Single Stage Matrix Vector Multiplication via Hadoop Map-Reduce

by

Johann Setiawan

A research paper
presented to the University of Waterloo
in partial fulfillment of the
requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Supervisor: Prof. Hans De Sterck

Waterloo, Ontario, Canada, 2011

I hereby declare that I am the sole author of this report. This is a true copy of the report, including any required final revisions, as accepted by my examiners.

I understand that my report may be made electronically available to the public.

Abstract

This report explores an algorithm to perform Power method iterations in the cloud via Hadoop Map-Reduce. With the inclusion of a preprocessing procedure, only a single Map-Reduce stage is required per iteration. Several Optimizations were considered that improved the speed of the algorithm. The proposed approach can also be altered such that under certain conditions, the preprocessing is incorporated into the first multiplication step. It can also be extended to perform Jacobi method iterations.

Acknowledgements

I would like to thank Professor Hans De Sterck and my fellow classmates who have helped me through out this process.

Dedication

This is dedicated to my Parents, who have sacrificed so much in order for me to have a better future.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The use of cloud computing to tackle computations on very large datasets provides interesting possibilities, especially since it can be scalable and cost-effective through the use of commodity machines. A popular open source cloud management framework is Hadoop. Through the use of the Hadoop File System (HDFS) and the Map-Reduce programming model this framework offers a reliable shared storage and analysis system [9] that can easily be deployed. A computation that can be performed in Hadoop is the repeated matrix-vector multiplication also known as the Power method, where the matrix is fixed, and the vector is updated each step until convergence is achieved. For example, the graph mining method Page-Rank performs such actions to determine the ranking of web pages.

The Generalized Iterative Matrix-Vector Multiplication (GIM-V) is a Map-Reduce method that is part of the Graph Mining Library PeGaSus [7]. This method requires two Map-Reduce stages chained together in order to perform one matrix-vector multiplication. In the case of Power methods where the base matrix remains unaltered for all iterations, a question can be asked whether it is possible to perform one multiplication in one Map-Reduce stage. With an initial fixed cost in a preprocessing step that couples the matrix and vector information together in a specific manner, such calculations can indeed be achieved. We will discuss the trade-off cost and performance when compared to the GIM-V approach. Initial tests indicate that we can observe good results. However, there are potential issues that should be considered before utilizing this approach.

Several forms of optimization were considered for both methods; by leveraging existing functionality in the Hadoop framework we can obtain faster speeds and potential reduction in data transfer between nodes in the cloud. We will also discuss how to build upon these algorithms in order to perform Jacobi iterations for solving matrix systems.

Chapter 2

Background and Related Work

2.1 Hadoop

Hadoop is a software framework developed under the Apache Project. Published papers on Google's File System (GFS) [6] and Map-Reduce framework [5] were the main inspiration to the development of Hadoop, and it was initially used to address the scaling issues of the WebCrawler engine Nutch. The main components of this framework are the Hadoop Distributed File System (HDFS) and Map-Reduce model. Scalability, data localization and fault tolerance are achieved in tandem.

- **Scalability**, when a Hadoop program is written, little if any change is required if the input data sizes or the available cluster resources are scaled up, and the program remains efficient for large problem sizes.
- **Data Locality**, the strategy of moving computation to the data, alleviating strain on network bandwidth [4]
- **Fault Tolerance**, any failed tasks is automatically reassigned to a free node

2.1.1 HDFS

HDFS is designed to reliably store and support very large amounts of information [2]. Data blocks are replicated and distributed among the cloud nodes. If an individual node in the cluster malfunctions, the data is still available (see Fig 2.1). HDFS is especially designed to sequentially read large data files.

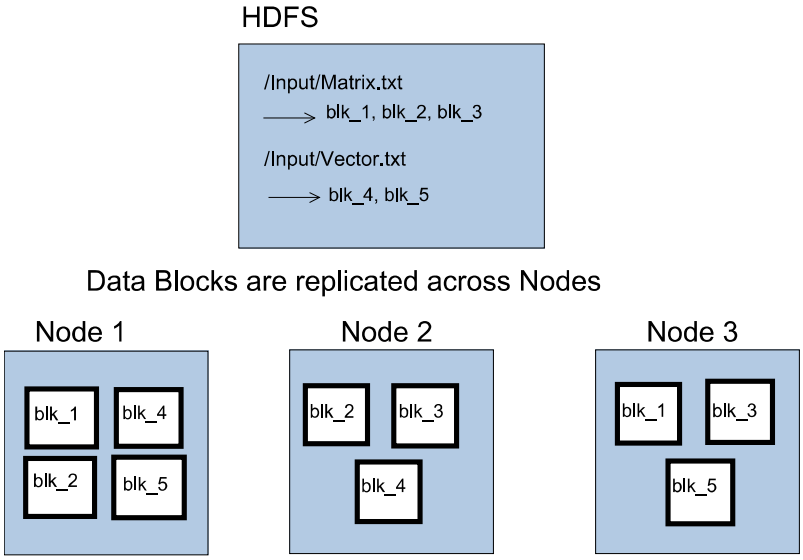

Figure 2.1: HDFS Data Blocks and Replication.

2.1.2 Map-Reduce

Map-Reduce is a scalable programming model [9]. It is abstracted such that only two main functions are written, Map and Reduce, where input and output are expressed in key-value object pairs. This model allows the programmer to simply focus on the logic through these two functions, and not worry about intra-node communication, task monitoring, or task-failure handling [9]. As well, both the map and reduce functions do not have to account for the size of data or even the underlying cluster that they are operating on.

Mapper (Map Phase) - The Hadoop framework reads input data from large files and split it into smaller portions (data blocks) which are then assigned to a mapper task. The data portions are read and wrapped into a key-value object pair and assigned to the map function. For an input of a key and a value, the function outputs (key, value) pair(s) which are then forwarded to the Reducers [3]. To enable fault tolerance, Mappers are deprived of any mechanisms to communicate with one another. Since data is replicated over nodes, many map tasks are activated in parallel, each processing different portions of the data that are stored locally on each node (Fig 2.2).

Reducer (Reduce Phase) - The output from the map functions is then sorted such that object values with the same key are grouped, assigned and transmitted to a reduce function. These grouped values are then processed by the Reduce functions and written back into the file system. (Fig 2.3, 2.4).

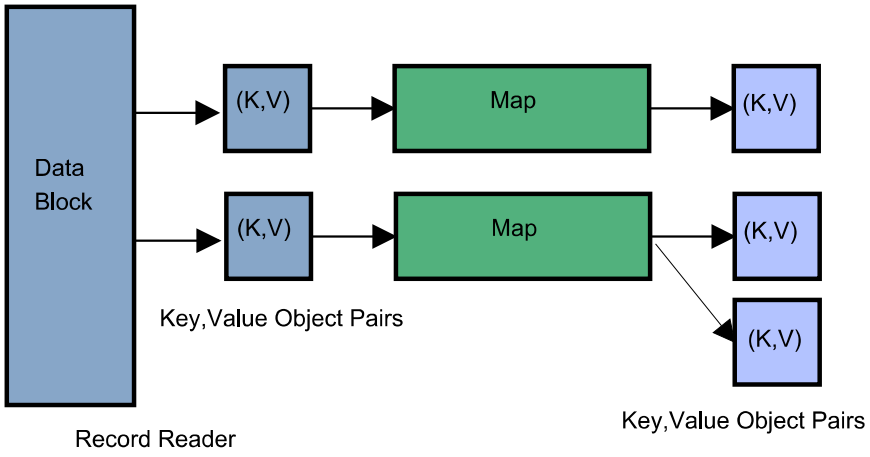

Figure 2.2: Map Phase.

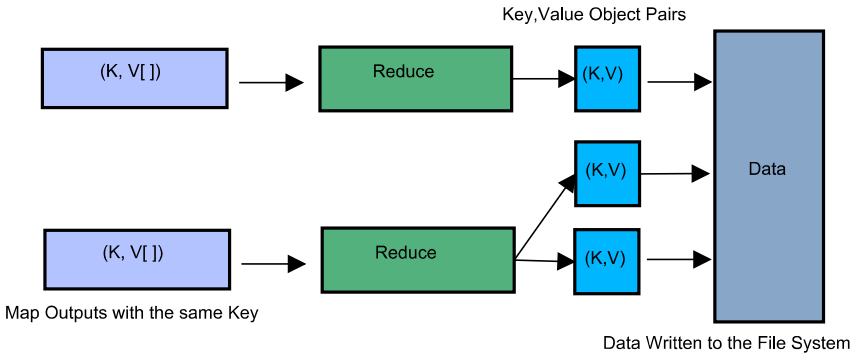

Figure 2.3: Reduce Phase.

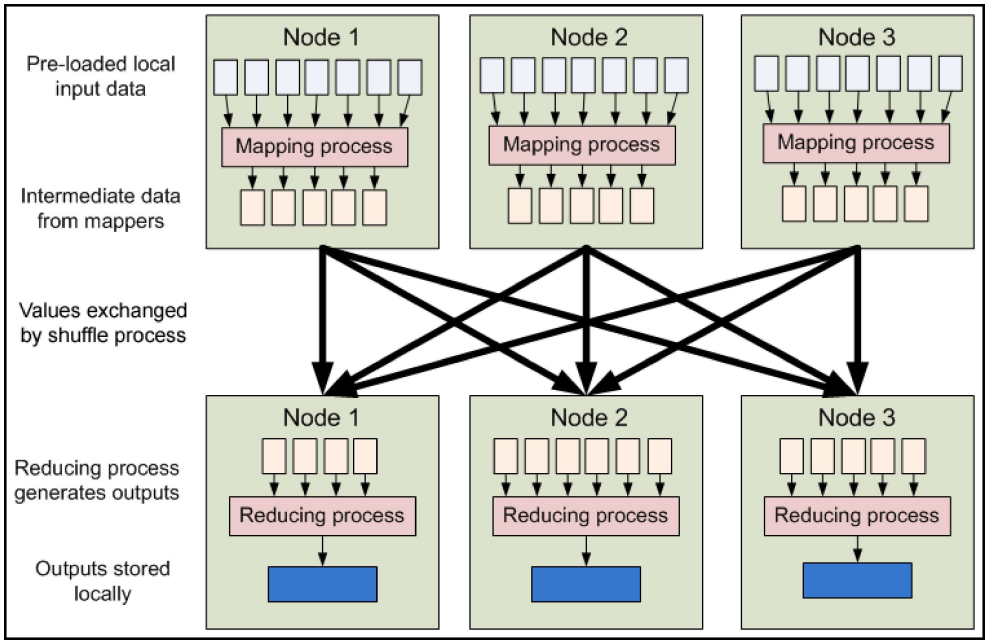

Figure 2.4: Map-Reduce Framework (from Yahoo Map-Reduce Tutorial).

Combiner - The combiner is an optional process that can be utilized after the Map and before the Reduce Phase. It is run on every node that has active map tasks. The outputs produced by the map functions on each node can first be sent to a local combine function. This function acts as a localized/mini reduce where local map outputs are grouped by their keys and assigned to it for processing. The combiner outputs is then sorted and transmitted to reducers. This is used to simplify map outputs, which can potentially decrease the transmission size or minimize the amount of calculations done by the reducer.

Map or Reduce tasks are executed in a fault-tolerant manner. Since there is no dependence between each activated map task or each activated reduce task, when one or many nodes fail during computation, the task is simply re-assigned to the remaining free healthy nodes.

2.2 Matrix-Vector Multiplication And Data Structures

When a matrix-vector multiplication is performed $v^{new} = M * v^{old}$ where $v^{old}, v^{new} \in \Re^c$ and $M \in \Re^{c \times c}$ and $1 \leq i \leq c$ and $1 \leq j \leq c$ we can express v_i^{new} as:

$$v_i^{new} = \sum_{j=1}^{c} q_{i,j}, \tag{2.1}$$

$$q_{i,j} = m_{i,j} * v_j^{old}. \tag{2.2}$$

We can view the component values $q_{i,j}$ as elements of a matrix, with the sum of the matrix values with the same row index i being the vector value v_i^{new}. During the process of multiplication, a vector element v_k will only be multiplied by matrix elements in column k. For example, v_1^{old} will only be multiplied by the matrix elements in the first column, resulting in component values $q_{1,1}, q_{2,1}, \ldots, q_{c,1}$. This association of a vector element with matrix column elements is important as it is the foundation of a method for performing matrix vector multiplication in Map-Reduce (Fig 2.5).

$$\begin{bmatrix} m_{1,1} & m_{1,2} & \cdots & m_{1,c} \\ m_{2,1} & m_{2,2} & \cdots & m_{2,c} \\ \vdots & \vdots & \cdots & \vdots \\ m_{r,1} & m_{r,2} & \cdots & m_{r,c} \end{bmatrix} \begin{bmatrix} v_1^{old} \\ v_2^{old} \\ \vdots \\ v_c^{new} \end{bmatrix} = \begin{bmatrix} q_{1,1} & + & q_{1,2} & + & \cdots & q_{1,c} \\ q_{2,1} & + & q_{2,2} & + & \cdots & q_{2,c} \\ \vdots & + & \vdots & + & \cdots & \vdots \\ q_{r,1} & + & q_{r,2} & + & \cdots & q_{r,c} \end{bmatrix} = \begin{bmatrix} v_1^{new} \\ v_2^{new} \\ \vdots \\ v_r^{new} \end{bmatrix}$$

Figure 2.5: Association between vector values and matrix column elements.

2.3 Data Structures

A way to represent and store both Matrix and Vector data for the use of Map-Reduce is to associate each nonzero matrix or vector elements with its coordinate. For a matrix, the coordinates are the row and column. Since a vector is a matrix of one column we can represent its element with the row as its coordinate. This structure helps the program to distinguish the element types by checking to see if a column coordinate is provided.

- Matrix element $m_{i,j} \Rightarrow m(row\ i, column\ j, value)$
- Vector element $v_k \Rightarrow v(row\ k, value)$

Treating the vector and matrix elements in such a way will have the added advantage of allowing map functions to process the two types of data at the same time.

Each matrix or vector element with its coordinate is a one line entry in the input file. A Record Reader function utilized by map tasks reads each line and transforms it to a key-value object. These objects are assigned as input to Map functions. Likewise, a Record Writer function is used by the reduce task to write output objects by the reduce functions as a line entry in the output file.

More elaborate structures are going to be utilized throughout this paper. However, they will essentially be containers that groups both matrix and vector elements together. These structures will also have corresponding Record Reader and Record Writer functions used by tasks to read and write to and from HDFS.

2.4 GIM-V Algorithm

GIM-V is a key algorithm for the Graph Mining Library PeGaSus that is implemented on the Hadoop platform. It is intended to scale for very large datasets [7]. This library performs many of the common graph mining operations ranging from diameter estimation to Page-Rank. Many of these operations can be composed by repeated matrix-vector multiplication where GIM-V is used.

The GIM-V approach uses two Map-Reduce stages to obtain the resulting vector v^{new}. Each stage has the following objective:

- **Stage-1** Read matrix and vector elements and calculate all nonzero components $q_{i,j}$.
- **Stage-2** Group and sum components $q_{i,1}, q_{i,2}, \ldots, q_{i,c}$ together based on their row index, i, to obtain the new vector value v_i^{new} for all i.

In Stage-1, the input data is a collection of matrix and vector elements. These elements are assigned to the Stage-1-Map function (Fig 2.6). The Stage-1-Map function directs the elements to the correct reducer through the output key. When a map function receives a vector value, it simply outputs the element with its row as the key $(v(row), v)$ and when it receives a matrix element, it outputs the element with the column as the key $(m(column), m)$.

When all map tasks have been completed, the outputs are grouped according to their key value. Outputs with the same key are then assigned to a reduce function. The Stage-1-Reduce function (Fig 2.7) receives a key k and a list of elements consisting of one vector

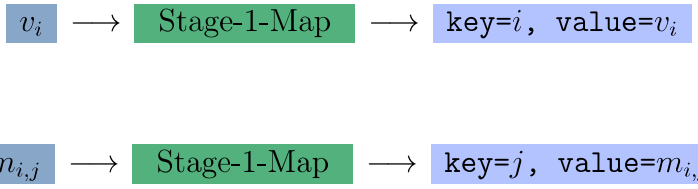

Figure 2.6: GIM-V Stage-1-Map function.

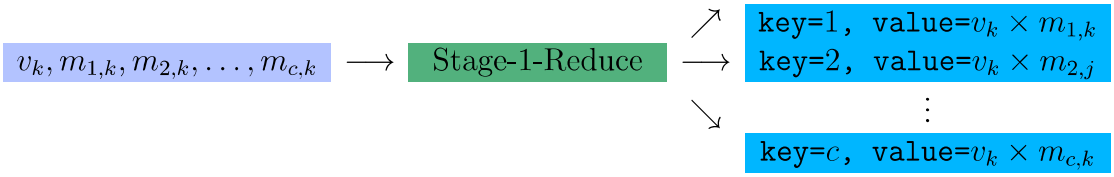

Figure 2.7: GIM-V Stage-1-Reduce function.

value v_k and all the nonzero matrix elements from column k. It then takes the vector value v_k from the element list and multiplies it with each of the matrix elements $m_{1,k}, \ldots, m_{c,k}$, obtaining the component values $q_{1,k}, \ldots, q_{c,k}$. The result of each product is then written back to the file system in the key-value pair form $(m(row), v_k \times m(value))$. To be optimal when outputting $q_{i,k}$, there is no point in storing the k index as we only need to know the row in order to calculate the new vector values.

There are cases where the reduce function does not return any output. These cases involve the fact that only nonzero elements are considered. When a reducer does not receive a vector element and-or matrix elements as input, then we know that all component values are 0 and there is no point to store them.

In Stage-2 (Fig 2.8), the component elements $q(row, value)$ are read back as input and are assigned to the Stage-2-Map function. This function does not perform any action except for simply outputting the assigned data with the row value as the key. The Stage-2-Reduce function will then receive row value k as key and a list of component values that it needs to sum over to obtain v_k^{new}.

Pseudocode for the two stages of GIM-V is given in Algorithms 1 and 2.

Algorithm 1 GIM-V Stage-1

Function Stage-1-Map(Key, Value)
Value is either an element from a Matrix → m(row,column,value)
or from a Vector → v(row,value)

```
if Value is a matrix element then
    m ← Value
    output(m(column), m)
else if Value is a vector element then
    v ← Value
    output(v(row), v)
end if
```

Function Stage-1-Reduce(Key, Values[])
Values contains Matrix elements → m(row,column,value)
and a Vector element → v(row,value)
Note: m(column) = v(row) = Key

```
v ← 0
matrixColumnElements ← []
for each element e in Values
    if e is a vector element then
        v ← e(value)
    else
        matrixColumnElements[end] ← e
    end if
end for
if v is not equal to 0 then
    for each matrix element m in matrixColumnElements
        q ← m(value) * v
        output(m(row), q)
    end for
end if
```

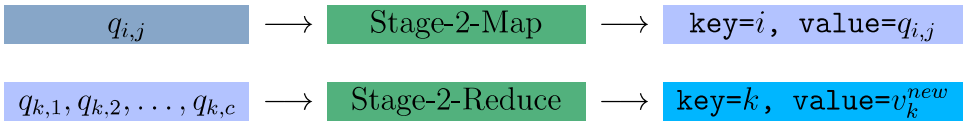

Figure 2.8: GIM-V Stage-2-Map, Stage-2-Reduce functions.

Algorithm 2 GIM-V Stage-2

Function Stage-2-Map(Key, Value)
Value is a component element → *q(row,value)*

$output(Value(row), Value(value))$

Function Stage-2-Reduce(Key, Values[])
Values contains component elements → *q(row,value)*
Note: The output will return the new vector value at row key

$vNew \leftarrow 0$
for each *element* **e in** $Values$
  $vNew \leftarrow vNew + e(value)$
end for
if $vNew$ not equal to 0 **then**
  $output(Key, vNew)$
end if

Let M be the total number of nonzero elements in the matrix, V_1 and V_2 be the total number of non zero elements in v^{old} and v^{new}, respectively, and Q be the total number of non zero components $q_{i,j}$. Note that if all vector elements are nonzero, then Q is the same as M. We have

$$Q = \sum_{j=1, v_j^{old} \neq 0}^{c} |M_{:,j}| = \sum_{j=1, v_j^{old} \neq 0}^{c} \sum_{i=1, m_{i,j} \neq 0}^{c} 1. \tag{2.3}$$

Therefore the total amount of data read from HDFS as input is $M + V_1 + Q$, where $(M + V_1)$ is for Stage-1 and Q is for Stage-2. The total amount of data written to HDFS is

	Stage-1	**Stage-2**	**Total**
Reads	$M + V_1$	Q	$M + Q + V_1$
Transfer	$M + V_1$	Q	$M + Q + V_1$
Writes	Q	V_2	$Q + V_2$

Table 2.1: GIM-V Data Read, Transfer and Write for one multiplication.

$Q + V_2$, where for Stage-1, Q components are produced by Stage-1-Reduce and V_2 elements are produced by Stage-2-Reduce. The maximum amount of data transferred between the map and reduce parts of each iteration is $M + Q + V_1$ where $(M + V_1)$ elements are passed in Stage-1 and Q elements are passed in Stage-2. (See Table 2.1)

Chapter 3

Single Stage Matrix Vector Multiplication (SSM-V)

When matrix vector multiplication is repeated many times with the same matrix, the GIM-V approach still reads the same matrix data and groups column elements with the associated vector value in each iteration. The knowledge that the matrix is constant can be used so that repeating Stage-1 for subsequent multiplications can be avoided. By removing Stage-1, we can then perform the multiplication in one stage.

In order to perform a multiplication in one stage, the ideal scenario would be to achieve the objective of GIM-V Stage-1 in the Map phase, and the objective of GIM-V Stage-2 in the Reduce phase. In other words, the map function should compute the $q_{i,j}$ component elements for each column of the matrix and the reduce adds the component values per row. This can be accomplished if the map function receives as input a vector value v_k and the corresponding matrix column elements $m_{1,k}, \ldots, m_{c,k}$ at the same time, and the Reduce function receives the components $q_{k,1}, \ldots, q_{k,c}$ along with matrix column elements $m_{1,k}, \ldots, m_{c,k}$ for the next iteration.

We believe that similar strategies may already be used to compute Page-Rank (see, e.g., [8]), but did not find descriptions in the published literature. In this algorithm the rank of an individual web page, represented as a vector value, is accompanied by all of its in-links, represented by matrix column elements. By grouping these values together, a single iteration of the Page-Rank algorithm can be performed in one Map-Reduce stage.

3.1 SSM-V Preprocessing

In SSM-V, a preprocessing step is first needed to structure the data, such that for every row k, a vector element v_k is paired with the k^{th} column elements of the matrix. This is very similar to GIM-V Stage-1 except that the Reducer simply structures and outputs the data to the file system rather than calculating the component elements $q_{1,k}, q_{2,k}, \ldots, q_{c,k}$. The format of the output is [k, v_k, $(1, m_{1,k}(value))$, $\ldots$,$(c, m_{c,k}(value))$] and is a one line entry in the output file (note that only nonzero matrix elements $m_{1,k}$ are included in the list). This structure is the expected input by the map function for each subsequent multiplication.

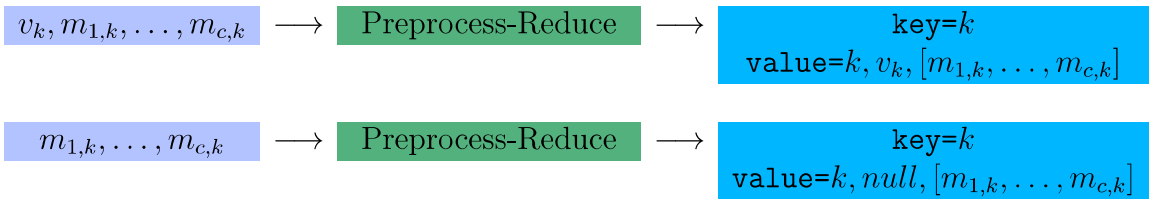

Figure 3.1: SSM-V Preprocessing Reduce function.

Recall that GIM-V Stage-1-Reduce does not output data when the vector value v_j is 0. If the current v_j is 0, then matrix column elements are not needed in the upcoming iteration. However, future values of v_j may be nonzero at which point the j^{th} column elements will be needed. Therefore the reduce function in the preprocessing and multiplication stage will always output data records as long as there are column elements in the matrix (Fig 3.1).

See Algorithm 3 for pseudocode of the SSM-V preprocessing stage.

3.2 SSM-V Multiplication Stage

Given the input [k, v_k, $(1, m_{1,k}(value))$, $\ldots$,$(c, m_{c,k}(value))$], the SSMVMap function multiplies the vector value, v_k, with each matrix element, outputting $(i, q_{i,k})$. The function also outputs the list of column elements, $[(1, m_{1,k}(value)), \ldots, (c, m_{c,k}(value))]$, with column k as the key (Fig 3.2).

The SSMVReduce function will receive a row value k as key, a list of component values $q_{k,1}, q_{k,2}, \ldots, q_{k,c}$ needed to obtain v_k^{new} and the associated column k elements of the matrix. It proceeds to calculate the new vector value and outputs [k, v_k^{new}, $(1, m_{1,k}(value))$,

Algorithm 3 SSM-V Preprocessing Stage

Function Preprocess-Map(Key, Value)
Value is either an element from a Matrix → m(row,column,value)
or a Vector → v(row,value)

```
if Value is a matrix element then
    m ← Value
    output(m(column), m)
else if Value is a vector element then
    v ← Value
    output(v(row), v)
end if
```

Function Preprocess-Reduce(Key, Values[])
Values contains Matrix elements → m(row,column,value)
and Vector element → v(row,value)
Note: m(column) = v(row) = Key

```
v ← 0
r ← Key
matrixColumnElements ← []
for each element e in Values
    if e is a vector element then
        v ← e(value)
    else
        matrixColumnElements[end] ← (e(row), e(value))
    end if
end for
output(Key, [r, v, matrixColumnElements])
```

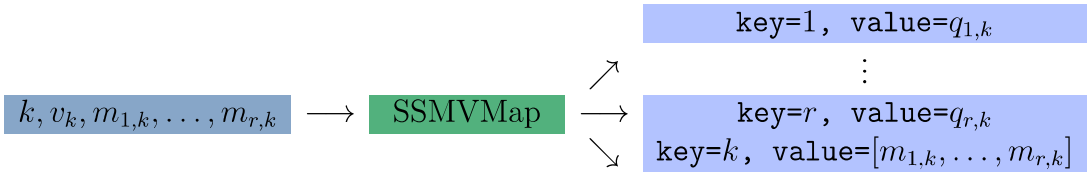

Figure 3.2: SSMVMap function.

$\ldots,(c, m_{c,k}(value))]$. This allows the new vector value to be paired with the list of column elements that is needed for the next iteration (Fig 3.3). The pseudocode for the SSM-V multiplication stage is given in Algorithm 4.

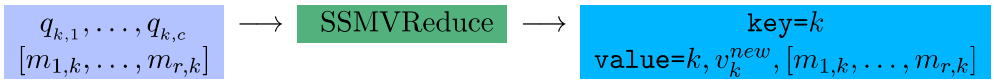

Figure 3.3: SSMVReduce function.

Since the preprocessing step is only executed once, the fixed cost is $M + V_1$ data reads, $M + V_1$ data transfers and $M + V_1$ data writes. For the multiplication stage the total amount of data read is $M + V_1$. The total data written in the file system is $M + V_2$. The total amount of data transferred between nodes is $M + Q$ where Q components and M matrix elements are passed by the Map function. (See Table 3.1)

	Preprocessing	**Multiplication**
Reads	$M + V_1$	$M + V_1$
Transfer	$M + V_1$	$M + Q$
Writes	$M + V_1$	$M + V_2$

Table 3.1: SSM-V Data Read, Transfer and Write for one multiplication

From the initial analysis we can see that for one multiplication the amount of data being read, transferred and written by this algorithm is comparable to the GIM-V method.

Algorithm 4 SSM-V Multiplication

Function SSMVMap(Key, Value)
Value is composed of [row j, vector value v_j, nonzero j^{th} matrix column elements]

```
v ← Value(vector value)
if v is not 0 then
  for each matrix element m in Value(MatrixElements)
    q ← m(value) * v
    output(m(row), q)
  end for
end if
output(Value(row), Value(matrixElements))
```

Function SSMVReduce(Key, Values[])
Values is a list of components e(row, value) and a set of Matrix Column Elements

```
vNew ← 0
r ← Key
matrixColumnElements ← []
for each element x in Values
  if x is a component element then
    vNew ← vNew + x(value)
  else
    matrixColumnElements ← x
  end if
end for
output(Key, [r, vNew, matrixColumnElements])
```

Chapter 4

SSM-V Optimal

The SSM-V algorithm discussed above, though it can be performed in one stage, is not an optimal algorithm to perform multiplication in Map-Reduce. Two improvements can be made resulting in faster performance and efficient transfer of data between nodes. These improvements utilize and exploit existing functionality offered by the Hadoop framework.

4.1 SSM-V Optimization by splitting

One of the major disadvantages that the initial SSM-V algorithm has is not taking full advantage of the Map-Reduce concept. By grouping vector and column elements together as input to the map function it essentially forces one map function to handle one v_k and $m_{1,k}, \ldots, m_{c,k}$. This does not scale very well on large matrix data, or, more specifically, on matrix columns with dense elements. If there exists one dense column in an otherwise sparse matrix, the whole map phase would not complete until this column is fully processed, which may unnecessarily slow down the whole multiplication step. In this case, it may be more suitable to have many map functions each performing part of the work.

To resolve this problem we can split the columns into smaller manageable chunks and give each chunk a copy of the vector value. This way many map functions can be used at once. This also has the added advantage of ensuring that map functions throughout the map phase are assigned a consistent input size. To make this change, both the reduce functions in the preprocessing and multiplication steps need to be altered (see Figures, 4.1 and 4.2). At the cost of replicating vector values, this modification may significantly improve the speed of the algorithm since it is using the full potential of the framework.

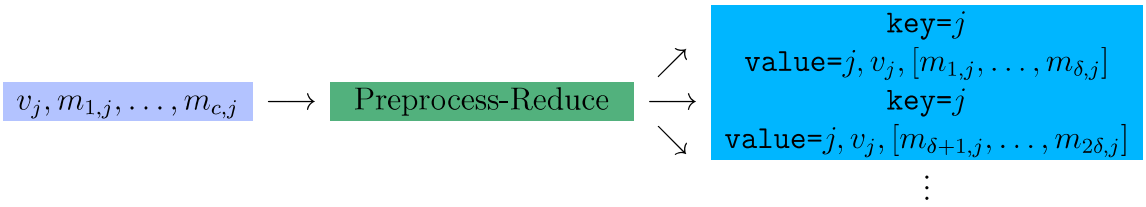

Figure 4.1: SSM-V Preprocessing Reduce function (where δ is the split size).

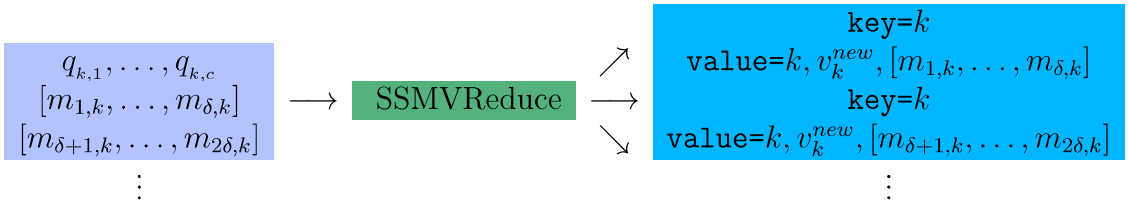

Figure 4.2: SSMVReduce function (where δ is the split size).

Due to replicating vector values, the SSM-V method would then write more data than the GIM-V method. Where it would have initially written $M + V$ values it will now write $M + \alpha V$ with $\alpha \geq 1$, and

$$\alpha * V = \sum_{i=1, v_i \neq 0}^{c} \lceil \frac{|M_{:,i}|}{\delta} \rceil, \tag{4.1}$$

where δ is the user-defined chunk size and $|m_{:,i}|$ is the total nonzero elements in the i^{th} column of the matrix. See Table 4.1 for a comparison.

	SSM-V Multiplication	**GIM-V Stage-1**	**GIM-V Stage-2**
Write	$M + \alpha V_2$	Q	V_2

Table 4.1: Data Write Comparison between SSM-V and GIM-V.

In the case of a sparse matrix and a full element vector, the difference would not be significant.

See Algorithms 5 and 6 for pseudocode with this optimization.

Algorithm 5 Optimized SSM-V (Preprocessing) Reduce function

Function Preprocess-Reduce(Key, Values[])
Values contains Matrix Elements → m(row,column,value)
and Vector Element → v(row,value)
Note: m(column) = v(row) = Key, and chunkSize is a fixed size defined by the user

```
v ← 0
r ← Key
δ ← chunkSize
matrixColumnElements ← []
for each element e in Values
  if e is a vector element then
    v ← e(value)
  else
    matrixColumnElements[end] ← (e(row), e(value))
  end if
end for
while matrixColumnElements is not empty do
  list ← remove δ elements from matrixColumnElements
  output(Key, [r, v, list])
end while
```

Algorithm 6 Optimized SSM-V (Multiplication) Reduce function

Function SSMVReduce(Key, Values[])
Values is a list of components e(row, value) and a set of Matrix Column Elements
Note: The output will return the new vector value at row key

```
vNew ← 0
r ← Key
matrixColumnElements[] ← []
for each element x in Values
  if x is a component element then
    vNew ← vNew + x(value)
  else
    matrixColumnElements[end] ← x
  end if
end for
for each matrix Column Elements L in matrixColumnElements
  output(Key, [r, vNew, L])
end for
```

4.2 Optimization through Combiner Function

Recall that a combiner function is essentially a localized reducer activated on all nodes that have executed map functions. As the intermediate step that sits between map and reduce, this function can be used to optimize both SSM-V and GIM-V methods.

Using combine functions we can potentially decrease the amount of data transfer between nodes and decrease the number of operations performed by the reducer. In particular the addition of component values $\Sigma_{k=1}^{c} q_{r,k}$ done by the Reducer to obtain v_j^{new} can partially be done on the combiner. This will then reduce the amount of component values sent over the network, and decrease the number of addition operations performed by the reducer. For example if a node with active map functions, for one input data block with multiple input records that are fed to the same map node, collectively outputs x component values where there are y unique keys then a combiner can do partial sums on values with the same key. This in effect decreases the data transfer from what would have been x values to y values.

For the GIM-V approach a combiner function can be applied in the second stage. In fact the combiner function will be the same as the Stage-2-Reduce function. A Combiner function can also be placed after the map phase of the multiplication step of SSM-V, see Fig 4.3 and Algorithm 7.

Figure 4.3: SSMVCombine function.

4.3 Absorbed Preprocessing

The preprocessing step is a fixed cost in the SSM-V algorithm. However there is a condition where the cost can be nullified. This condition is dependent entirely on the values of the initial vector. If this condition is satisfied, not only can we decrease the amount of data read by the map tasks of the first multiplication step, but we can also absorb the preprocessing step. This may not be a common scenario, however if does apply, the following optimization may be used. The condition is:

Algorithm 7 SSM-V Combiner Function

Function SSMVCombine(Key, Values[])

Values is a list of components e(row, value) and a set of Matrix Column Elements with a common key, generated by a shuffling process on the local node

```
vPartialNew ← 0
r ← Key
for each element x in Values
  if x is a component element then
    vPartialNew ← vPartialNew + x(value)
  else
    output(r, x)
  end if
end for
output(r, vPartialNew)
```

- Values of the initial vector can be determined by a function, in such a way that, given any matrix element $m_{i,k}$ and its coordinate (i, k), the function can produce the appropriate vector value v_k.

If this condition in satisfied, only matrix elements are needed as input for the first multiplication. Since a Map function, in itself, can produce the appropriate vector value v_k, the function can calculate the component values $q_{i,k}$ right away. So given a matrix element $m_{i,k}$, the map function will output two data items, the component value with the row index as key, and the matrix value with the column index as key. Then the reduce function will receive component values $q_{k,1}, q_{k,2}, \ldots, q_{k,c}$ and matrix column elements $m_{1,k}, \ldots, m_{c,k}$. With these data the reducer then calculates v_k^{new} and pairs it with the column elements for the next iteration, see Figures 4.4 and 4.5 and Algorithm 8. The general SSM-V method is then applied for subsequent multiplications.

Since we do not need to read the initial vector value, the total data that is read is M with $M + Q$ data transfer and $M + V_2$ data writes for the first multiplication.

Algorithm 8 SSM-V Incorporated Preprocessing Stage

Function SSMVFirstMap(Key, Value)
Value is an element from a Matrix → m(row,column,value)
Note: f is a function used to derive the initial vector value

$v \leftarrow f(m(row))$
$output(m(row), v * m(value))$
$output(m(column), m)$

Function SSMVFirstCombine(Key, Values[])
Values contains Matrix → m(row,column,value) and Component → q(row,value) elements
$vPartialNew \leftarrow 0$
for each *element* **x in** $Values$
if x is a component element **then**
$vPartialNew \leftarrow vPartialNew + x(value)$
else
$output(r, x)$
end if
end for
$output(r, vPartialNew)$

Function SSMVFirstReduce(Key, Values[])
Values contains Matrix → m(row,column,value) and Component → q(row,value) elements
Note: m(column) = v(row) = Key
$vNew \leftarrow 0$
$r \leftarrow Key$
$\delta \leftarrow chunkSize$
$matrixColumnElements \leftarrow []$
for each *element* **e in** $Values$
if e is a component element **then**
$vNew \leftarrow vNew + e(value)$
else
$matrixColumnElements[end] \leftarrow (e(row), e(value))$
end if
end for
while matrixColumnElements is not empty **do**
$list \leftarrow$ remove δ elements from $matrixColumnElements$
$output(Key, [r, vNew, list])$
end while

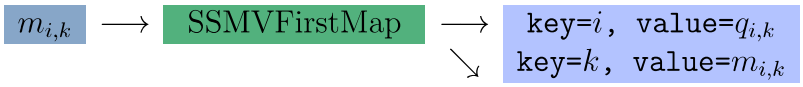

Figure 4.4: SSMVFirstMap function in the first Multiplication step, absorbing the Preprocessing stage.

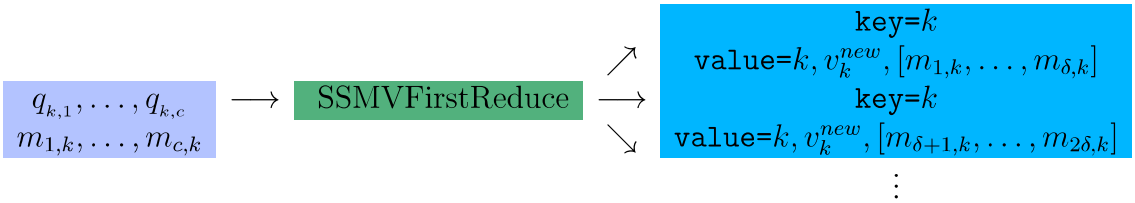

Figure 4.5: SSMVFirstReduce function in the first Multiplication step, absorbing the Preprocessing stage. Here, δ is the chunk size.

4.4 SSM-V with Direct Reads and Writes to HDFS

All of the algorithms discussed so far have relied on the Hadoop framework to efficiently divide and assign the input data through Map Tasks and to write output data through Reduce Tasks. As a result, both map and reduce functions do not directly interact with the data in HDFS. However, these functions can directly read and write data in the file system. This ability combined with the framework can be used to modify the SSM-V algorithm to minimize the total data reads and writes. In particular the matrix data $m_{i,j}$ always remains the same, and does not necessarily have to be sent along with the other data in all steps of the iterations.

Direct interaction with HDFS can be used to make the framework process the matrix data and to rely on the map and the reduce function to retrieve and write vector values to and from the file system. First consider a preprocessing stage in which matrix elements with the same column index are grouped in chunks, and each chunk is a line entry in a new large input file. This file will be used as input for each subsequent multiplication stage. Vector values are stored in individual files with the row index as their name. In the multiplication iterations, the framework assigns a set of matrix column elements, $m_{i,k}, \ldots, m_{i+\delta,k}$, to a map function. The map function with the knowledge of the column index k retrieves the value v_k^{old} from HDFS and outputs the component values $q_{i,k}, \ldots, q_{i+\delta,k}$ with the row index

as key (Fig 4.6). A reducer will be assigned all of the component values, $q_{k,1}, \ldots, q_{k,c}$, with same row index k, which is then used to obtain the new vector value, v_k^{new}. The reduce function then overwrites the value in the file with the row index as the name (Fig 4.7).

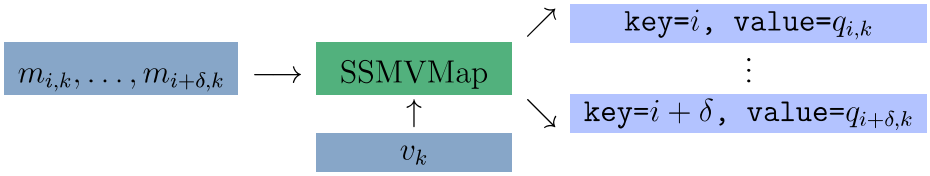

Figure 4.6: SSMVMap function with v_k read directly from HDFS.

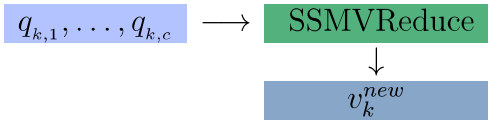

Figure 4.7: SSMVReduce function with v_k^{new} written directly to HDFS.

With this approach a single multiplication only requires M Hadoop reads, Q data transfers and no Hadoop data writes, which is an improvement over the original $M + \alpha V_1$ Hadoop reads $M + Q$ data transfers and $M + \alpha V_2$ data writes, see Table 3.1. Note, however that this strategy adds αV_1 HDFS reads and αV_2 HDFS writes, with each read and write accessing a single small file.

Note that this modification does not compromise the fault tolerance offered by the framework. There are no dependency issues in the map phase since all of the vector values, v_k^{old}, are already written in files before the phase begins. Also, since each row of a vector is only assigned to one reduce function, there will never be a case where two or more functions are writing to the same file.

Unfortunately, in implementation, we do not observe faster performance. In fact, performance becomes very slow as the matrix dimension increases. This problem occurs because HDFS is not designed to efficiently store and access small files. It is primarily designed for streaming access of large files [10, 2]. Every file in HDFS, no matter the size, consumes a fixed amount of memory that is used by the file system to manage it [10]. As we increase the dimensions, more memory is required to maintain all of these small files. Also, reading

through small files usually results in many seeks and jumping from one node to another to retrieve each small file, all of which results in an inefficient data access pattern [10].

If future improvements can be made to HDFS such that it can efficiently handle storing and accessing small files, this modified SSM-V would offer interesting options. An alternative is to store v_k values in HBase (Hadoop Database), which should have better read and write performance.

Chapter 5

Performance Experiments

For testing, Hadoop was set up on a loosely coupled cluster of three servers connected together with Gigabit Ethernet. Large sparse matrix data was generated ranging from 2.5 to 80 million nonzero elements and vectors with dense elements. Random sparse matrices with dimensions 1000 x 1000 and varied densities was initially generated in Matlab. These matrices were then used to create large test matrices by replicating them as blocks over larger dimensions. Several tests were performed initially comparing how the suggested optimizations affect the overall performance of SSM-V, followed by a comparison to GIM-V.

To ensure that there are enough resources, each server is configured to have a maximum of 4 active map tasks and 4 active reduce tasks (each server has 4 or more cores). In the file system, a data block is defined to be 64 megabytes. The largest test data used is larger than 550 megabytes. Since a single data block is assigned to a single map task, there are sufficient resources to process all the test data in parallel and map tasks don't wait in the Hadoop queue.

5.1 Effects of Optimization

Two optimizations were applied to the initial SSM-V method. A matrix with dimension 100000×100000 composed of roughly 80 million nonzero floating point elements was used to test the alterations. In total more than 550 megabytes of data was used as input.

As expected, both optimizations, when applied, improved the performance. The one that caused a significant improvement in speed was the chunking of matrix column ele-

Num	Optimization	Run Time
1	NULL	25 Minutes
2	Chunk Column elements to a fixed size Use Combiner Functions	2 Minutes

Table 5.1: Effect of Optimization on a problem with more than 80 Million nonzero Records.

ments. This allowed many map functions to participate in processing a vector and its corresponding matrix column elements. Also, map functions in general will always be assigned inputs whose size is consistent. Without chunking, map functions can have varied input sizes and if one function is assigned a large input it can slow down the map phase. The use of combiner functions only improved the timing by a few seconds. However, it's still a desirable approach as it tries to minimize the data transfer between nodes. By making full use of the Map, Combine and Reduce functions provided by the framework a 10 fold speed reduction was observed. (See Table 5.1)

5.2 Results of SSM-V with direct interaction to HDFS

The modified SSM-V algorithm which directly reads and writes data in HDFS in theory should offer good results. A matrix with dimension 50000×50000 composing of roughly 2.5 million nonzero floating point elements was used to compare this algorithm with the optimized SSM-V. Though the Hadoop data transfers and writes are smaller, it does not translate in faster speeds. By shifting part of the reading and writing of data originally performed by the framework to the functions, a significant deterioration was observed. (See Table 5.2)

NNZ elements	SSM-V Optimal	SSM-V with direct interaction to HDFS
2.5×10^6	35 Seconds	19 Minutes

Table 5.2: Comparison between optimized SSM-V and direct interaction to HDFS. A dataset of more than 2.5 Million nonzero Records was used.

5.3 Comparison between GIM-V and SSM-V

The SSM-V method offers a single stage matrix-vector multiplication unlike the 2 stage GIM-V. However there are 2 tradeoffs to consider:

- The fixed preprocessing cost
- Increase of data writes (See Tables 2.1, 3.1 and 4.1)

We now investigate these tradeoffs. Several different matrix and vector test data were tried and compared. The SSM-V method generally performed better per iteration. Over multiple iterations and with the addition of the preprocessing, the SSM-V still performs well. (See Tables 5.3, 5.4 and 5.5)

Num	NNZ elements	Run Time
1	2.5×10^6	33 seconds
2	5.0×10^6	42 seconds
3	1.0×10^7	1:09 minutes
4	4.0×10^7	1:23 minutes
5	8.0×10^7	1:43 minutes

Table 5.3: Preprocessing Cost for SSM-V.

Num	NNZ elements	GIM-V (for both stages)	SSM-V (without preprocessing)
1	2.5×10^6	1:07 minutes	35 seconds
2	5.0×10^6	1:16 minutes	44 seconds
3	1.0×10^7	1:43 minutes	54 seconds
4	4.0×10^7	2:02 minutes	1:21 minutes
5	8.0×10^7	2:37 minutes	2:03 minutes

Table 5.4: Comparison for 1 multiplication between GIM-V and SSM-V.

Num	NNZ elements	GIM-V	SSM-V (with preprocessing)
1	2.5×10^6	5:36 minutes	3:48 minutes
2	5.0×10^6	6:20 minutes	4:40 minutes
3	1.0×10^7	8:59 minutes	5:43 minutes
4	4.0×10^7	10:21 minutes	8:05 minutes
5	8.0×10^7	13:05 minutes	11:59 minutes

Table 5.5: Comparison between GIM-V and SSM-V, for 5 multiplication iterations.

For one multiplication the GIM-V method would read and transfer more data. On the other hand, the SSM-V method performs more data writes. In HDFS, writing to the file

system requires replicating information among nodes that is handled by the file system. There are 2 reasons why the SSM-V method writes more data (see Tables 5.6 and 5.7) :

- The replication of vector values to accommodate chunking
- The writing of column elements, even if the current associated vector value is 0

	GIM-V	**SSM-V**	**GIM-V - SSM-V**
Reads	$M + V_i + Q_i$	$M + \alpha V_i$	$Q_i - (\alpha - 1)V_i$
Transfer	$M + V_i + Q_i^{opt}$	$M + Q_i^{opt}$	V_i
Writes	$Q_i + V_{i+1}$	$M + \alpha * V_{i+1}$	$Q_i - M - (\alpha - 1)V_{i+1}$

Table 5.6: Comparing Data Read, Transfer and Write between GIM-V and SSM-V.

Per iteration, the SSM-V writes an extra $(\alpha - 1)V_{i+1}$, due to replication, and $M - Q_i$, due to column elements and its usage in future iterations. If we anticipate the majority of values in the vector to be nonzero, then Q_i is close to M, and the difference is small. In fact, if in each iteration the vector elements are all nonzero, Q_i is equal to M and the extra writes are attributed only to replication. On the other hand, if there are many zero entries in the vector the SSM-V method may not be an efficient method: for every row k in the vector whose value is zero, there are $|M_{:,c}|$ less component values, $q_{1,k}, \ldots, q_{c,k}$, written by GIM-V (see Table 5.7).

Tables 5.8 and 5.9 give a detailed comparison of number of bytes read and written.

	GIM-V	**SSM-V**	**GIM-V - SSM-V**
Reads	$2M + V_i$	$M + \alpha V_i$	$M - (\alpha - 1)V_i$
Transfer	$M + V_i + M_i^{opt}$	$M + M_i^{opt}$	V_i
Writes	$M + V_{i+1}$	$M + \alpha * V_{i+1}$	$(1 - \alpha)V_{i+1}$

Table 5.7: Comparing Data Read, Transfer and Write between GIM-V and SSM-V with the assumption that the vector is dense.

		Time	**Map Task**	**Reduce Task**	**Bytes Read**	**Bytes Write**
GIM-V	Stage-1	1:40	9	10	515548046	509839300
	Stage-2	0:57	9	9	509839300	1429966
	Total	**2:37**	**18**	**19**	**1025387346**	**511269266**
SSM-V	Preprocess	**1:43**	9	10	515548046	588733160
	Multiplication	**2:03**	**9**	**10**	**588733160**	**626040780**

Table 5.8: Comparing Data Read, Transfer and Write between GIM-V and SSM-V on a test problem with 80 Million nonzero matrix elements, and all vector elements are nonzero.

		Time	**Bytes Write**
GIM-V	Stage-1	1:35	452839300
	Stage-2	0:47	1251855
	Total	**2:22**	**454091155**
SSM-V	**Multiplication**	**1:98**	**623976380**

Table 5.9: Comparing Data Read, Transfer and Write between GIM-V and SSM-V on a test problem with 80 Million nonzero matrix elements and a vector which has 20 percent zero entries.

Chapter 6

Extension to Jacobi Method

The Jacobi method is a well known iterative algorithm for matrix systems $Mx = b$ that can easily be parallelized. In fact, both the GIM-V and the SSM-V can naturally be extended to perform this calculation.

For a square system of n linear equations $Mx = b$, we can decompose the matrix M as $M = D + R$, with

$$D = \begin{bmatrix} m_{1,1} & 0 & 0 \\ 0 & m_{i,i} & 0 \\ 0 & 0 & m_{c,c} \end{bmatrix}, R = \begin{bmatrix} 0 & m_{1,2} & & m_{1,c} \\ m_{2,1} & 0 & & m_{2,c} \\ & & 0 & \\ m_{c-1,1} & & 0 & m_{c-1,c} \\ m_{c,1} & & m_{c,i} & 0 \end{bmatrix}. \tag{6.1}$$

We can then rewrite the equation as $Dx + Rx = b$ and obtain an iterative method as follows:

$$x^{k+1} = D^{-1}(b - Rx^k), \tag{6.2}$$

$$x_i^{k+1} = \frac{1}{m_{i,i}}(b_i - \Sigma_{j \neq i} m_{i,j} x_j^k), \tag{6.3}$$

$$x_i^{new} = \frac{1}{m_{i,i}}(b_i - \Sigma_{j \neq i} q_{i,j}). \tag{6.4}$$

6.1 General Approach

The main objectives used to derive Matrix-Vector multiplication can slightly be adjusted to accommodate the Jacobi method. These new objectives are:

- Compute all components $q_{i,j}$ where $i \neq j$
- Group Component elements $q_{i,1}, .., q_{i,i-1}, q_{i,i+1}, .., q_{i,c}$ along with the associated diagonal element $m_{i,i}$ and an element b_i from the b vector to calculate x_i^{new}.

6.2 Jacobi via GIM-V

Once again, Stage-1 is used to read and group vector and column elements together. However, in this case, it will only calculate non diagonal components. The component values and the diagonal elements are written to the file system. There is very little change needed for Stage-1, in fact we can reuse the Stage-1-Map function and make slight adjustments to Stage-1-Reduce.

In Stage-2 (see Figures 6.1 and 6.2), the map functions are assigned data that was generated from the previous stage as well as the vector b elements. If the Stage-2-Map function receives a component value $q_{i,j}$, it will output $-q_{i,j}$, while if it receives an element b_i or $m_{i,i}$, it will simply output the value. In all cases, the row value i will be the output key. By reversing component values in the map stage, we allow the reducer to simply sum over these values. Stage-2-Reduce will then receive all the necessary values it needs, $-q_{i,1}, \ldots, -q_{i,i-1}, -q_{i,i+1}, \ldots, -q_{i,c}, b_i, m_{i,i}$, to obtain v_i^{new}. We can also apply a combine function in this stage to perform partial sums of non-diagonal components and vector b elements (see Algorithms 9 - 12).

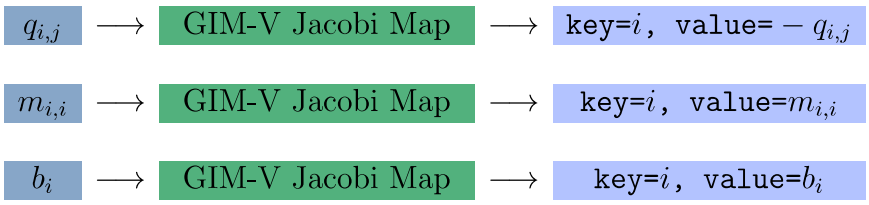

Figure 6.1: Stage-2, GIM-V Stage-2-Map function extended for Jacobi.

Figure 6.2: GIM-V Stage-2-Reduce function extended for Jacobi.

Algorithm 9 Jacobi via GIM-V Stage-1

Note: the Stage-1-Map function does not change

Function Stage-1-Reduce(Key, Values[])
Values contains Matrix Elements → m(row,column,value)
and a Vector Element → v(row,value)

```
v ← 0
matrixColumnElements ← []
for each element e in Values
  if e is a vector element then
    v ← e(value)
  else
    matrixColumnElements[end] ← e
  end if
end for
if v is not equal to 0 then
  for each matrix element m in matrixColumnElements
    if m(row) = m(column) then
      output(m(row), m)
    else
      q ← m(value) * v
      output(m(row), q)
    end if
  end for
end if
```

Algorithm 10 Jacobi via GIM-V Stage-2 (Map Phase)

Function Stage-2-Map(Key, Value)
Value is either a Component Element → e(row,value) or
Diagonal Element → d(row,column,value) or element from vector → b(row,value)

if $Value$ is either a diagonal element or element of b vector **then**
  $output(Value(row), Value)$
else
  $output(Value(row), -1 * Value)$
end if

Algorithm 11 Jacobi via GIM-V Stage-2 Combiner

Function Combine(Key, Values[])
Value is either a Component Element → e(row,value) or
Diagonal Element → d(row,column,value) or element from vector → b(row,value)

$vPartialNew \leftarrow 0$
for each *element* **e in** $Values$
  if e is a diagonal element **then**
    $output(key, e)$
  else
    $vPartialNew \leftarrow vNew + e(value)$
  end if
end for
if $vPartialNew$ not equal to 0 **then**
  $output(key, vPartialNew)$
end if

Algorithm 12 Jacobi via GIM-V Stage-2 (Reduce Phase)

Function Stage-2-Reduce(Key, Values[])
Values contains component Elements → e(row,value),
Diagonal Element → d(row,column,value) and an element from vector → b(row,value)

```
vNew ← 0
diagValue ← 0
for each element e in Values
  if e is a diagonal element then
    diagValue ← e(value)
  else
    vNew ← vNew + e(value)
  end if
end for
vNew ← vNew/diagValue
if vNew not equal to 0 then
  output(key, vNew)
end if
```

6.3 Jacobi via SSM-V

Similarly, SSM-V can be altered to calculate the Jacobi method by following many of the same changes done for GIM-V Stage-2. Only the Multiplication stage needs to be altered. The SSMVMap function will receive two types of data structures as input. One type is the chunked vector-matrix column elements list $[j,\ v_j,\ (i, m_{i,j}(value)),\ ,(i+\delta, m_{i+\delta,j}(value))]$, which is used to calculate the non-diagonal components $q_{i,j}$ and outputs the reverse $-q_{i,j}$ and the matrix column list. The other type is an element b_i from the b vector, in which case it simply outputs the value. Again, in all cases, the row index is used as the output key.

Then each SSMVReduce function in general receives the element b_k, component values $-q_{k,1}, .., -q_{k,k-1}, -q_{k,k+1}, .., -q_{k,c}$ and column elements $m_{1,k}, \ldots, m_{c,k}$. The function then extracts the diagonal element, $m_{k,k}$, from the column list and evaluates the v_k^{new}. The new vector value is then written to output accompanied with the column list for the next iteration. (See Figures 6.3 and 6.4, and Algorithms 13, 14 and 15).

Comparison tests were not executed, but performance is expected to be similar to matrix-vector product performance.

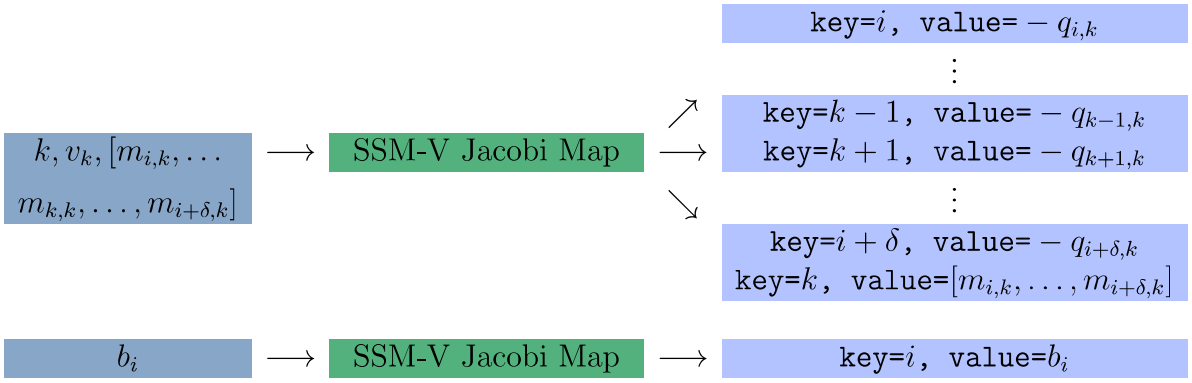

Figure 6.3: SSMVMap function extended for Jacobi.

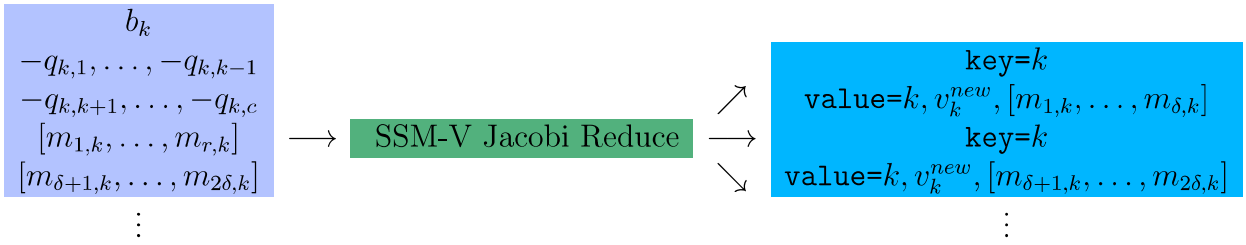

Figure 6.4: SSMVReduce function extended for Jacobi (where δ is the split size).

Algorithm 13 Jacobi via SSM-V Multiplication (Map Phase)

Function SSMV-Jacobi-Map(Key, Value)
Value is either [row j, vector value v_j, j^{th} matrix column elements]
or element from vector b(row,value)

```
if Value is an element of b vector then
  output(Value(row), Value)
else
  r ← Value(row)
  v ← Value(vector value)
  for each matrix element m in Value(MatrixElements)
    if m(column) ≠ r then
      q ← m(value) * v
      output(m(row), −1 * q)
    end if
  end for
  output(Value(row), Value(matrixElements))
end if
```

Algorithm 14 Jacobi via SSM-V Combiner

Function SSMV-Jacobi-Combine(Key, Values[])
Values is a list of components q(row, value), an element of b b(row, value)
and Matrix Column Elements

```
vPartialNew ← 0
r ← Key
for each element x in Values
  if x is a component element or vector b element then
    vPartialNew ← vPartialNew + x(value)
  else
    output(r, x)
  end if
end for
output(r, vPartialNew)
```

Algorithm 15 Jacobi via SSM-V Multiplication (Reduce Phase)

Function SSMV-Jacobi-Reduce(Key, Values[])
Values is a list of components q(row, value), an element of b b(row, value) and Matrix Column Elements
Note: The output will return the new vector value at row key

```
vNew ← 0
bValue ← 0
diagValue ← 0
r ← Key
matrixColumnElements ← []
for each element x in Values
  if x is a component element or vector b element then
    vNew ← vNew + x(value)
  else
    matrixColumnElements[end] ← x
    if x has a diagonal element then
      diagValue ← x(diagonalValue)
    end if
  end if
end for
vNew ← vNew/diagValue
for each matrix Column Elements L in matrixColumnElements
  output(Key, [r, vNew, L])
end for
```

Chapter 7

Conclusions

The SSM-V approach offers an alternative to the two stage GIM-V method for calculating Matrix-Vector multiplication over many iterations. Even though it can be performed in a single Map-Reduce phase, and the overall data read and transfer rates are attractive, it does require writing more data, and a fixed preprocessing cost. The SSM-V algorithm with its optimization provides an interesting alternative that can be applied not only to Matrix-Vector multiplication but also to the Jacobi Method.

References

[1] Hadoop Project Description. http://wiki.apache.org/hadoop/ProjectDescription. Accessed April 30, 2011.

[2] HDFS Design and Data Replication. http://hadoop.apache.org/common/docs/current/hdfs_design.html. Accessed June 25, 2011.

[3] Map Reduce. http://developer.yahoo.com/hadoop/tutorial/module4.html. Accessed June 25, 2011.

[4] The Hadoop approach and Data Distribution. http://developer.yahoo.com/hadoop/tutorial/module1.html. Accessed June 25, 2011.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.

[7] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: Mining Peta-Scale Graphs. In Knowledge and Information Systems (KAIS), Springer, 2010.

[8] Song Liu. Pagerank on MapReduce. http://joycrawler.googlecode.com/files/Pagerankisagoodthing.pdf. Accessed September 20, 2011.

[9] Tom White. Hadoop: The Definitive Guide. O'Reilly, 2009.

[10] Tom White. The Small Files Problem. http://www.cloudera.com/blog/2009/02/the-small-files-problem/. Accessed October 25, 2011.