</tr>
<tr>
<td>$CC$</td>
<td>communication cost</td>
</tr>
<tr>
<td></td>
<td>$= \sum_{M\text{-tasks}}$ (Output size of the M-task)</td>
</tr>
<tr>
<td></td>
<td>$= \sum_{R\text{-tasks}}$ (Input size of the R-task)</td>
</tr>
<tr>
<td>$q$</td>
<td>reducer size = size of the largest reducer</td>
</tr>
<tr>
<td>$w$</td>
<td>reducer workload = most work of any reducer</td>
</tr>
<tr>
<td>$W$</td>
<td>Total work of all the reducers</td>
</tr>
</tbody>
</table>

For multi-pass algorithms, we consider $CC$, $q$ and $w$ for each pass.

\[
CC = \sum_{\text{reduce\_tasks}} (Input \ size \ of \ the \ reduce\_task). \tag{2.2}
\]

We ignore the communication between the reduce phase of one pass, and the map phase of the next pass. This is because the map tasks of a pass could be executed on the same compute nodes as the reducers of the previous pass.
CHAPTER 3

Strassen’s Algorithm

From now onwards, let $n$ be a power of $2$; $n = 2^k$, $k = \lg n$. Let matrices $A$, $B$ and $C$ be divided into 4 submatrices each, of size $(n/2) \times (n/2)$ (see Figure 3.1). In the straightforward matrix multiplication algorithm, the result matrix $C$ is computed as follows:

\[
\begin{align*}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{align*}
\] (3.1)

So, to compute the product of two $n \times n$ matrices, we need to compute 8 products of $(n/2) \times (n/2)$ matrices. The runtime of this algorithm is given by the recurrence $T(n) = 8T(n/2) + \Theta(n^2)$; its solution is $T(n) = \Theta(n^3)$.

We will see many recurrence relations in this thesis. We use the following well-known theorem [3] to solve them.

**Master Theorem.** Let $T(n) = aT(n/b) + \Theta(f(n))$ for some constants $a \geq 1$, $b > 1$. Compare $f(n)$ to $n^{\log_b a}$.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.

![Figure 3.1: Dividing a matrix into submatrices](image)
3. If \( f(n) = \Omega(n^{\log_b a + \epsilon}) \) for some constant \( \epsilon > 0 \), and \( af(n/b) \leq cf(n) \) for some constant \( c < 1 \), for all large \( n \), then \( T(n) = \Theta(f(n)) \). \( \diamond \)

Strassen’s algorithm \([3, 17]\) computes \( C \) by computing only 7 (instead of 8) products of \((n/2) \times (n/2)\) matrices. It first computes the following 7 matrices (each of dimension \( n/2 \)) from \( A \):

\[
\begin{align*}
S_1(A) &= A_{11} & S_2(A) &= A_{22} \\
S_3(A) &= A_{11} + A_{12} & S_4(A) &= A_{21} + A_{22} \\
S_5(A) &= A_{11} + A_{22} & S_6(A) &= A_{11} - A_{21} & S_7(A) &= A_{12} - A_{22}
\end{align*}
\]

\( S_1(B), \ldots, S_7(B) \) are computed from \( B \) analogously. Then, the algorithm computes the following 7 products recursively.

\[
\begin{align*}
P_1 &= S_1(A) \times S_7(B) & P_2 &= S_2(A) \times S_6(B) \\
P_3 &= S_3(A) \times S_2(B) & P_4 &= S_4(A) \times S_1(B) \\
P_5 &= S_5(A) \times S_5(B) & P_6 &= S_6(A) \times S_3(B) \\
P_7 &= S_7(A) \times S_4(B)
\end{align*}
\]

Then, the product \( C = A \times B \) is computed as:

\[
\begin{align*}
C_{11} &= P_5 + P_7 - P_2 - P_3 & C_{12} &= P_1 + P_3 \\
C_{21} &= P_4 - P_2 & C_{22} &= P_1 + P_5 - P_4 - P_6
\end{align*}
\]

The runtime of this algorithm is given by the recurrence \( T(n) = 7T(n/2) + \Theta(n^2) \); its solution is \( T(n) = \Theta(n^{\log_2 7}) \).

For future reference, for \( 1 \leq i \leq 7 \), let \( P_i(A, B) \) denote the product matrix \( P_i \) above. For \( 1 \leq i, j \leq 2 \), let \( C_{ij}(A, B) \) denote the matrix \( C_{ij} \) above.
[7] presented an iterative version of Strassen’s algorithm using tensor products. In this chapter, we present a simpler iterative version. First, we have the following:

**Observation 4.1.** Each of the products $P_i(A,B)$ above is of the form $S_i(A) \times S_j(B)$. $S_i(A)$ depends only on $A$, and $S_j(B)$ depends only on $B$. So, if we unwind the recursion in Strassen’s algorithm, the algorithm would consist of 3 parts:

**Part I.** First, compute $7^{\log_7 n} = n^{\log_7 7}$ real numbers from $A$:

$$S_{i_1,i_2,\ldots,i_k}(A) = S_{i_k}(S_{i_{k-1}}(\ldots(S_{i_1}(A))))), \ 1 \leq i_1, \ldots, i_k \leq 7.$$  \hspace{1cm} (4.1)

We call these real numbers final values (FVs); $fv(A)$ denotes the list of $n^{\log_7 7}$ FVs, each of which has a $k$ dimensional index of $(i_1,i_2,\ldots,i_k)$. Similarly, compute $fv(B)$.

In Figure 4.1, an example shows how a $2^3 \times 2^3$ matrix will produce $(2^3)^{\log_7 7} = 7^3$ final values. By unwinding the recursion, we see the middle submatrices needed to compute the
Table 4.1: Part II: $S_i(A)$ is paired with $S_{T[i]}(B)$

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

final values. The specific final value marked in the figure is computed through a submatrix chain of #2, #7, and #4, thus it has an index of (2,7,4). We’ll give two more detailed examples later in this chapter.

**Part II.** Pair each value in $fv(A)$ with the corresponding value in $fv(B)$. The correspondence is established using the products $P_i$ in Chapter 3. For instance, considering $P_1(A,B) = S_1(A) \times S_7(B)$, the $n^{\lg 7}/7$ values in $fv(A)$ that are in $fv(S_1(A))$ are paired with the $n^{\lg 7}/7$ values in $fv(B)$ that are in $fv(S_7(B))$; this latter pairing is done recursively.

Table 4.1 gives the pairing in general: $S_{i_1,i_2,...,i_k}(A)$ is paired with $S_{T[i_1],T[i_2],...,T[i_k]}(B)$. Then, compute the product for each such pair. Now, we have $n^{\lg 7}$ real numbers.

**Part III.** Group the $n^{\lg 7}$ real numbers into $n^{\lg 7}/7$ groups of 7 numbers each: For each $(i_1, i_2, ..., i_{k-1})$, $1 \leq i_1, ..., i_{k-1} \leq 7$, there is a group consisting of the 7 real numbers

$$S_{i_1,i_2,...,i_k}(A) \times S_{T[i_1],T[i_2],...,T[i_k]}(B), 1 \leq i_k \leq 7.$$  \hspace{1cm} (4.2)

Each group constitutes a group of $P_1, \ldots, P_7$; compute the 4 reals ($C_{11}, C_{12}, C_{21}, C_{22}$) from them (see Chapter 3). Now, the 7 real numbers are replaced by a $2 \times 2$ matrix. So, we have $n^{\lg 7}/7$ such matrices.

Group them into $n^{\lg 7}/7^2$ groups, one for each $(i_1, i_2, ..., i_{k-2})$. Each group (of 7 $2 \times 2$ matrices) constitutes a group of $P_1, \ldots, P_7$; compute the 4 values ($C_{11}, C_{12}, C_{21}, C_{22}$) from them; each $C_{ij}$ is a $2 \times 2$ matrix. Now, the $7 \times 2 \times 2$ matrices in each group are replaced by
Table 4.2: Example 4.2: For \( n = 2^2 \): \( f(v)(A) = S_{i_1 i_2}(A) = S_{i_2}(S_{i_1}(A)) \)

| \( i_1 | i_2 \) | 1     | 2     | 3     | 4     | 5     | 6     | 7     |
|----------------|-------|-------|-------|-------|-------|-------|-------|
| 1              | \( a_{11} \) | \( a_{22} \) | \( a_{11} + a_{12} \) | \( a_{21} + a_{22} \) | \( a_{11} + a_{22} \) | \( a_{11} - a_{21} \) | \( a_{12} - a_{22} \) |
| 2              | \( a_{33} \) | \( a_{44} \) | \( a_{33} + a_{34} \) | \( a_{43} + a_{44} \) | \( a_{33} + a_{44} \) | \( a_{33} - a_{43} \) | \( a_{34} - a_{44} \) |
| 3              | \( a_{11} + a_{13} \) | \( a_{22} + a_{24} \) | \( a_{11} + a_{12} \) | \( a_{21} + a_{22} \) | \( a_{11} + a_{22} \) | \( a_{11} - a_{21} \) | \( a_{12} - a_{22} \) |
| 4              | \( a_{31} + a_{33} \) | \( a_{42} + a_{44} \) | \( a_{31} + a_{32} \) | \( a_{41} + a_{42} \) | \( a_{31} + a_{42} \) | \( a_{31} - a_{41} \) | \( a_{32} - a_{42} \) |
| 5              | \( a_{11} + a_{33} \) | \( a_{22} + a_{44} \) | \( a_{11} + a_{12} \) | \( a_{21} + a_{22} \) | \( a_{11} + a_{22} \) | \( a_{11} - a_{21} \) | \( a_{12} - a_{22} \) |
| 6              | \( a_{11} - a_{31} \) | \( a_{22} - a_{42} \) | \( a_{11} + a_{12} \) | \( a_{21} + a_{22} \) | \( a_{11} + a_{22} \) | \( a_{11} - a_{21} \) | \( a_{12} - a_{22} \) |
| 7              | \( a_{13} - a_{33} \) | \( a_{24} - a_{44} \) | \( a_{13} + a_{14} \) | \( a_{23} + a_{24} \) | \( a_{13} + a_{24} \) | \( a_{13} - a_{23} \) | \( a_{14} - a_{24} \) |

a \( 4 \times 4 \) matrix. So, we have \( n^{\log_2 7} / 7^2 \) such matrices. This is repeated until we end with one \( n \times n \) matrix that is the result \( C = A \times B \). ◦

**Example 4.1.** Let \( n = 2^1; \ k = \log n = 1 \). In Part I, we compute \( f(v)(A) \) and \( f(v)(B) \). For instance,

\[
 f(v)(A) = (S_1(A), S_2(A), \ldots, S_7(A)) = (a_{11}, a_{22}, a_{11} + a_{12}, a_{21} + a_{22}, a_{11} + a_{22}, a_{11} - a_{21}, a_{12} - a_{22}).
\]

In Part II, we pair each value in \( f(v)(A) \) with a value in \( f(v)(B) \); \( S_i(A) \) is paired with \( S_{T[i]}(B) \) (see Table 4.1). For each such pair, their product \( S_i(A).S_{T[i]}(B) \) gives \( P_i(A, B) \).

In Part III, we compute the \( 2 \times 2 \) result matrix \( C \), using the equations for \( C_{ij} \) given in Chapter 3. ◦
Example 4.2. Let \( n = 2^2; k = \lg n = 2 \). In Part I, we compute \( f_v(A) \) and \( f_v(B) \). Each is a list of \( 7^k = 49 \) FVs. \( f_v(A) = (S_{i_1,i_2}(A)), 1 \leq i_1, i_2 \leq 7 \) (see Table 4.2). \( f_v(B) \) is similar.

In Part II, we pair each value in \( f_v(A) \) with a value in \( f_v(B) \). \( S_{i_1,i_2}(A) \) is paired with \( S_{T[i_1],T[i_2]}(B) \) (see Table 4.1). For example, \( S_{7,3}(A) \) is paired with \( S_{4,2}(B) \). For each pair, compute their product; this gives 49 reals.

In Part III, we first compute the \( 7 \times 2 \times 2 \) matrices \( P_i(A,B) \). To do this, we group the 49 reals from Part II into 7 groups of 7 reals each; each group is used to compute one \( P_i(A,B) \). For instance, \( P_7(A,B) = S_7(A) \times S_4(B) \) is computed from the group of 7 reals \( S_{7,i_2}(A).S_{4,T[i_2]}(B) \), using the equations for \( C_{ij}(S_7(A),S_4(B)) \) in Chapter 3. For instance, its \((1,2)\) entry is computed as:

\[
C_{12}(S_7(A),S_4(B)) = P_1(S_7(A),S_4(B)) + P_3(S_7(A),S_4(B))
\]

\[
= S_1(S_7(A)).S_7(S_4(B)) + S_3(S_7(A)).S_2(S_4(B))
\]

\[
= S_{7,1}(A).S_{4,7}(B) + S_{7,3}(A).S_{4,3}(B).
\]

Now, we have \( 7 \times 2 \times 2 \) matrices \( P_i(A,B) \), for a total of \( 7 \times 4 = 28 \) reals. Finally, using
these, we compute the $4 \times 4$ result matrix $C$, using the equations for $C_{ij}(A,B)$ (Chapter 3).

Of the 3 parts above, Parts II and III are already iterative. In Part III, we first compute $n^{k^2/7} 2 \times 2$ matrices; then $n^{k^2/7^2} 4 \times 4$ matrices; and so on. This can be implemented using 3 nested for loops; there is NO recursion. Only Part I is recursive; we now present an iterative version. Consider the computation of $fv(A)$.

**Definition 4.1.** $[|L_1|, L_1 \circ L_2, L_1 \oplus L_2, L_1 \ominus L_2]$ Let $L_1, L_2$ be two lists of real numbers.

- $|L_1|$ denotes the length (number of elements) of $L_1$.
- $L_1 \circ L_2$ denotes the concatenation of $L_1$ and $L_2$.
- If $|L_1| = |L_2|$: $L_1 \oplus L_2$ denotes the list whose $i$th element is the sum of the $i$th elements of $L_1$ and $L_2$.
- $L_1 \ominus L_2$ denotes the list whose $i$th element is the difference of the $i$th elements of $L_1$ and $L_2$.

**Fact 4.1.** $fv(A) = fv(S_1(A)) \circ fv(S_2(A)) \ldots \circ fv(S_7(A))$. Further, $fv(S_3(A)) = fv(A_{11} + A_{12}) = fv(A_{11}) \oplus fv(A_{12})$; in particular, $S_{i_2, \ldots, i_k}(A) = S_{i_2, \ldots, i_k}(A_{11}) + S_{i_2, \ldots, i_k}(A_{12})$. Similarly for $fv(S_4(A))$ and $fv(S_5(A))$. $fv(S_6(A)) = fv(A_{11} - A_{21}) = fv(A_{11}) \ominus fv(A_{21})$; similarly for $fv(S_7(A))$.

Fact 4.1 recursively expresses $fv(A)$ in terms of $fv(A_{11})-fv(A_{22})$. Unwinding this recursion gives the following iterative procedure for computing $fv(A)$:

1. Partition $A$ into $(n/2)^2 2 \times 2$ submatrices.
2. Replace each such submatrix by its $fv()$.
3. Using Fact 4.1, replace each $4 \times 4$ submatrix by its $fv()$; then, replace each $8 \times 8$ submatrix by its $fv()$; and so on, until we end with $fv(A)$.

This can be implemented using 3 nested for loops, in $\Theta(n^{k^2})$ time.
Now, consider a straight-forward mapReduce implementation of Parts I-III.

**Part I.** Compute $fv(A)$ and $fv(B)$; this takes $\lg n$ passes. Consider the computation of $fv(A)$. In the first pass, $A$ is transformed into $7 n/2 \times n/2$ matrices $S_{i_1}(A)$. In the second pass, each $S_{i_1}(A)$ is transformed into $7 n/4 \times n/4$ matrices $S_{i_2}(S_{i_1}(A))$; and so on. From each pass to the next, total number of reals output increases by the factor $7/4$.

In each pass, each reducer takes 1 or 2 reals and outputs one real. For instance, considering $S_3(A) = A_{11} + A_{12}$, for each $(x,y)$, a reducer gets the $(x,y)$th entries of $A_{11}$ and $A_{12}$, and outputs their sum. Maximum reducer size is $q_I = 2$, and maximum reducer workload is $w_I = \Theta(1)$.

Consider the $p$th pass, $1 \leq p \leq \lg n$.

Number of reducers is $R_p = 7^p (n/2^p)^2 = (7/4)^p n^2$;

communication cost is $CC_p = 2R_p$.

Total number of reducers is $R_I = \sum_p R_p = (7/3)n^{\lg 7}$.

Total communication cost is $CC_I = 2R_I = 2(7/3)n^{\lg 7}$.

Total work done is $W_I = \Theta(R_I) = \Theta(n^{\lg 7})$.

Same performance measures apply to computing $fv(B)$.

**Part II.** Each value in $fv(A)$ is multiplied by the corresponding value from $fv(B)$. So, we end with a total of $n^{\lg 7}$ reals. This takes one mapReduce pass.

Each reducer takes 2 reals and outputs one real. So, $q = 2$, $w = \Theta(1)$, and $R = n^{\lg 7}$; $CC = 2n^{\lg 7}$, and $W = \Theta(n^{\lg 7})$.

**Part III.** The $n^{\lg 7}$ reals from Part II are transformed (using the equations for $C_{ij}$ in Chapter 3) into $n^2$ reals that constitute the product matrix $C$. This takes $\lg n$ mapReduce passes.

In the first pass, the $n^{\lg 7}$ reals are transformed into $n^{\lg 7}/7 \times 2 \times 2$ matrices. In the
second pass, these matrices are transformed into \((n^{\log_7 7})^4\) 4 × 4 matrices; and so on. From each pass to the next, total number of reals output decreases by the factor \(4/7\).

Consider the equations for \(C_{11} - C_{22}\) in Chapter 3. In each pass, each reducer takes 2 or 4 reals (average over \(C_{11} - C_{22}\) is 3), and outputs one real. So, \(q = 4\) and \(w = \Theta(1)\).

For the \(p\)th pass, \(R_p = (4/7)^p n^{\log_7 7}\), and \(CC_p = 3R_p\).

So, \(R = \sum_p R_p = (4/3)n^{\log_7 7}\), \(CC = 3R = 4n^{\log_7 7}\), and \(W = \Theta(R) = \Theta(n^{\log_7 7})\).

Summarizing this, we have the following.

**Theorem 5.1.** Strassen’s algorithm for computing \(C = A \times B\) can be implemented using 2\(\log n\) + 1 mapReduce passes, divided into 3 parts:

**Part I.** Compute \(fv(A)\) and \(fv(B)\) in \(\log n\) passes.

**Part II.** Compute the product of each value in \(fv(A)\) with the corresponding value in \(fv(B)\), in one pass.

**Part III.** Compute \(C\) using the \(n^{\log_7 7}\) reals, in \(\log n\) passes.

For each part, \(q\) and \(w\) are \(\Theta(1)\); \(R, CC, W\) are \(\Theta(n^{\log_7 7})\).

Our objective is an implementation of Strassen’s algorithm with the following properties:

**D1.)** It uses only a constant number of passes.

**D2.** \(W\) and \(CC\) are \(O(n^{\log_7 7})\).

**D3.** \(q\) and \(w\) are \(O(n)\).

We believe that Part III can be so implemented; this is future work. The rest of this thesis deals exclusively with Part I; we show that it can be implemented in two passes.

In Chapter 6, we present some one-pass algorithms for Part I; they all fail to meet either D2 or D3. In Chapter 7, we present a two pass algorithm that meets both D2 and D3.
In this chapter, we consider one-pass algorithms to compute \( f_v(A) \). First, consider the two extremes for such algorithms. At one extreme, referred to as \textit{Extreme2} (E2), we have a single reducer that takes \( A \) as input, and computes \( f_v(A) \). This algorithm is just Part I of Strassen’s (sequential) algorithm. We have

\[
R = 1, q = n^2, CC = n^2, \text{ and } W = w = \Theta(n^{\log_7 7}).
\]  

(6.1)

This meets D2, but not D3. Among our one-pass algorithms, E2 is (essentially) the only one that meets D2.

In Section 6.1, we present the other extreme \textit{Extreme1} (E1). In Section 6.2, we present a more general one-pass algorithm; E1 and E2 are special cases of it.

### 6.1 Extreme1 One-Pass Algorithm

Each \( v \in f_v(A) \) is the sum/difference of at most \( n \) entries of \( A \) (see Table 4.2).

**Definition 6.1.** [signature/entries/size/full/nonfull FV] Consider an FV \( v = S_{i_1, i_2, \ldots, i_k}(A) \).

- **Signature of** \( v \), denoted by \( \text{sign}(v) \), is \( (i_1, i_2, \ldots, i_k) \).

- **Entries of** \( v \), denoted by \( E(v) \), is the set of entries of \( A \) needed to compute \( v \). \( E_{i_1, i_2, \ldots, i_k}(A) \) denotes \( E(v) \).

- **Size of** \( v \), denoted by \( s(v) \), is \( |E(v)| \).

- If \( s(v) = n \), \( v \) is a full FV (FFV); else (i.e., \( s(v) < n \)), \( v \) is a nonfull FV (NFFV). 

**Example 6.1.** Let \( n = 2^2, k = \log n = 2 \). Consider the FV \( v = S_{6,2}(A) = a_{22} - a_{42} \) (see Table 4.2); \( \text{sign}(v) = (6, 2) \). \( E(v) = E_{6,2}(A) = \{a_{22}, a_{42}\} \); \( s(v) = 2 \); \( v \) is an NFFV.
First, consider the Dumb Extreme1 (DE1) algorithm: There is one reducer to compute each FV. We have \( R = n^\lg 7 \),
\[
q = n, \quad w = \Theta(n); \quad CC = \sum_v s(v), \quad W = \Theta(CC).
\]
Based on the equations for \( S_1(A) - S_7(A) \) in Chapter 3,
\[
CC(n) = 2 \times CC(n/2) + 5 \times 2 \times CC(n/2) = 12 \times CC(n/2);
\]
the two terms in the sum correspond to computing \( fv(S_1(A)) - fv(S_2(A)) \) and \( fv(S_3(A)) - fv(S_7(A)) \), respectively. So,
\[
CC(n) = n^\lg 12 \quad \text{and} \quad W = \Theta(n^\lg 12).
\]
This \( W \) is much worse than the work \( \Theta(n^3) \) of the straight-forward sequential matrix multiplication algorithm. We can do better by keeping only reducers of size \( n \). Each such reducer will compute one FFV, and some NFFVs.

**Fact 6.1.** Consider an FV \( v = S_{i_1, i_2, \ldots, i_k}(A) \).

- \( s(v) = 2^x \), where \( x \) is the number of indices \( i_j \) (1 \( \leq \) \( j \) \( \leq \) \( k \)), such that 3 \( \leq \) \( i_j \) \( \leq \) 7.

- \( v \) is an FFV iff 3 \( \leq \) \( i_j \) \( \leq \) 7, for all \( j \). (By Equation 3.2, only submatrices of \#3 through \#7 are computed by adding or subtracting two submatrices)

- There are \( 5^\lg n = n^\lg 5 \) FFVs. \( \diamond \)

**Definition 6.2.** A full signature (FS) is the signature of an FFV. It is of the form \( (i_1, i_2, \ldots, i_k) \), 3 \( \leq \) \( i_j \) \( \leq \) 7, for all \( j \). \( \diamond \)

Our algorithm Extreme1 (E1) will have \( n^\lg 5 \) reducers. For each FS \( (i_1, i_2, \ldots, i_k) \), there is one reducer \( r_{i_1, i_2, \ldots, i_k} \). Its input is \( E_{i_1, i_2, \ldots, i_k}(A) \); so its size is \( n \). It will compute the FFV with that signature. It will also compute the NFFVs \( S_{i_1', i_2', \ldots, i_k'}(A) \): For 1 \( \leq \) \( j \) \( \leq \) \( k \),
\[
i_j' = \begin{cases} 
1 & \text{if } i_j = 3 \\
2 & \text{if } i_j = 4 
\end{cases} \quad (6.2)
\]

This means, for a certain FFV with a signature \( (i_1, i_2, \ldots, i_k) \), the reducer that contains the corresponding entries will not only compute this FFV, but also some NFFVs. The
Table 6.1: Notations from Chapter 6

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sign}(v)$</td>
<td>signature of FV $v \in f v(A)$</td>
</tr>
<tr>
<td>$\text{sign}(S_{i_1, i_2, \ldots, i_k}(A))$</td>
<td>$(i_1, i_2, \ldots, i_k)$</td>
</tr>
<tr>
<td>$E(v)$</td>
<td>entries of $A$ needed to compute $v \in f v(A)$</td>
</tr>
<tr>
<td>$E_{i_1, i_2, \ldots, i_k}(A)$</td>
<td>$E(S_{i_1, i_2, \ldots, i_k}(A))$</td>
</tr>
<tr>
<td>$s(v)$</td>
<td>size of FV $v$, is $</td>
</tr>
<tr>
<td>FFV</td>
<td>full FV $v$: $s(v) = n$</td>
</tr>
<tr>
<td>FFV</td>
<td>full FV $S_{i_1, i_2, \ldots, i_k}(A)$, $3 \leq i_1, \ldots, i_k \leq 7$</td>
</tr>
<tr>
<td>$f f v(A)$</td>
<td>list of all $n^{\log_5 5}$ FFVs of $A$</td>
</tr>
<tr>
<td>NFFV</td>
<td>nonfull FV $v$: $</td>
</tr>
<tr>
<td>FS</td>
<td>full signature: $(i_1, i_2, \ldots, i_k)$, $3 \leq i_1, \ldots, i_k \leq 7$</td>
</tr>
</tbody>
</table>

rules is whenever $i_j$ is 3, it computes the corresponding NFFV with $i'_j$ equals 1 at that position, and do the similar thing when $i_j$ is 4. The theory is that for $i'_j = 1$ or 2, the NFFV is computed simply by half of the entries needed to compute the FFV when $i_j$ is 3 or 4, because a index of 1 or 2 stands for a single submatrix rather than an addition/subtraction of two submatrices (Recall Equation 3.2).

So, the following equation indicates all the FVs, including both the FFV and the NFFVs, that one particular reducer $r_{i_1, i_2, \ldots, i_k}$ will compute:

$$
    i'_j = \begin{cases} 
    i_j & \text{if } 5 \leq i_j \leq 7 \\
    1 \text{ or } 3 & \text{if } i_j = 3 \\
    2 \text{ or } 4 & \text{if } i_j = 4 
    \end{cases} 
$$

(6.3)
Table 6.2: Example 6.3: For \( n = 2^2 \): \((i_1, i_2)\)th entry gives the NFFVs \( S_{i_1, i_2}(A) \) computed by reducer \( r_{i_1, i_2} \)

\[
\begin{array}{c|c|c|c|c|c|c}
 i_1 & i_2 & 3 & 4 & 5 & 6 & 7 \\
1 & 2 & (1, 1), (1, 3), (3, 1) & (1, 2), (1, 4), (3, 2) & (1, 5) & (1, 6) & (1, 7) \\
1 & 3 & (1, 2), (2, 3), (4, 1) & (2, 2), (2, 4), (4, 2) & (2, 5) & (2, 6) & (2, 7) \\
1 & 4 & (5, 1) & (5, 2) & & & \\
1 & 5 & (6, 1) & (6, 2) & & & \\
1 & 6 & (7, 1) & (7, 2) & & & \\
\end{array}
\]

Conversely, each FV \( S_{i_1, i_2, \ldots, i_k}(A) \) is computed by the reducer \( r_{i_1, i_2, \ldots, i_k} \): For \( 1 \leq j \leq k \),

\[
i_j = \begin{cases} 
  i_j' & \text{if } 3 \leq i_j' \leq 7 \\
  3 & \text{if } i_j' = 1 \\
  4 & \text{if } i_j' = 2
\end{cases}
\]

Note that \( E_{i_1, i_2, \ldots, i_k}(A) \subseteq E_{i_1, i_2, \ldots, i_k}(A) \).

Example 6.2. Let \( n = 2^1 \); \( k = \lg n = 1 \).

\[
fv(A) = (S_1(A), S_2(A), \ldots, S_7(A))
\]

\[
= (a_{11}, a_{22}, a_{11} + a_{12}, a_{21} + a_{22}, a_{11} + a_{22}, a_{11} - a_{21}, a_{12} - a_{22})
\]

There are 5 reducers, one for each FS \((i_1)\), \(3 \leq i_1 \leq 7\). Reducer \( r_{i_1} \) computes the FFV \( S_{i_1}(A) \). In addition, \( r_3 \) computes the FV \( S_1(A) = a_{11} \); \( r_4 \) computes \( S_2(A) = a_{22} \). ◦

Example 6.3. Let \( n = 2^2 \); \( k = \lg n = 2 \). \( fv(A) = (S_{i_1, i_2}(A)) \), \( 1 \leq i_1, i_2 \leq 7 \) (see Table 4.2).

There are \( 5^2 = 25 \) reducers, one for each FS \((i_1, i_2)\), \(3 \leq i_1, i_2 \leq 7\). Reducer \( r_{i_1, i_2} \) computes the FFV \( S_{i_1, i_2}(A) \); this accounts for the 25 FVs that are FFVs. The remaining \( 7^2 - 5^2 = 24 \) NFFVs are computed as shown in Table 6.2.

For instance, \( r_{3,5} \) computes the FFV \( S_{3,5}(A) = a_{11} + a_{22} + a_{13} + a_{24} \) (see Table 4.2), and the NFFV \( S_{1,5}(A) = a_{11} + a_{22} \). ◦
Consider Algorithm E1. Reducer \( r_{i_1,i_2,...,i_k} \) computes all the FVs \( S_{i'_1,i'_2,...,i'_k}(A) \), where \( i'_j \) are given by Equation 6.3. It does this recursively. For instance, if \( i_1 = 3 \), it first computes \( S_{i'_1,i'_2,...,i'_k}(A_{11}) \) and \( S_{i'_1,i'_2,...,i'_k}(A_{12}) \), for all such \( i'_j \). Then, using Fact 4.1, it outputs both \( S_{i'_1,i'_2,...,i'_k}(A_{11}) \) as \( S_{i_1,i'_2,...,i'_k}(A) \), and \( S_{i'_1,i'_2,...,i'_k}(A_{11}) + S_{i'_1,i'_2,...,i'_k}(A_{12}) \) as \( S_{i_1,i'_2,...,i'_k}(A) \).

Similarly for \( i_1 = 4 \).

For \( i_1 = 5 \), the reducer first computes \( S_{i'_1,i'_2,...,i'_k}(A_{11}) \) and \( S_{i'_1,i'_2,...,i'_k}(A_{22}) \). Then, using Fact 4.1, it outputs only
\[
S_{i'_1,i'_2,...,i'_k}(A_{11}) + S_{i'_1,i'_2,...,i'_k}(A_{22}) \text{ as } S_{i_1,i'_2,...,i'_k}(A).
\]
Similarly for \( i_1 = 6, 7 \).

Now, consider the reducer workload. Reducers \( r_{i_1,i_2,...,i_k} \) with each \( i_j = 3 \) or 4 compute the most number of FVs, and have the maximum workload. The number of FVs computed by such a reducer satisfies the recurrence \( N(n) = 2N(n/2), N(1) = 1; \) so, \( N(n) = n \). From the equation for \( fv(S_3(A)) \) and \( fv(S_4(A)) \) in Fact 4.1, the workload of such a reducer is given by the recurrence
\[
w(n) = 2w(n/2) + \Theta(N(n/2)) = 2w(n/2) + \Theta(n); \tag{6.4}
\]
so, \( w(n) = \Theta(n \lg n) \).

From now onwards, we will use asymptotic notations \( \tilde{O} \) and \( \tilde{\Theta} \) that ignore polylogarithmic factors [14]; recall that \( \lg^p n = o(n^\epsilon) \), for all constants \( \epsilon > 0 \) and \( p \) [3]. For example, our bound \( w(n) = \Theta(n \lg n) \) above gives \( w(n) = \tilde{\Theta}(n) \).

Table 6.3: Mapper for \( E1 \): \( j \)th component \( i_j \) of reducer signature \( (i_1, \ldots, i_j, \ldots, i_k) \) based on \(( (x - 1)_j, (y - 1)_j) \)

<table>
<thead>
<tr>
<th>((x - 1)_j)</th>
<th>((y - 1)_j)</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3, 5, 6</td>
<td>3, 7</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4, 6</td>
<td>4, 5, 7</td>
<td></td>
</tr>
</tbody>
</table>
Now, consider the mapping phase of Algorithm E1. Consider the equations for $S_3(A) - S_7(A)$ in Chapter 3. Each entry in $A_{11}$ needs to be mapped only to some reducers with $i_1 = 3, 5$ or 6; similarly for $A_{12}, A_{21}, A_{22}$. This rule has to be applied recursively within each submatrix, to determine the complete reducer signatures.

The complete rule is as follows. For an entry $a_{xy}$ in $A$, consider the $j$th bit ($1 \leq j \leq k$) of $x - 1$ (denoted by $(x - 1)_j$) and the $j$th bit of $y - 1$ (denoted by $(y - 1)_j$). Table 6.3 gives the values for the $j$th component $i_j$ of the reducer signature $i_1 \ldots i_k$. $a_{xy}$ should be sent to each such reducer; so, $a_{xy}$ gives rise to the key-value pairs $(i_1 \ldots i_j \ldots i_k, (x, y, a_{xy}))$.

**Example 6.4.** Let $n = 2^2; k = \log n = 2$. Consider the entry $a_{xy} = a_{12}$. $x - 1 = 0$ is 00 in binary, and $y - 1 = 1$ is 01 in binary. Based on their first bits (0, 0), $i_1$ is 3, 5 or 6. Based on their second bits (0, 1), $i_2$ is 3 or 7. $a_{12}$ should be sent to such reducers $r_{i_1, i_2}$; so, $a_{12}$ yields the 6 key-value pairs $(i_1 i_2, (1, 2, a_{12}))$, where $i_1 = 3, 5$ or 6, and $i_2 = 3$ or 7.

Note that the size of each key-value pair is $\Theta(\log n)$. This is another reason to use $\tilde{O}$ and $\tilde{\Theta}$, instead of $O$ and $\Theta$.

**Theorem 6.1.** Algorithm Extreme1 (E1) computes $f v(A)$ in one pass. It uses $R = n^{\log 5}$ reducers, each of size $q = \tilde{\Theta}(n)$; so, $CC = R.q = \tilde{\Theta}(n^{\log 10})$. Entries $a_{xx}$ on the main diagonal of $A$ are mapped to the most number of reducers; each such entry is mapped to $3^{\log n} = n^{\log 3}$ reducers.

$$w = \tilde{\Theta}(n); \quad W = O(R.w) = \tilde{\Theta}(n^{\log 10}). \tag{6.5}$$

### 6.2 General One-Pass Algorithm

Let $f f v(A)$ denote the list of FFVs of $A$, in increasing order of their signatures. Consider an FFV $v \in f f v(A)$; $v = S_{i_1, i_2, \ldots, i_k}(A), 3 \leq i_j \leq 7$. By Fact 4.1, $E(v)$ contains *identically situated entries* in two of the four submatrices $A_{11}, A_{12}, A_{21}, A_{22}$ of $A$; $E(v)$ contains no entries from the other two submatrices. For instance, if $i_1 = 3$,

$$E(v) = E_{3, i_2, \ldots, i_k}(A) = E_{i_2, \ldots, i_k}(A_{11}) \cup E_{i_2, \ldots, i_k}(A_{12});$$
so, $E(v)$ contains the $(i,j)$th entry of $A_{11}$ iff it contains the $(i,j)$th entry of $A_{12}$. This holds recursively within each submatrix.

Consider Algorithm E1 above. For each FS $(i_1,i_2,\ldots,i_k)$, reducer $r_{i_1,i_2,\ldots,i_k}$ gets the entries in $E_{i_2,\ldots,i_k}(A_{xy})$ from two of the four submatrices $A_{xy}$ of $A$; $|E_{i_2,\ldots,i_k}(A_{xy})| = n/2$. The reducer has size $n$, and can compute only one FFV. Instead, suppose we provide a reducer $r$ with the entries $E_{i_2,\ldots,i_k}(A_{xy})$ from three submatrices, say $A_{11}, A_{12}$ and $A_{22}$. Then, $r$ will have size $3n/2$, and can compute the 3 FFVs $S_{i_1,i_2,\ldots,i_k}(A)$, $i_1 = 3, 5, 7$. We will still need another reducer $r'$ (of size $3n/2$) that gets similar entries from $A_{11}, A_{21}$ and $A_{22}$, to compute $S_{i_1,i_2,\ldots,i_k}(A)$, $i_1 = 4, 6$. Then, $r$ and $r'$, together, would have size $3n$, and compute the 5 FFVs $S_{i_1,i_2,\ldots,i_k}(A)$, $3 \leq i_1 \leq 7$. This is an improvement over Algorithm E1 that uses 5 reducers (total size $5n$) to compute those 5 FFVs.

We can do even better. Have a single reducer $\hat{r}$ that gets the entries in $E_{i_2,\ldots,i_k}(A_{xy})$, from all the 4 submatrices $A_{xy}$. It would have size $2n$, and can compute all the 5 FFVs $S_{i_1,i_2,\ldots,i_k}(A)$, $3 \leq i_1 \leq 7$. We would need a total of $5^{k-1} = n^{\lg n}/5$ such reducers, each of size $2n$.

**Example 6.5.** Let $n = 2^2$; $k = \lg n = 2$. In Algorithm E1, consider the reducer $r_{3,7}$ that computes the FFV $S_{3,7}(A) = (a_{12} - a_{22}) + (a_{14} - a_{24})$. It gets the $n$ entries in

$$E_{3,7}(A) = E_7(A_{11}) \cup E_7(A_{12}) = \{a_{12}, a_{22}\} \cup \{a_{14}, a_{24}\},$$

and computes one FFV. Algorithm E1 uses 5 such reducers $r_{i_1,7}$ to compute the 5 FFVs $S_{i_1,7}(A)$; their total size is $5n$.

Instead, suppose we provided a reducer $r$ with

$$E_7(A_{11}) \cup E_7(A_{12}) \cup E_7(A_{22}) = \{a_{12}, a_{22}\} \cup \{a_{14}, a_{24}\} \cup \{a_{34}, a_{44}\}.$$ 

$r$ can compute the 3 FFVs $S_{i_1,7}(A)$, $i_1 = 3, 5, 7$. We would still need another reducer $r'$ that takes

$$E_7(A_{11}) \cup E_7(A_{21}) \cup E_7(A_{22}) = \{a_{12}, a_{22}\} \cup \{a_{32}, a_{42}\} \cup \{a_{34}, a_{44}\}.$$ 

$r'$ would compute the 2 FFVs $S_{i_1,7}(A)$, $i_1 = 4, 6$. Together, $r$ and $r'$ would have size $3n$, and compute 5 FFVs $S_{i_1,7}(A)$.
Even better, suppose we have a single reducer $r$ that takes

$$\cup_{x,y}E_7(A_{xy}) = \{a_{12}, a_{22}\} \cup \{a_{14}, a_{24}\} \cup \{a_{32}, a_{42}\} \cup \{a_{34}, a_{44}\}.$$ 

It would have size $2n$, and can compute all 5 FFVs $S_{i_1,7}(A)$. We would need a total of 5 such reducers $r$, $3 \leq i_2 \leq 7$. Each reducer $r$ computes 5 FFVs $S_{i_1, i_2}(A)$, $3 \leq i_1 \leq 7$. 

So, compared to Algorithm E1 that had $n^{lg 5}$ reducers, each of size $n$, we now have $n^{lg 5}/5$ reducers, each of size $2n$. This can be extended to any integer $m$, $0 \leq m \leq k$: We would have $n^{lg 5}/5^m$ reducers, each of size $2^m n$; each reducer would compute $5^m$ FFVs. Our **General One-Pass Algorithm** ($G_m$) is as follows. Subdivide matrix $A$ into $2^m$ submatrices $A'_{de}$, $1 \leq d, e \leq 2^m$, each $A'_{de}$ is a $2^{k-m} \times 2^{k-m}$ matrix. For each FS $(i_{m+1}, \ldots, i_k)$, there is one reducer $r_{i_{m+1}, i_{m+2}, \ldots, i_k}$.

Its input is $\cup_{d, e}E_{i_{m+1}, i_{m+2}, \ldots, i_k}(A'_{de})$; so, its size $q = \Theta(2^{k-m} \cdot 2^{2m}) = \Theta(2^m \cdot n)$.

Number of reducers is $R = 5^{k-m} = n^{lg 5}/5^m$.

$$CC = R \cdot q = \Theta((2/5)^m n^{lg 10}).$$

Reducer $r_{i_{m+1}, i_{m+2}, \ldots, i_k}$ computes the $5^m$ FFVs $S_{i_1, \ldots, i_{m}, i_{m+1}, \ldots, i_k}(A)$, $3 \leq i_1, \ldots, i_m \leq 7$.

It also computes the FVs $S_{i_1, \ldots, i_{m}, i'_{m+1}, \ldots, i'_k}(A)$, where:

$1 \leq i_1, \ldots, i_m \leq 7; \ i'_j, m+1 \leq j \leq k$, are given by Equation 6.3.

It does this recursively. It first computes the FVs

$S_{i_2, \ldots, i_{m}, i'_{m+1}, \ldots, i'_k}(A_{xy})$ for all four submatrices $A_{xy}$ of $A$ (Figure 3.1). It then outputs:

$$S_{i_1, i_2, \ldots, i_{m}, i'_{m+1}, \ldots, i'_k}(A) = S_{i_2, \ldots, i_{m}, i'_{m+1}, \ldots, i'_k}(A_{11}),$$

$$S_{i_1, i_2, \ldots, i_{m}, i'_{m+1}, \ldots, i'_k}(A) = S_{i_2, \ldots, i_{m}, i'_{m+1}, \ldots, i'_k}(A_{12}),$$

$$S_{i_1, i_2, \ldots, i_{m}, i'_{m+1}, \ldots, i'_k}(A) = S_{i_2, \ldots, i_{m}, i'_{m+1}, \ldots, i'_k}(A_{22}),$$

and so on, till

$$S_{i_1, i_2, \ldots, i_{m}, i'_{m+1}, \ldots, i'_k}(A) = S_{i_2, \ldots, i_{m}, i'_{m+1}, \ldots, i'_k}(A_{12}) - S_{i_2, \ldots, i_{m}, i'_{m+1}, \ldots, i'_k}(A_{22})$$

(6.7)
When the recursion gets down to the submatrices $A'_{de}$, the reducer has only the $2^{k-m}$ entries in $E_{i_m+1,i_{m+2},...,i_k}(A'_{de})$; it computes $S_{i_m+1,...,i_k}(A'_{de})$ as Algorithm E1 does in Section 6.1.

**Example 6.6.** Let $n = 2^5$, $m = 2$; so, $k = 5$, $k - m = 3$.

Matrix $A$ is subdivided into $2^{2m} = 16$ submatrices $A'_{de}$, $1 \leq d, e \leq 2^m = 4$; each $A'_{de}$ is a $2^3 \times 2^3$ matrix.

For each FS $(i_3, i_4, i_5)$, there is a reducer $\hat{r}_{i_3,i_4,i_5}$. Consider the reducer $\hat{r}_{5,3,7}$. Its input is $\cup_{d,e} E_{5,3,7}(A'_{de})$. $|E_{5,3,7}(A'_{de})| = \dim(A'_{de}) = 2^3$; $q = 2^4.2^3 = 2^2.2^5 = 2^m n$. It outputs the $5^m = 25$ FFVs $S_{1_1,i_2,5,3,7}(A)$, $3 \leq i_1, i_2 \leq 7$. Altogether, it outputs FFVs $S_{1_1,i_2,i_3',i_4',i_5'}(A)$, $1 \leq i_1, i_2 \leq 7$; and $i_j'$ are given by Equation 6.3; $i_3' = 5$, $i_5' = 7$, and $i_4' = 1$ or $3$.

Summarizing this, $\hat{r}_{5,3,7}$ outputs:

$5^m = 25$ FFVs $S_{i_1,i_2,5,3,7}(A)$, $3 \leq i_1, i_2 \leq 7$;

and $7^2 = 49$ NFFVs $S_{i_1,i_2,5,1,7}(A)$, $1 \leq i_1, i_2 \leq 7$.

It computes them recursively. It first computes the FVs:

$S_{i_2,i_3',i_4',i_5'}(A_{xy})$, for all 4 submatrices $A_{xy}$ of $A$ (Figure 3.1).

From them, it computes the FVs $S_{i_1,i_2,i_3',i_4',i_5'}(A)$, $1 \leq i_1 \leq 7$, using the equations for $S_1(A)$-$S_7(A)$ in Chapter 3 (and Fact 4.1).

Consider the computation of $S_{i_2,i_3',i_4',i_5'}(A_{xy})$, for a particular submatrix $A_{xy}$, say $A_{11}$. $\hat{r}_{5,3,7}$ does this recursively. It first computes the FVs:

$S_{5,i_4',7}(A_{11})_{xy}$, for all 4 submatrices $(A_{11})_{xy}$ of $A_{11}$.

From them, it computes the FVs $S_{i_2,i_3',i_4',i_5'}(A_{11})$, using the equations for $S_1(A_{11})$-$S_7(A_{11})$ in Chapter 3.

Now, consider the computation of $S_{5,i_4',7}(A_{11})_{xy}$.

Each $(A_{11})_{xy}$ is a $2^{k-m} \times 2^{k-m}$ submatrix $A'_{de}$; for instance, $(A_{11})_{22}$ is $A'_{22}$.

The recursion bottoms-out here.

Reducer $\hat{r}_{5,3,7}$ has the $2^{k-m}$ entries in $E_{5,3,7}(A'_{de})$. It computes the FVs $S_{5,i_4',7}(A'_{de})$, as explained in Chapter 6.1. ◀
Let \( w(x, y) \) denote the maximum workload of a reducer of size \( y \), on a matrix of dimension \( x \); let \( N(x, y) \) denote the number of FVs computed by it. From Chapter 6.1, we have \( N(n, n) = n \); from above, we have the recurrence

\[
N(n, 2^m n) = 7N(n/2, 2^{m-1}(n/2)); \quad \text{so,} \quad N(n, 2^m n) = (7/2)^m n.
\]

From Chapter 6.1, we have \( w(n, n) = \Theta(n) \). From above,

\[
w(n, 2^m n) = 4w(n/2, 2^{m-1}(n/2)) + 5\Theta(N(n/2, 2^{m-1}(n/2))).
\]

Its solution is \( w(n, 2^m n) = \Theta(N(n, 2^m n)) = \Theta((7/2)^m n) \).

Now, consider the mapping phase of Algorithm \( G_m \). Modifying the rule for Algorithm \( E_1 \) (Chapter 6.1), we have the following. For an entry \( a_{xy} \) in \( A \), and for \( 1 \leq j \leq k - m \), consider the \( j \)th bits of

\[
[(x - 1) \mod 2^{k-m}] \text{ and } [(y - 1) \mod 2^{k-m}].
\]

Table 6.3 gives the values for the \( j \)th component \( i_{m+j} \) of the reducer signature \( i_{m+1} \ldots i_{m+j} \ldots i_k \).

So, \( a_{xy} \) gives rise to the key-value pairs \((i_{m+1} \ldots i_{m+j} \ldots i_k, (x, y, a_{xy}))\).

**Theorem 6.2.** For any integer \( m, 0 \leq m \leq k = \lg n \), Algorithm \( G_m \) computes \( f v(A) \) in one pass. It has \( R = n^{\lg 5}/5^m \) reducers, each of size \( q = \Theta(2^m n) \); so, \( CC = R.q = \Theta((2/5)^m n^{\lg 10}) \).

Entries \( a_{xy} \), where \( x - y = 0 \mod 2^{k-m} \), are mapped to the most number of reducers; each such entry is mapped to \( 3^{k-m} = n^{\lg 3}/3^m \) reducers.

\[
w = \Theta((7/2)^m n); \quad W = O(R.w) = \Theta((7/10)^m n^{\lg 10}).
\]

Note that Algorithm \( G_0 \) is \( E_1 \), and \( G_k \) is \( E_2 \). \( G_m \) provides a range of algorithms between the two extremes \( E_1 \) and \( E_2 \). To have \( W = \tilde{O}(n^{\lg 7}) \), we need \( m = k - O(1) \); so, \( G_k = E_2 \) is essentially the only algorithm (among our one-pass algorithms) that meets the requirement D2 (Chapter 5).

But, for \( m > ((\lg 8/10)/(\lg 7/10)) \lg n \approx 0.63 \lg n \), we can get \( W = o(n^3) \). Letting \( m = \alpha \lg n \), and \( G'_\alpha \equiv G_{\alpha \lg n} \) in Theorem 6.2, we have the following.

**Corollary 6.1.** Let \( \alpha, 0 \leq \alpha \leq 1 \), be a constant, such that \( m = \alpha \lg n \) is an integer.

Algorithm \( G'_\alpha \equiv G_{\alpha \lg n} \) computes \( f v(A) \) in one pass. It has
\[ R = n^{(1-\alpha)\lg 5} \text{ reducers, each of size } q = \tilde{\Theta}(n^{1+\alpha}); \]

so, \[ CC = R.q = \tilde{\Theta}(n^{1+\alpha+(1-\alpha)\lg 5}). \]

Each entry in \( A \) is mapped to at most \( n^{(1-\alpha)\lg 3} \) reducers.

\[ w = \tilde{\Theta}(n^{1-\alpha+5\lg 7}); \quad W = O(R.w) = \tilde{\Theta}(n^{(1-\alpha)(1+\lg 5)+\alpha\lg 7}). \]

In particular, we have

**Corollary 6.2.** Algorithm \( G'_{2/3} \equiv G_{(2/3)\lg n} \) uses

\[ R = n^{(1/3)\lg 5} = n^{0.774} \text{ reducers, each of size } q = \tilde{\Theta}(n^{5/3}); \]

so, \[ CC = R.q = \tilde{\Theta}(n^{5+\lg 5}/3) = \tilde{\Theta}(n^{2.44}); \]

\[ w = \tilde{\Theta}(n^{1+2\lg 7}/3) = \tilde{\Theta}(n^{2.205}); \]

\[ W = O(R.w) = \tilde{\Theta}(n^{(1+\lg 5+2\lg 7)/3}) = \tilde{\Theta}(n^{2.98}). \]
CHAPTER 7
Two-Pass Algorithms

In this chapter, we consider two-pass algorithms to compute $fv(A)$. In Section 7.1, we present a basic two-pass algorithm. In Section 7.2, we present a more general two-pass algorithm; it combines the ideas from Sections 6.2 and 7.1.

7.1 Basic Two-Pass Algorithm

Let $h$ be an integer, $0 \leq h \leq k = \lg n$. The Basic Algorithm $B_h$ is as follows. Subdivide $A$ into $2^h$ submatrices $A'_{de}$, $1 \leq d, e \leq 2^h$; each $A'_{de}$ is a $2^{k-h} \times 2^{k-h}$ matrix. In the first pass, there is one reducer $r_{de}$, for each submatrix $A'_{de}$. It computes $fv(A'_{de})$ using Algorithm E2 (Chapter 6). So,

$$R_1 = 2^{2h}, \quad q_1 = \tilde{\Theta}(2^{2(k-h)}), \quad \text{and} \quad CC_1 = R_1.q_1 = \tilde{\Theta}(n^2);$$

$$w_1 = \tilde{\Theta}(7^{k-h}); \quad W_1 = O(R_1.w_1) = \tilde{\Theta}(2^{2h}.7^{k-h}).$$

Now, we have $fv(A'_{de}), 1 \leq d, e \leq 2^h$. Each $fv(A'_{de})$ is a list of $7^{k-h}$ values

$$S_{i_{h+1},i_{h+2},\ldots,i_k}(A'_{de}), 1 \leq i_{h+1}, i_{h+2}, \ldots, i_k \leq 7.$$  

For each signature $(i_{h+1}, \ldots, i_k),$ let $\hat{A}_{i_{h+1},\ldots,i_k}$ be the $2^h \times 2^h$ matrix whose $(d,e)$th entry is $S_{i_{h+1},i_{h+2},\ldots,i_k}(A'_{de}).$

In the second pass, there are $7^{k-h}$ reducers. Reducer $\hat{r}_{i_{h+1},\ldots,i_k}$ gets $\hat{A}_{i_{h+1},\ldots,i_k}$ as input. It computes $fv(\hat{A}_{i_{h+1},\ldots,i_k})$ using Algorithm E2. This output, over all the reducers $\hat{r}$, constitutes $fv(A)$:

**Fact 7.1.** For $1 \leq i_1, i_2, \ldots, i_h, i_{h+1}, \ldots, i_k \leq 7$: $S_{i_1,i_2,\ldots,i_h}(\hat{A}_{i_{h+1},\ldots,i_k}) = S_{i_1,i_2,\ldots,i_k}(A)$. ⊗

For the second pass, we have

$$R_2 = 7^{k-h}, \quad q_2 = \tilde{\Theta}(2^{2h}), \quad \text{and} \quad CC_2 = R_2.q_2 = \tilde{\Theta}(2^{2h}.7^{k-h});$$

$$w_2 = \tilde{\Theta}(7^h); \quad W_2 = O(R_2.w_2) = \tilde{\Theta}(n^{\lg 7}).$$

26
Theorem 7.1. For any integer \( h, 0 \leq h \leq k = \lg n \), Basic Algorithm \( B_h \) computes \( f_v(A) \) in two passes. It has

\[
R = R_1 + R_2 = 2^{2h} + 7^{k-h},
\]

\[
q = \max(q_1, q_2) = \tilde{\Theta}(2^{\max(k-h,h)}),
\]

and \( CC = CC_1 + CC_2 = \tilde{\Theta}(n^2 + 2^{2h}7^{k-h}). \)

In each pass, each item is mapped to only one reducer.

\[
w = \max(w_1, w_2) = \tilde{\Theta}(7^{\max(k-h,h)});
\]

\[
W = W_1 + W_2 = \tilde{\Theta}(n^{\lg 7}).
\]

So, when \( h = \lfloor k/2 \rfloor \), \( q \) and \( w \) have their minimum values \( \tilde{\Theta}(n) \), and \( \tilde{\Theta}(n^{(1/2)\lg 7}) \); \( R = n^{(1/2)\lg 7}, CC = \tilde{\Theta}(n^{1+(1/2)\lg 7}) \), and \( W = \tilde{\Theta}(n^{\lg 7}) \). \( \diamond \)

Let \( \delta, 0 \leq \delta \leq 1 \), be a constant, such that \( h = \delta \lg n \) is an integer. For use in Section 7.2, we restate the Basic Algorithm \( B'_\delta \equiv B_h \) in terms of \( \delta \). We first subdivide \( A \) into \( n^{2\delta} \) submatrices \( A'_{de}, 1 \leq d, e \leq n^{\delta} \); each \( A'_{de} \) is a \( n^{1-\delta} \times n^{1-\delta} \) matrix. In the first pass, reducer \( r_{de} \) computes \( f_v(A'_{de}) \) using Algorithm E2. So,

\[
R_1 = n^{2\delta}, q_1 = \tilde{\Theta}(n^{2(1-\delta)}), \text{ and } CC_1 = R_1.q_1 = \tilde{\Theta}(n^2);
\]

\[
w_1 = \tilde{\Theta}(n^{(1-\delta)\lg 7}); \text{ } W_1 = O(R_1.w_1) = \tilde{\Theta}(n^{2\delta+(1-\delta)\lg 7}).
\]

Now, we have \( f_v(A'_{de}), 1 \leq d, e \leq n^{\delta} \). Each \( f_v(A'_{de}) \) is a list of \( n^{(1-\delta)\lg 7} \) values. For each \( t, 1 \leq t \leq n^{(1-\delta)\lg 7} \), let \( \hat{A}_t \) be the \( n^{\delta} \times n^{\delta} \) matrix whose \((d,e)\)th entry is the \( t \)th value in \( f_v(A'_{de}) \).

In the second pass, there are \( n^{(1-\delta)\lg 7} \) reducers. Reducer \( \hat{r}_t \) gets \( \hat{A}_t \) as input. It outputs \( f_v(\hat{A}_t) \) using Algorithm E2. This output, over all the reducers \( \hat{r} \), constitutes \( f_v(A) \). We have

\[
R_2 = n^{(1-\delta)\lg 7}, q_2 = \tilde{\Theta}(n^{2\delta}), \text{ and } CC_2 = R_2.q_2 = \tilde{\Theta}(n^{2\delta+(1-\delta)\lg 7});
\]

\[
w_2 = \tilde{\Theta}(n^{\delta\lg 7}); \text{ } W_2 = O(R_2.w_2) = \tilde{\Theta}(n^{\lg 7}).
\]

Corollary 7.1. Let \( \delta, 0 \leq \delta \leq 1 \), be a constant, such that \( h = \delta \lg n \) is an integer. Basic Algorithm \( B'_\delta \equiv B_{\delta\lg n} \) computes \( f_v(A) \) in two passes. It has

\[
R = R_1 + R_2 = n^{2\delta} + n^{(1-\delta)\lg 7},
\]
\[ q = \max(q_1, q_2) = \tilde{\Theta}(n^{2\max(1-\delta,\delta)}) , \]
and \[ CC = CC_1 + CC_2 = \tilde{\Theta}(n^2 + n^{2\delta+(1-\delta)\lg 7}). \]

In each pass, each item is mapped to only one reducer.

\[ w = \max(w_1, w_2) = \tilde{\Theta}(n^{(\lg 7)\max(1-\delta,\delta)}); \]
\[ W = W_1 + W_2 = \tilde{\Theta}(n^{\lg 7}). \]

When \( \delta = 1/2 \), \( q \) and \( w \) have their minimum values \( \tilde{\Theta}(n) \), and \( \tilde{\Theta}(n^{(1/2)\lg 7}) \); \( R = n^{(1/2)\lg 7} \), \( CC = \tilde{\Theta}(n^{1+(1/2)\lg 7}) \), and \( W = \tilde{\Theta}(n^{\lg 7}). \)

So, Algorithm \( B'_{0,5} \) meets requirement D2, but not D3.

### 7.2 General Two-Pass Algorithm

The basic two-pass algorithm \( B_\delta \) (Section 7.1) uses Algorithm E2 to compute \( f v(A_{de}) \) in the first pass. The general two-pass algorithm differs from \( B_\delta \) in only the first pass: It uses Algorithm \( G'_\alpha \) (Section 6.2), instead of E2.

Consider constants \( \alpha \) and \( \delta, 0 \leq \alpha, \delta \leq 1 \), such that \( h = \delta \lg n \) and \( m = (1-\delta) \lg n \) are integers. The General Two-Pass Algorithm \( T_{\delta,\alpha} \) uses Algorithm \( G'_\alpha \) in the first pass.

In the first pass of \( T_{\delta,\alpha} \), we have \( n^{2\delta} \) submatrices \( A'_{de}, 1 \leq d, e \leq n^\delta \). Each \( A'_{de} \) is a \( n^{1-\delta} \times n^{1-\delta} \) matrix. To compute each \( f v(A'_{de}) \), Algorithm \( G'_\alpha \) uses (Corollary 6.1)

\[ R = n^{(1-\delta)(1-\alpha)\lg 5}, \text{ and } q_1 = \tilde{\Theta}(n^{(1-\delta)(1+\alpha)}); \]
\[ CC = R \cdot q_1 = \tilde{\Theta}(n^{(1-\delta)[1+\alpha+(1-\alpha)\lg 5]}). \]

Each entry in \( A \) is mapped to \( \leq n^{(1-\delta)(1-\alpha)\lg 3} \) reducers.

\[ w_1 = \tilde{\Theta}(n^{(1-\delta)(1-\alpha+\lg 5)}); \]
\[ W = O(R \cdot w_1) = \tilde{\Theta}(n^{(1-\delta)[(1-\alpha)(1+\lg 5)+\alpha \lg 7]}). \]

Over all the \( n^{2\delta} \) submatrices \( A'_{de}, R_1, CC_1 \) and \( W_1 \) are \( n^{2\delta} \) times the values for \( R, CC \) and \( W \) above; \( q_1 \) and \( w_1 \) are as specified above. The measures for the second pass are as for Algorithm \( B'_{\delta} \), given prior to Corollary 7.1.

**Theorem 7.2.** Let \( \alpha \) and \( \delta, 0 \leq \alpha, \delta \leq 1 \), be constants, such that \( h = \delta \lg n \) and \( m = \)
\( \alpha(1 - \delta) \log n \) are integers. The General Two-Pass Algorithm \( T_{\delta, \alpha} \) computes \( fV(A) \) in two passes. It has

\[
R = R_1 + R_2 = n^{2\delta + (1 - \delta)(1 - \alpha) \log 5} + n^{(1 - \delta) \log 7},
\]

\[
q = \max(q_1, q_2) = \tilde{\Theta}(n^{\max((1 - \delta)(1 + \alpha), 2\delta)}),
\]

and \( CC = CC_1 + CC_2 = \tilde{\Theta}(n^{(1 + \delta) + (1 - \delta)[\alpha + (1 - \alpha) \log 5]} + n^{2\delta + (1 - \delta) \log 7}) \).

In the first pass, each entry in \( A \) is mapped to at most \( n^{(1 - \delta)(1 - \alpha) \log 3} \) reducers. In the second pass, each value is mapped to only one reducer.

\[
w = \max(w_1, w_2) = \tilde{\Theta}(n^{\max((1 - \delta)(1 - \alpha + \alpha \log 7), \delta \log 7)}) ;
\]

\[
W = W_1 + W_2 = \tilde{\Theta}(n^{1 + \delta + (1 - \delta) [\log 5 - \alpha (1 + \log 5) + \alpha \log 7] + \log 7}). \]

Consider the performance measures given in Theorem 7.2. First, note that \( CC = o(W) \). So, to meet requirement D2, we only need \( W_1 = O(n^{\log 7}) \); i.e.,

\[
1 + \delta + (1 - \delta)(\log 5 - \alpha \log 10/7) \leq \log 7. \]

So,

\[
\alpha \geq \frac{1 - \delta(\log 5/2)/(\log 10/7)}{1 - \delta} \approx \frac{1 - 2.57\delta}{1 - \delta} \tag{7.1}
\]

Consider requirement D3. Note that \( q_2 = \tilde{\Theta}(n^{2\delta}) \) and \( w_2 = \tilde{\Theta}(n^{\delta \log 7}) \) are independent of \( \alpha \); but \( q_1 \) and \( w_1 \) increase as \( \alpha \) increases. So, we will take \( \alpha \) to be the minimum value in Equation 7.1. Then,

\[
q_1 = \tilde{\Theta}(n^{(1 - \delta)(1 + \alpha)}) = \tilde{\Theta}(n^{2 - 3.57\delta}),
\]

\[
w_1 = \tilde{\Theta}(n^{(1 - \delta)(1 + \alpha \log 7/2)}) = \tilde{\Theta}(n^{\log 7 - 5.645\delta}).
\]

We consider two distinct optimizations: minimizing \( q \), and minimizing \( w \). For minimizing \( q \), note that \( q_1 \) decreases as \( q_2 \) increases (as \( \delta \) increases). So, \( q \) is minimized when \( q_1 = q_2 \); i.e., \( 2 - 3.57\delta = 2\delta \), or \( \delta = 2/5.57 \approx 0.36 \).

**Corollary 7.2.** Among the values of \((\delta, \alpha)\) for which Algorithm \( T_{\delta, \alpha} \) meets requirement D2, the algorithm has minimum reducer size when \( \delta \approx 0.36 \), and \( \alpha \approx (1 - 2.57\delta)/(1 - \delta) \approx 0.117 \). Algorithm \( T_{0.36,0.117} \) has the following performance measures.

\[
R = R_1 + R_2 = n^{2.032} + n^{1.797}, \quad q = \tilde{\Theta}(n^{0.72}),
\]

29
and \( CC = CC_1 + CC_2 = \tilde{\Theta}(n^{2.75} + n^{2.52}). \)

In the first pass, each entry in \( A \) is mapped to at most \( n^{0.896} \) reducers. In the second pass, each value is mapped to only one reducer.

\[
w = \max(w_1, w_2) = \tilde{\Theta}(\max(n^{0.775}, n^{1.01})), \quad W = \tilde{\Theta}(n^{\lg 7}). \quad \diamond
\]

Now, consider minimizing \( w \). As for \( q \), \( w \) is minimized when \( w_1 = w_2 \); i.e., \( \lg 7 - 5.645\delta = \delta \lg 7 \), or \( \delta = \lg 7/(\lg 7 + 5.645) \approx 0.332. \)

**Corollary 7.3.** Among the values of \( (\delta, \alpha) \) for which Algorithm \( T_{\delta, \alpha} \) meets requirement D2, the algorithm has minimum reducer workload when \( \delta \approx 0.332 \), and \( \alpha \approx (1 - 2.57\delta)/(1 - \delta) \approx 0.219. \) Algorithm \( T_{0.332, 0.219} \) has the following performance measures.

\[
R = R_1 + R_2 = n^{1.875} + n^{1.875},
\]

\[
q = \max(q_1, q_2) = \tilde{\Theta}(\max(n^{0.814}, n^{0.664})),
\]

and \( CC = CC_1 + CC_2 = \tilde{\Theta}(n^{2.69} + n^{2.54}). \)

In the first pass, each entry in \( A \) is mapped to at most \( n^{0.827} \) reducers. In the second pass, each value is mapped to only one reducer.

\[
w = \tilde{\Theta}(n^{0.932}), \quad W = \tilde{\Theta}(n^{\lg 7}). \quad \diamond
\]

Note that the algorithm in Corollary 7.2 barely misses meeting requirement D3, because of \( w_2 \). The algorithm in Corollary 7.3 meets requirement D3 (and D2).
In this thesis, we studied the multiplication of two $n \times n$ matrices. A straight-forward sequential algorithm for computing the product takes $\Theta(n^3)$ time. Several papers have studied mapReduce implementations of this algorithm. This algorithm can be implemented using a constant number mapReduce passes. [1, 15] presented tight bounds on the tradeoff between communication cost and parallelism, for one- and two-pass algorithms.

Strassen [17] presented a sequential algorithm that takes $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$ time. In this thesis, we studied the mapReduce implementation of this recursive algorithm. If we unwind the recursion, this algorithm consists of three parts, Parts I-III. Direct mapReduce implementations of the three parts take $\lg n$, 1 and $\lg n$ passes, for a total of $2\lg n + 1$ passes. We showed that Part I can be implemented in 2 passes, with small reducer size and reducer workload. We believe that Part III can also be implemented in a constant number of passes (possibly 4 passes); this is future work.
LIST OF REFERENCES


