Incremental PageRank acceleration using Sparse Matrix-Sparse Vector Multiplication

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Shridhar Ramachandran

Graduate Program in Computer Science and Engineering

The Ohio State University

2016

Master’s Examination Committee:

Dr. P. Sadayappan, Advisor
Dr. Srinivasan Parthasarathy

Approved for Public Release
Distribution Unlimited
© Copyright by
Shridhar Ramachandran
2016
Abstract

PageRank is an important measure to evaluate the relative importance of a node within a graph. Traditional linear algebraic methods to compute PageRank scores of nodes in a graph include the power method, which involves an iterative series of Sparse Matrix Vector multiplication (SpMV) operations. As graphs dynamically change over time, it becomes important to keep the PageRank scores updated in an efficient manner. Through the work done in this thesis, we try and solve this problem of efficiently updating PageRank scores by devising an algorithm that formulates the power method computation on the updated graph as a series of Sparse Matrix Sparse Vector (SpMSpV) multiplication operations - a method that improves upon the naive technique of recomputing PageRank scores from scratch. We present a rigorous mathematical proof for this. We also include several optimizations in our design and implementation of the algorithm based on theoretical and heuristic observations of the SpMSpV routine and our mathematical formulation of incremental PageRank. In addition to these optimizations, we implement a smart threshold-based technique during the SpMSpV routine to calculate a high quality approximation of PageRank scores, while improving efficiency significantly. Many of these optimization and approximation techniques were extended in the multicore and GPU implementations of our algorithm. We show promising empirical results for a wide variety of scale-free graphs.
To my dad and my life compass, K.B. Ramachandran
Acknowledgments

At the very outset, I would like to express my deepest gratitude to Dr. P Sadayappan for giving me this wonderful opportunity of working with him. His ever-curious nature, constant guidance and clarity of thought continue to awe me to this day and have shaped me as a student and researcher. It wouldn’t be an understatement to say that he is the most inspiring teacher I have had the fortune to learn from. I would also like to thank Dr. Srini Parthasarathy - my interest in data mining and network science can be solely attributed to his superbly taught courses and the direction he has provided. This thesis is influenced almost entirely by the counsel and encouragement that these 2 professors have given me.

I also had the pleasure of working with Dr. John Eisenlohr and Dr Gerald Sabin and thank them for their insightful and though-provoking comments during the meetings we had.

I’d like to thank my lab mates, past and present - Venmugil, Shashank, Prashant, Wen-lei, Qingpeng, Ankur, Gaganjit, Kiran, Changwan and Nisa. All those discussions with them taught me much about the many challenging aspects of high performance computing research, and all those fights for nodes on Cerberus taught me that resource contention is not restricted to the memory hierarchy. The past 16 months at DL 574 have been truly memorable.
My acknowledgment would be incomplete without mentioning my family of friends here in Columbus - to Gaganjit, Kaushik, Albert and Meraj - with whom, through procrastination filled lunches, chai sessions and enlightening conversations, the myriad frustrations of grad school were not only dissipated, but conquered. And to Karun, Varsha, Sahil, Soumyo - the many hours spent playing board games, watching cricket, bar hopping in Short North and beating you all at squash were unforgettable. Kaushik and Kiran, though already mentioned, deserve a special shout out for setting the bar on how cool roommates can be.

A huge thank-you as well to Gaurab "GG" Ghosh (from Austin) and Shivangi (from Minneapolis) for helping me stay (in)sane with countless virtual and non-virtual, intellectual and non-intellectual conversations, as we jaunted across different cities, countries and continents in the past year.

Last but in no ways the least, I would like to thank my mother and sister back home in Chennai. Without their constant support, love and belief in me, I wouldn’t have taken the 14000 kilometer plunge to pursue the wonderfully eye-opening experience that has been graduate school in the US.

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Vita

2012 ........................................... B.E Computer Science and Engineering,
Birla Institute of Technology and Science - Pilani, India

2012-2014 .................. Software Development Engineer,
Amazon.com, Inc.
Bangalore, India

2014-present .................. Graduate Research Associate,
Department of Computer Science and Engineering
The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 PageRank and the web</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Power Iteration and SpMV</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Incremental PageRank</td>
<td>3</td>
</tr>
<tr>
<td>2. An alternate formulation of Incremental PageRank</td>
<td>5</td>
</tr>
<tr>
<td>3. SpMSpV - Design and Implementation</td>
<td>9</td>
</tr>
<tr>
<td>3.1 Data structure representation of the Sparse Vector</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Compressed Sparse Column representation of Sparse Matrices</td>
<td>10</td>
</tr>
<tr>
<td>3.3 Reducing reads and writes in the Gather phase</td>
<td>12</td>
</tr>
<tr>
<td>3.4 In-place SpMSpV</td>
<td>12</td>
</tr>
<tr>
<td>3.5 Performance, Analysis and Improvements</td>
<td>15</td>
</tr>
<tr>
<td>3.5.1 Datasets</td>
<td>15</td>
</tr>
<tr>
<td>3.5.2 Benchmarking SpMSpV vs SpMV</td>
<td>15</td>
</tr>
<tr>
<td>3.5.3 Analysis and Improvement</td>
<td>15</td>
</tr>
</tbody>
</table>

vii
4. Approximation techniques ........................................... 22
   4.1 Normalized Discounted Cumulative Gain .................... 23
   4.2 Threshold-based approach to SpMSpV ....................... 25
   4.3 Performance of Threshold-based Approach .................. 25

5. Parallelization of Incremental PageRank ....................... 30
   5.1 Multicore ......................................................... 30
      5.1.1 Parallelism within columns of the Sparse Matrix .... 30
      5.1.2 Load balanced work-sharing across columns of the Matrix with atomics ........................................ 32
      5.1.3 Load balanced work-sharing across columns of the Matrix with segmented updates ............................... 33
   5.2 GPU implementation ............................................ 35

6. Conclusion ......................................................... 38

Appendices

A. Running Incremental PageRank .................................. 40

Bibliography .......................................................... 42
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Data structure representation of a SparseVector</td>
</tr>
<tr>
<td>3.2</td>
<td>Data structure representation of a SparseMatrix</td>
</tr>
<tr>
<td>3.3</td>
<td>Comparison of Reads, Writes and FLOPs for SpMV and SpMSpV</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>19</td>
</tr>
<tr>
<td>3.5</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>28</td>
</tr>
</tbody>
</table>
4.3 GFLOPS comparison of SpMSpV-PageRank over PageRank from scratch.
The number of flops was assumed to be two times the sum of the number of non-zeros in the Sparse Matrix and the number of elements in the PageRank vector. The algorithms were run on an Intel(R) Xeon(R) CPU E3-1275 V2 @ 3.50GHz

5.1 Speedup of column work-sharing version over base SpMSpV implementation on a 4-core machine with input vector density of 10%

5.2 Speedup of 4 core and 8 core versions over base SpMSpV implementation using the multicore-atomics method for an input vector with density 10%

5.3 Speedup of 4 core and 8 core versions over base SpMSpV implementation using the multicore segmented approach for an input vector with density 10%

5.4 GFLOP/s of SpMV by ModernGPU vs SpMSpV with Merge Sort vs SpMSpV with Radix Sort for input vector with density 10%
Chapter 1: Introduction

1.1 PageRank and the web

The PageRank [10] algorithm is a very effective measure of ranking nodes in graphs. Devised in 1998, it is now one of the most popular ranking algorithms on web graphs and social networks. PageRank scores of web pages is based solely on the hyperlink structure of the web, and this provides a crucial measure of web page importance. Much research has been performed on PageRank computation. In [3], Berkhin provides a detailed survey of PageRank algorithms. There are broadly 2 categories of methods to compute PageRank- Linear Algebraic methods (also known as Power method) and Monte Carlo methods. This thesis will concentrate on the Power method computation.

The PageRank of a node measures the relative importance of that node in the graph by counting the number of inbound links weighted by the page rank of the source of the inbound link. Equation 1.1 shows the formula for page rank of a node. $M(v_i)$ is the set of nodes which link to $v_i$, $L(v_j)$ is the number of nodes to which $v_i$ links, $\alpha$ is a damping factor, and $N$ is the total number of nodes in the graph.

\[ PR(v_i) = \frac{\alpha}{N} + (1 - \alpha) \sum_{v_j \in M(v_i)} \frac{PR(v_j)}{L(v_j)} \]  
(1.1)
Page Rank can be computed in an iterative manner, beginning with a vector $V$ in which each coordinate is set to $\frac{1}{N}$. Each iteration uses Equation 1.1 to update the coordinates of $V$, using the values at iteration $k$ as inputs for determining the values at iteration $k+1$.

From the above, we can observe that computing computationally expensive. While in [10], the authors heuristically show that the number of iterations is several orders of degree smaller than the size of the graph, there is still the matter of summing up all inbound PageRank values for each node.

1.2 Power Iteration and SpMV

The linear algebraic representation of Equation 1.1 is:

$$V_i = \alpha V_{\text{init}} + (1 - \alpha) A^T.V_{i-1}$$ (1.2)

Here, $V_{\text{init}}$ is the initial vector and $A$ is the row normalized adjacency matrix of the graph $G$. A common application of PageRank is in web graphs and social networks, where the adjacency matrix is a sparse matrix. Hence $A^T.V_{i-1}$ is a Sparse Matrix Vector multiplication (SpMV) operation.

Algorithm 1: Power Iteration method to compute PageRank

<table>
<thead>
<tr>
<th>Input</th>
<th>Matrix $A$, number of nodes $n$, damping factor $\alpha$, tolerance $\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>PageRank vector $V_{PR}$</td>
</tr>
</tbody>
</table>

1 procedure PageRank $(A, n, \alpha, \epsilon)$

2 $V_0 = 0$; $V_{old} = 0$; $V_{new} = 0$

3 while IsNotConverged($V_{new}$; $V_{old}$; $\epsilon$) do

4 $V_{old} = V_{new}$

5 $V_{new} = \alpha \cdot V_0 + (1 - \alpha) \cdot \text{SpMV}(A,V_{old})$

6 end

7 return $V_{new}$
In the Power Iteration algorithm, the most computationally dominant step is the SpMV step. Thus, the performance of the SpMV kernel is critical to determining the performance of the PageRank algorithm. SpMV is a major component of iterative methods that solve large-scale linear systems and eigenvalue problems. It is also a heavily explored topic in the realm of graph theory. These significant research efforts have resulted in many optimized versions of the SpMV operation on CPUs, modern-day multicores as well as general-purpose GPU (GPGPU) platforms. For example, libraries such as Intels MKL and Nvidias cuSPARSE and Cusp provide users with highly tuned SpMV implementations designed to maximize core efficiency and memory bandwidth.

1.3 Incremental PageRank

PageRank stability as a graph changes has been an important topic of research as well. An important question to be asked is - when a graph changes, can the PageRank values of nodes be efficiently updated? The naive approach of recomputing PageRank from scratch is deemed too computationally expensive and hence it is worth investigating any improvements over this. As mentioned in [3], many approaches to improving PageRank and Incremental PageRank involve reducing the number of iterations to convergence. McSherry [9] also devises a parametric approach to reduce the number of iterations, which can be extended to incrementally updated graphs. Other approaches include [1], where the authors develop an ”Evolving Graph Model”, where the differences in the network/graph are not known to the algorithm, and instead, nodes whose PageRank scores have a high likelihood of being updated are
smartly “guessed” and updated to obtain a good approximation of the new PageRank scores.

The remainder of this thesis explores in detail approaches to improve performance of Incremental PageRank, beyond just accelerating it’s convergence. Firstly, a formulation for recomputing PageRank on an incrementally updated graph using Sparse Matrix Sparse Vector operations is examined, followed by various algorithmic and implementation-specific optimizations that were devised to design the final Incremental PageRank algorithm. Chapter 4 discusses techniques used to improve performance by calculating an approximation of the PageRank scores as well as the reasons behind it. Subsequent chapters also include Multicore and GPU implementations of the algorithm. Results are showcased throughout, as and when relevant. The thesis is concluded with a summary of all the contributions made.
Chapter 2: An alternate formulation of Incremental PageRank

As mentioned in the previous chapter, the linear algebraic representation of Equation 1.1 is:

\[ V_i = \alpha V_{\text{init}} + (1 - \alpha) A^T V_{i-1} \]  

(2.1)

Here, \( V_{\text{init}} \) is the initial vector and \( A \) is the row normalized adjacency matrix of the graph \( G \). A common application of PageRank is in web graphs and social networks, where the adjacency matrix is a sparse matrix. Hence \( A^T V_{i-1} \) is an SpMV operation.

In dynamic graphs, the structure of the graph is subject to changes over time. Thus the adjacency matrix and consequently, the PageRank vector changes as well. If the row normalized adjacency matrix of the modified graph \( G' \) is \( (A + \Delta A) \), then the new PageRank vector can be computed as follows:

\[ V_i = \alpha V_{\text{init}} + (1 - \alpha) (A + \Delta A)^T V_{i-1} \]  

(2.2)

Faster convergence of Equation 2.2 is obtained by keeping \( V_0 = V_* \), where \( V_* \) is the final computed PageRank vector of the original graph \( G \).

The different iterations of Equation 2.2 can be expressed as follows:
Iteration 1:

\[ V_1 = \alpha V_{init} + (1 - \alpha)(A + \Delta A)^T.V_s \]
\[ = \alpha V_{init} + (1 - \alpha)A^T.V_s + (1 - \alpha)\Delta A^T.V_s \]
\[ = V_s + (1 - \alpha)\Delta A^T.V_s \]

\((1 - \alpha)\Delta A^T.V_s\) is the product of an ultra-sparse matrix and a dense vector, which is a sparse vector. Let this resultant sparse vector be \(\lambda^{sp}\).

Iteration 2 would be:

\[ V_2 = \alpha V_{init} + (1 - \alpha)(A + \Delta A)^T.V_1 \]
\[ = \alpha V_{init} + (1 - \alpha)(A + \Delta A)^T.(V_s + \lambda^{sp}) \]
\[ = \alpha V_{init} + (1 - \alpha)A^T.V_s + (1 - \alpha)\Delta A^T.V_s \]
\[ + (1 - \alpha)(A + \Delta A)^T.\lambda^{sp} \]
\[ = V_s + \lambda^{sp} + (1 - \alpha)(A + \Delta A)^T.\lambda^{sp} \]

\((1 - \alpha)(A + \Delta A)^T.\lambda^{sp}\) is the product of a sparse matrix and a sparse vector (SpMSpV), which is a sparse vector. Let this resultant sparse vector be \(V_1^{sp}\).

Iteration 3:

\[ V_3 = \alpha V_{init} + (1 - \alpha)(A + \Delta A)^T.V_2 \]
\[ = \alpha V_{init} + (1 - \alpha)(A + \Delta A)^T.(V_s + \lambda^{sp} + V_1^{sp}) \]
\[ = \alpha V_{init} + (1 - \alpha)A^T.V_s + (1 - \alpha)\Delta A^T.V_s \]
\[ + (1 - \alpha)(A + \Delta A)^T.(\lambda^{sp} + V_1^{sp}) \]
\[ = V_s + \lambda^{sp} + (1 - \alpha)(A + \Delta A)^T.(\lambda^{sp} + V_1^{sp}) \]
\[ = V_s + \lambda^{sp} + V_2^{sp} \]
On expanding further iterations, we get the following:

\[ V_i = V_* + \lambda^{sp} + V_{i-1}^{sp} \]  

(2.3)

where

\[ V_i^{sp} = (1 - \alpha)(A + \Delta A)^T(\lambda^{sp} + V_{i-1}^{sp}) \]  

(2.4)

and \( V_0^{sp} \) is the Zero vector

The sum of the 2 vectors \((\lambda^{sp} + V_{i-1}^{sp})\) is a sparse vector and hence Equation 2.4, which is iterative, is essentially a series of SpMSpV operations. On convergence, this equation will result in a vector \( V_{final}^{sp} \) and, thus Equation 2.3 will yield:

\[ V_{final} = V_* + \lambda^{sp} + V_{final}^{sp} \]

The change in the PageRank vector \( \Delta V = V_{final} - V_* \) is:

\[ \Delta V = \lambda^{sp} + V_{final}^{sp} \]

The change in PageRank can thus be expressed as an iterative computation of SpMSpV operations.

The above mathematical proof is condensed into pseudo code, which is detailed in Algorithm 2. It is important to note that the algorithmic complexity of the Incremental PageRank algorithm outlined is not different from the re-computation of PageRank values from scratch i.e, the number of iterations to convergence in Algorithm 2 is the same as that in Algorithm 1.

This concludes this section on the alternate representation of the Incremental PageRank algorithm using SpMSpV operations. Note that while it is intuitive that an SpMSpV operation should perform better, if not the same as the corresponding SpMV operation, this is not formally examined until the next chapter. In the following
Algorithm 2: Incremental PageRank using Sparse Matrix Sparse Vector Multiplication operations

| **Input** | Sparse Matrix \((A + \Delta A)^T\), Sparse Matrix \(\Delta A^T\), Dense Vector \(V_s\), \(\alpha\), tolerance \(\epsilon\) |
| **Output**: Updated PageRank vector \(V_{PR}\) |

1. **procedure** IncrementalPageRank \(((A + \Delta A)^T, \Delta A^T, V_s, \alpha, \epsilon)\)
2. \(\lambda_{sp} = (1 - \alpha) \ast \text{SpMV}(\Delta A^T, V_s)\)
3. \(V_{sp\ old} = 0; V_{sp\ new} = V_s\)
4. **while** IsNotConverged\((V_{sp\ new}, V_{sp\ old}, \epsilon)\) **do**
5. \(V_{old} = V_{sp\ new}\)
6. \(V_{sp\ new} = (1 - \alpha) \ast \text{SpMSpV}((A + \Delta A)^T, \text{SparseVectorSum}(\lambda_{sp}, V_{sp\ new}))\)
7. **end**
8. **return** \(V_{sp\ new} + V_s + \lambda_{sp}\)

In this chapter, we provide more details about the design of the SpMSpV algorithm as well as algorithmic and data structure related optimizations that were made to SpMSpV in order to improve performance.
Chapter 3: SpMSpV - Design and Implementation

As mentioned in previous chapters, the most computationally expensive step in PageRank is the SpMV operation. This translates directly into the SpMSpV operation being the most computationally expensive step in the Incremental PageRank algorithm. Thus, it is crucial to devise an optimized version of the SpMSpV method. The efficient design and implementation of this method has both a data structure and an algorithmic flavor to it, which will be discussed in this chapter.

3.1 Data structure representation of the Sparse Vector

As seen in Table 3.1, the Sparse Vector is represented as 2 “sparse” arrays and 2 dense arrays, along with 2 variables $nnz$ (indicating the total number of non-zeros in the Vector) and $size$ (indicating the actual size of the vector were it not sparse). $indexesSparse$ contains the indexes in the vector that have non-zero values, and $valsSparse$ contains the corresponding list of values. $indexesDense$ is a boolean array that indicates whether or not the value at a particular index is non-zero or not. $valsDense$ is essentially the same as the dense representation of the vector. The sizes of the arrays are mentioned in Table 3.1. The relationship between these 4 arrays is:

1. For all $i \in [0, nnz)$, $indexesDense[indexesSparse[i]] = true$, and all other values in $indexesDense$ are false
2. For all $i \in [0, \text{nnz})$, $\text{valsDense}[\text{indexesSparse}[i]] = \text{valsSparse}[i]$.

The reasons behind representing the SparseVector in this particular fashion will be explained in further sections of this chapter.

### 3.2 Compressed Sparse Column representation of Sparse Matrices

Traditional SpMV implementations consist of the Sparse Matrix in Compressed Sparse Row (CSR) format and the Vector as an array. The output Vector is computed by traversing through the *column* and *values* arrays of the Matrix and computing its product with the corresponding value in the Vector’s array. The total number of Floating Point Operations (FLOPs) will always be equal to the number of non-zeros in the Sparse Matrix regardless of the density of the input Vector. The number of memory reads will be equal to thrice the number of non-zeros in the Sparse Matrix data structure (input vector *value*, Sparse Matrix *column* and Sparse Matrix *value*). In addition, the reads of the input vector are not guaranteed to exhibit good spatial or temporal locality. Thus, we are not really making use of the density of the input vector.
Table 3.2: Data structure representation of a SparseMatrix

<table>
<thead>
<tr>
<th>SparseMatrix A</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{int } \text{nnz} )</td>
</tr>
<tr>
<td>( \text{int } \text{numCols} )</td>
</tr>
<tr>
<td>( \text{colPtrs} [\text{numCols} + 1] )</td>
</tr>
<tr>
<td>( \text{rows} [\text{nnz} ] )</td>
</tr>
<tr>
<td>( \text{vals} [\text{nnz} ] )</td>
</tr>
</tbody>
</table>

Even if we extend this to using the SparseVector data structure defined in the previous section, while we can easily adjust the algorithm to reduce the number of FLOPs (by performing the multiply-and-add only for the non-zero values in the Vector), the number of memory reads do not change. This is crucial as SpMV is traditionally a memory-bound problem. To solve this issue, we instead make use of the Compressed Sparse Column (CSC) representation of a matrix, as shown in Table 3.2. The algorithm now consists of going through each column \( i \) of the Sparse Matrix only for those \( i \) that are in \( \text{indexesSparse} \). Thus, the total number of FLOPs and memory reads are reduced significantly. The algorithm can now be summarized in 3 phases as shown below:

1. Clearing \( \text{denseIndexes} \) and \( \text{denseVals} \)
2. Performing the actual SpMSpV by reading from \( \text{indexesSparse} \) and \( \text{valsSparse} \) and writing to \( \text{indexesDense} \) and \( \text{valsDense} \)
3. Gathering the new sparse indexes and values from \( \text{valsDense} \).
3.3 Reducing reads and writes in the Gather phase

The gather phases consists of going through all size elements in $valsDense$ and populating $indexesSparse$ and $valsSparse$ accordingly. However, it can be noted and proved that in Incremental PageRank, an index whose value is non-zero in iteration $k$ is guaranteed to be non-zero in iteration $k + 1$. Thus, we can reduce the number of reads from $valsDense$ and writes to $indexesSparse$ in the gather phase.

3.4 In-place SpMSpV

A corollary of the above design choices is that the SpMSpV can be performed in-place. i.e, no new data structures are created for the output vector and instead the input vector’s memory is reused to store the resultant vector. This is done smartly to ensure no data is lost or read incorrectly. The entire SpMSpV algorithm, after the improvements made, is shown in Algorithm 3.

While it is evident that the number of floating point operations (FLOPs) in an SpMSpV operation is fewer than those in the corresponding SpMV operation, the exact columns of the Sparse Matrix participating in the SpMSpV operation are required to be known in order to quantify the exact performance gain. Since SpMV is traditionally a memory-bound, in addition to floating point operations, it is useful to compare the number of reads and writes. A characterization of the exact number of flops and reads and writes are mentioned in Table 3.3. Note that $numFlops$ in the table refers to the number of floating point operations that occur in the SpMSpV operation. This is equivalent to the sum of the sizes of all the columns participating in the SpMSpV operation.
### Algorithm 3: In-place SpMSpV

**Input**: Sparse Matrix $A$, Sparse Vector $V$

**Output**: Resultant Sparse Vector $V$

1. **procedure** SpMSpV ($A,V$)
   
   /* clear all values in the vector */

2. for $i$ in $V$.indexesSparse[] do
   
3. | $V$.valsDense[$V$.indexesSparse[$i$]] = 0
   
4. end

5. addedIndexes[] ← φ

6. for $i$ in $V$.indexesSparse[] do

7. | for currColPtr in range($A$.cols[$i$],$A$.cols[$i$+1]) do

8. | | currRow = $A$.rows[currColPtr]


10. | | if $V$.indexesDense[currRow] is false then

11. | | | addedIndexes.push(currRow)

12. | | end

13. end

14. /* Update SparseVector $V$ with new values */

15. for $i$ in $V$.indexesSparse[] do

16. | $V$.valsSparse[$i$] = $V$.valsDense[addedIndexes[$i$]]

17. end

18. for $i$ in $V$.addedIndexes[] do

19. | $V$.indexesSparse.push(addedIndexes[$i$])

20. | $V$.valsSparse.push($V$.valsDense[addedIndexes[$i$]])

21. end

22. return $V$

---

<table>
<thead>
<tr>
<th></th>
<th>SpMV</th>
<th>SpMSpV</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOPs</td>
<td>$\text{nnz}$</td>
<td>$\text{numFlops}$</td>
</tr>
<tr>
<td>Reads</td>
<td>$3*\text{nnz}$</td>
<td>$3*\text{numFlops} + 2*\text{OutputSparseVector.size}$</td>
</tr>
<tr>
<td>Writes</td>
<td>$\text{nnz}$</td>
<td>$\text{numFlops}$</td>
</tr>
</tbody>
</table>

Table 3.3: Comparison of Reads, Writes and FLOPs for SpMV and SpMSpV
Figure 3.1: This figure shows a pictorial representation of different phases in Algorithm 3. The Sparse Matrix is shown as a dense matrix for the purpose of clarity. The cells marked in yellow represent the non-zero values in the Sparse Vector. The cells marked in green represent the newly added/updated non-zero values and the cells marked in red represent the newly added non-zero values in the sparse representation of values. Note that it requires only 6 FLOPs to compute the resultant vector, as opposed to the 16 that would have been required to compute in case of SpMV.
3.5 Performance, Analysis and Improvements

3.5.1 Datasets

The list of matrices used to run IncrementalPageRank and other benchmarks on were: cant, consph, mc2depi, filter3D, mac_econ_fwd500, mc2depi, pdb1HYS, pwtk, rma10, scircuit, webbase-1M, shipsec1, 2cubes_sphere, cage12, hood, m133-b3, majorbasis, mario002, mono_500Hz, offshore, poisson3Da, qcd5_4, email-Enron, facebook_combined, WIPHI_graph, cit-HepPh, DIP_unweighted_graph, loc-gowalla_edges.

These are a set of matrices that are commonly used in the data mining and high performance computing community. They were taken from SNAP [11] and the University of Florida Sparse Matrix Collection [6].

3.5.2 Benchmarking SpMSpV vs SpMV

Figure 3.2 consists of results of a subset of the graphs after benchmarking the SpMSpV and SpMV subroutines as a function of input vector density for different graphs. From the figure, we see that there is a significant speedup for lower input vector densities. The maximum densities at which SpMSpV outperforms SpMV are between 35-45%, depending on the graph.

3.5.3 Analysis and Improvement

Recall that in the proof for our formulation of Incremental PageRank as a series of SpMSpV operations, we made an assertion that \( V_{sp} \) should remain sparse for all iterations as it is always the sum of a sparse vector (\( \lambda_{sp} \) and the product of an ultra-sparse matrix and a sparse vector(\( A + \Delta A.V_{sp} \)). While \( \lambda_{sp} \) is guaranteed to be sparse as it is the same for all iterations, the Sparse Matrix-Vector product is not necessarily
Figure 3.2: SpMSpV vs SpMV speedup as a function of vector density for different graphs
Figure 3.3: Speedup of Incremental PR using SpMSpV over PageRank from scratch
sparse. This is dependent on the nature of the graph. When a Sparse Matrix represents the adjacency matrix (or a stochastic transformation of the adjacency matrix) of the graph and a Vector represents nodes in a graph, the resulting vector of the SpMV operation gives information about the one hop-distance of the nodes that were represented by the input vector.

It is easy to see that if this continues in an iterative fashion, i.e. the Sparse Matrix is multiplied with the resultant vector of the previous iteration, then after \( k \) iterations, the resultant vector will provide some information about the \( k \)-hop neighbors of the graph.

An implication of this is that - even if the initial input vector is very sparse, the resultant vector after an iterative series of SpMV operations can become dense. The rate at which this vector’s density increases is dependent on the nature of the graph. Graphs with power law degree distribution have low diameter, so the increase in density is exponential. While disconnected graphs with large diameters have a much slower (and a constant) rate of increase in density.

On plotting the density of \( V^s_{new} \) as a function of iteration for different graphs, we see that for almost all graphs, the density of the vector reaches nearly 100% within a handful of iterations. This is showcased in Table 3.4. Juxtaposing this with the results in Figure 3.2, we obtain a concrete reason behind the poor performance of our Incremental PageRank algorithm shown in 3.3 - since the performance of SpMSpV is poorer than SpMV for vector densities beyond 35-40% and the majority of iterations in incremental PageRank perform SpMSpV at densities greater than 90%, performance of incremental PageRank based on SpMSpV deteriorates. The one outlier is the matrix corresponding to the webbase-1M dataset - running incremental
PageRank using SpMSpV shows a speedup of 2.18 over the SpMV PageRank. This can be attributed to the density of $V^{sp}$ plateauing at 8% in Figure 3.4.

The nature of our algorithm allows for one possible simplistic solution to this problem. Recall Equation 2.3 from the previous section.

$$ V_i = V_* + \lambda^{sp} + V_i^{sp} $$

This equation implies that the intermediate PageRank vector at any iteration of the original SpMV-PageRank algorithm can be computed from the intermediate $V_i^{sp}$ vector of our SpMSpV-incremental approach. Thus, we can set a vector density...
threshold which we can use to switch between the SpMSpV approach and the SpMV approach. All that is required is to calculate $V_{\text{new}}$ from $V_{\text{new}}^{\text{sp}}$ as soon as we cross a certain threshold. This approach, illustrated in algorithm 4, is guaranteed to give us the exact same values of PageRank. The updated switch-based algorithm was run, setting the threshold to 30%, and the performance did improve, as can be seen in Figure 3.5.

**Algorithm 4:** Incremental PageRank that switches between SpMSpV and SpMV

**Input:** Sparse Matrix $(A + \Delta A)^T$, Sparse Matrix $\Delta A^T$, Dense Vector $V_*$, $\alpha$, tolerance $\epsilon$, density threshold $\theta$

**Output:** Updated PageRank vector $V_{PR}$

```
procedure IncrementalPageRank ($(A + \Delta A)^T, \Delta A^T, V_*, \alpha, \epsilon)$

1. $\lambda^{sp} = (1 - \alpha) \ast \text{SpMV}(\Delta A^T, V_*)$
2. $V_{\text{old}}^{sp} = 0; V_{\text{new}}^{sp} = V_*$
3. while IsNotConverged($V_{\text{new}}^{sp}, V_{\text{old}}^{sp}, \epsilon$) do

4. \hspace{1em} $V_{\text{old}}^{sp} = V_{\text{new}}^{sp}$
5. \hspace{1em} $V_{\text{new}}^{sp} = (1 - \alpha) \ast \text{SpMSpV}((A + \Delta A)^T, \text{SparseVectorSum}(\lambda^{sp}, V_{\text{new}}^{sp}))$
6. \hspace{1em} if $\rho(V_{\text{new}}^{sp}) > \theta$ then
7. \hspace{2em} break;
8. \hspace{1em} end
9. \hspace{1em} end
10. $V_{\text{new}} = V_{\text{new}}^{sp} + V_* + \lambda^{sp}$
11. $V_{\text{old}} = V_{\text{old}}^{sp} + V_* + \lambda^{sp}$
12. while IsNotConverged($V_{\text{new}}, V_{\text{old}}, \epsilon$) do
13. \hspace{1em} $V_{\text{old}} = V_{\text{new}}$
14. \hspace{1em} $V_{\text{new}} = \alpha \ast V_0 + (1 - \alpha) \ast \text{SpMV}(A.V_{\text{old}})$
15. \hspace{1em} end
16. return $V_{\text{new}}$
```

20
Figure 3.5: Speedup of Incremental PR using SpMspV over PageRank using the switch-based algorithm
Chapter 4: Approximation techniques

PageRank on the web at present often involves web crawls over billions of web pages. When taken in the context of web search, even though search queries may result in hundreds of thousands of results, only a few dozen are presented as relevant to the user. In addition, while the ranking of the pages presented does matter, the exact PageRank scores are more often than not completely opaque and irrelevant to the user. These points have resulted in several approximations to the PageRank model. various techniques have been proposed for speeding up these analyses by distributing the link graph among multiple sites.

Thus, a pertinent question to be asked is - is it possible to perform a more efficient version of a PageRank-like computation such that the ranks of the top tens (or hundreds) of results do not alter greatly? More formally, if running PageRank on a graph $G(V, E)$ results in an ordered set of nodes $O_1$ and running an approximated version of PageRank results in an ordered set of nodes $O_2$ (both $O_1$ and $O_2$ ordered in decreasing order of PageRank/approximated PageRank score). If we are interested in say, the top $k$ nodes of the output vector, then the approximated version can be considered good if the top $k$ nodes in $O_1$ are approximately the same as in $O_2$.

Note that the above paragraph intentionally uses qualitative terms such as ”good” and ”approximately the same” just to provide an intuition as to why approximations
to PageRank are worth considering. In order to quantitatively compare the correctness of the approximated PageRank with the original, a metric known as Normalized Discounted Cumulative Gain is used.

4.1 Normalized Discounted Cumulative Gain

Normalized Discounted Cumulative Gain (NDCG) is a popular metric for measuring the efficacy of any ranking algorithm. It is used in a wide variety of ranking applications and extensive empirical and theoretical [12] research has been conducted to analyze the accuracy of the metric.

The formal definition of NDCG, according to [4] is a normalization of the Discounted Cumulative Gain (DCG) measure. DCG is a weighted sum of the degree of relevancy of the ranked items. The weight is a decreasing function of the rank (position) of the object, and therefore called discount. The original reason for introducing the discount is that the probability that a user views a document decreases with respect to its rank. NDCG normalizes DCG by the Ideal DCG (IDCG), which is simply the DCG measure of the best ranking result. Thus NDCG measure is always a number in [0, 1]. Strictly speaking, NDCG is a family of ranking measures, since there is flexibility in choosing the discount function. The logarithmic discount $1/\log(1 + r)$, where $r$ is the rank, dominated the literature and applications. We will refer to NDCG with logarithmic discount as the standard NDCG. Search engine systems also use a cut-off top-k version of NDCG. That is, the discount is set to be zero for ranks larger than $k$. Such NDCG measures are usually referred to as NDCG@k.

NDCG is thus a useful metric, as it provides a computationally inexpensive normalized scheme to compare vastly different algorithms.
Algorithm 5: In-place SpMSpV used to calculate approximate Incremental PageRank scores

**Input**: Sparse Matrix \( A \), Sparse Vector \( V \), Threshold \( \psi \)

**Output**: Resultant Sparse Vector \( V \)

1. **procedure** \( \text{SpMSpV} (A, V, \psi) \)
   2. /* clear all values in the vector */
   3. for \( i \) in \( V.\text{indexesSparse}[] \) do
   4. \( V.\text{valsDense}[V.\text{indexesSparse}[i]] = 0 \)
   5. end
   6. addedIndexes[] \( \leftarrow \phi \)
   7. for \( i \) in \( V.\text{indexesSparse}[] \) do
   8. if \( V.\text{valsSparse}[i] \neq 0 \) then
   9. for \( \text{currColPtr} \) in range \( (A.\text{cols}[i], A.\text{cols}[i+1]) \) do
   10. currRow = \( A.\text{rows[}\text{currColPtr}] \)
   11. \( V.\text{valsDense}[\text{currRow}] = A.\text{vals[}\text{currColPtr}] \times V.\text{valsSparse}[i] \)
   12. if \( \text{V.\text{indexesDense[}\text{currRow}] is false} \) then
   13. addedIndexes.push(\( \text{currRow} \))
   14. end
   15. end
   16. end
   17. /* Update SparseVector \( V \) with new values */
   18. for \( i \) in \( V.\text{indexesSparse}[] \) do
   19. if \( V.\text{valsDense}[V.\text{indexesSparse}[i]] > \psi \) then
   20. \( V.\text{valsSparse}[i] = V.\text{valsDense}[\text{addedIndexes[i]}] \)
   21. else
   22. \( V.\text{indexesDense} = \text{false} \)
   23. \( V.\text{valsDense} = 0 \)
   24. end
   25. end
   26. for \( i \) in \( V.\text{addedIndexes}[] \) do
   27. if \( V.\text{valsDense}[\text{addedIndexes[i]}] > \psi \) then
   28. \( V.\text{indexesSparse}.\text{push(}\text{addedIndexes[i]}\))
   29. \( V.\text{valsSparse}.\text{push(}\text{V.\text{valsDense}[\text{addedIndexes[i]}]}\))
   30. else
   31. \( V.\text{indexesDense} = \text{false} \)
   32. \( V.\text{valsDense} = 0 \)
   33. end
   34. end
   35. return \( V \)
4.2 Threshold-based approach to SpMSpV

Running the original algorithms - Algorithm 2 and Algorithm 5 on the datasets did not provide satisfactory results. The reason, as mentioned in the previous section was the burgeoning density of the input Vector. To overcome that, we devised an approximation approach, the motive being - since the number of changes to the graph is small, PageRank values will not change significantly for most nodes and hence nodes with values that are very small will have a negligible effect on their neighbors’ values. In other words, we maintain a threshold such that only values above that threshold will be entered into the resultant vector in the SpMSpV operation.

Algorithm 5 shows the updated algorithm to account for thresholds while reinserting values into the resultant vector in SpMSpV. The highlighted lines indicate where the changes were made over the original non-approximated version. A simple example illustrating the algorithm is provided in Figure 4.1.

4.3 Performance of Threshold-based Approach

This algorithm was then run and much better results were obtained, showcased in Figure 4.3, with 10 out of the 28 graphs having a speedup of $\geq 2$. In addition, high NDCG@k values (for $k=1000,2500,10000$), with the average NDCG@10000 value for all graphs being 0.98, proved both the efficacy and efficiency of this approximation approach.

Note that all the performance results so far have been comparisons of SpMSpV-PageRank with the default version of PageRank using the Power Method written by us. While there has been a lot of research in improving the performance of PageRank, it was difficult to get an apples-to-apples comparison. The closest that was
Figure 4.1: This figure shows a pictorial representation of different phases in Algorithm 3. The Sparse Matrix is shown as a dense matrix for the purpose of clarity. The cells marked in yellow represent the non-zero values in the Sparse Vector. The cells marked in green represent the newly added/updated non-zero values and the cells marked in red represent the newly added non-zero values in the sparse representation of values. The cells marked in orange represent the values that did not cross the threshold (of 0.001) and hence are set back to 0.
found is the Power Method PageRank algorithm implementation provided by [7], and this was hence used as an additional yardstick for comparison. This implementation uses WebGraph [4], a framework for graph compression to perform operations such as Matrix-Vector multiply and Matrix-Matrix addition on graphs. This framework first compresses graphs using various compression techniques such as flat codes, that work especially well with integers with power-law distribution in a certain exponent range. In addition, WebGraph offers algorithms to access compressed graphs without actually decompressing them, using lazy techniques that delay the decompression until it is actually necessary. These are the reasons behind improved performance of WebGraph based incremental PageRank over our SpMSpV version, showcased in Figure 4.3.
Figure 4.2: Speedup of Incremental PR using Approximated SpMspV over PageRank from scratch
Figure 4.3: GFLOPS comparison of SpMSpV-PageRank over PageRank from scratch. The number of flops was assumed to be two times the sum of the number of non-zeros in the Sparse Matrix and the number of elements in the PageRank vector. The algorithms were run on an Intel(R) Xeon(R) CPU E3-1275 V2 @ 3.50GHz
Chapter 5: Parallelization of Incremental PageRank

Several attempts were made to extend incremental PageRank to multicore systems (specifically using OpenMP) and the GPU. Again, the focus of performance improvement was on the SpMSpV kernel. Most of the other changes required to make the incremental PageRank algorithm work on multicore systems are trivial (Sparse Vector-Sparse Vector addition). Hence, all the results and analyses performed in this chapter are from an SpMSpV standpoint.

5.1 Multicore

5.1.1 Parallelism within columns of the Sparse Matrix

The first attempt made was one that required minimum changes to the algorithm or implementation - work-sharing across values in a column in the sparse matrix. The advantage with this method is that there is no need for any atomics. However, the disadvantage of such an approach is the multiple implicit barriers due to the `pragma omp parallel` in the inner loop (line 8 in Algorithm 5).

As can be seen in Figure 5.1.1, the performance of the multi-core algorithm (when run on a 4-core machine) was quite poor, even for input vector density as low as 10% - with a maximum speedup of 2 and the single core version outperforming the multicore version for many matrices. An analysis of participating column sizes (or...
Figure 5.1: Speedup of column work-sharing version over base SpMSpV implementation on a 4-core machine with input vector density of 10%
degree distribution) revealed that many columns in the matrices aren’t large enough to warrant parallelism. Due to the scale-free nature of the graphs, most of the graphs had most column sizes $< 10$). Thus, the poor performance can be attributed to the thread scheduling overhead after the implicit barrier at the end of each column’s computations.

5.1.2 Load balanced work-sharing across columns of the Matrix with atomics

The next attempt was to share the work across columns of the Matrix. If the total number of FLOPs performed in an SpMSpV operation is $F$, and there are $p$ processors, the goal is to share the FLOPs across the $p$ processors such that each processor gets approximately $F/p$ FLOPs. The following was done to achieve this:

1. Create a temporary array (say `sizes_cols`) that contains the sizes of the cols in SpMat that are participating in the SpMSpV. This `sizes_cols` array can be created in parallel from the `cols` vector of the Sparse Matrix

2. Perform a parallel prefix sum on `sizes_cols`. This is essentially a prefix sum of the FLOPs that are computed in this SpMSpV (hence let the prefix sum array be named `prefix_sum_flops`)

3. The goal is to load balance the total FLOPs in the SpMSpV across threads. Hence, each thread now needs to find its starting and ending indexes in `prefix_sum_flops` such that each thread gets approximately $F/p$ flops. This ‘find’ is basically a binary search on the sorted array `prefix_sum_flops`. Since each thread knows what its approximate starting and ending FLOP values should be, the binary search can be done in parallel.
4. Each thread now proceeds with the standard SpMSpV algorithm, only working on its FLOPs, with 2 differences. Firstly, the Sparse Vector’s valsDense values are updated atomically. Secondly, an atomic-test-and-set is used to update indexesDense. If atomic-test-and-set(indexesDense[i]) returns false, it implies that the index i is being set for the first time (which means that an index who’s value was zero is now becoming non-zero), so it pushes i into a processor-local vector of indexes_added.

5. The processor-local vectors of each processor are then collated and appended to the input vector’s valsSparse to get the resultant vector’s sparse representation.

This algorithm, while performing better than the previous algorithm, was not scaling well. When checking performance on 1,2,4 and 8 cores, the GFLOPS did not increase proportionately. The speedup of 4 core and 8 core versions over the base SpMSpV implementation is shown in Figure 5.2. As can be observed, the scaling is good only for 8 of the 28 graphs. This can be attributed to the atomic operations while updating the dense representation of the Sparse Vector. This is difficult to characterize as it is heavily dependent on both the input vector and matrix.

5.1.3 Load balanced work-sharing across columns of the Matrix with segmented updates

The final multicore algorithm that was devised was a segment-based implementation. This approach was developed to avoid the use of atomic operations, with the aim of finding another load-balanced way of populating the resultant sparse vector. The first 3 steps of the algorithm are the same as that of the algorithm described
in the previous section, i.e. at the end of step 3, each thread knows exactly which FLOPs it needs to compute. The rest of the algorithm proceeds as follows:

1. The dense representation of the output Sparse Vector is divided into $p$ segments. Each thread maintains an array of $p$ lists, inserting a \((\text{flop-computed-value}, \text{index})\) pair into list $i$ if $\text{index}$ lies in segment $i$ in the dense vector representation. At the end of this step, each process will have an array of $p$ lists, with each array-of-lists containing a total of $F/p$ pairs of elements.

2. The ownership of the different arrays of lists is now flipped, i.e. processor $i$ goes through each element in list $i$ of each of the array-of-lists, and populates the $i$th segment of \texttt{dense_vals} and \texttt{dense_indexes}, as well as \texttt{sparse_indexes} of the resultant vector appropriately.
3. *sparse_vals* is populated in parallel using the elements in *sparse_indexes* and *dense_vals*

In step 2 of the above approach, each processor updates only the segment that it contributes toward. And since no 2 segments overlap, there is no need for any atomic operations.

This algorithm was implemented and run. As can be seen in Figure 5.3, the results were seemingly random, with the common theme being a lack of scalability when it comes to moving from 4-core to 8-core. On delving further as to what the possible reason could be, it was discovered that the number of elements in the flipping of the array-of-lists results in an imbalance of flops. That is, even thought the total number of elements in each array-of-lists is roughly the same, the sum of the number of elements owned by processor $i$ is different from the sum of the number of elements owned by processor $j$ in step 2. This load imbalance degrades performance for several graphs.

**5.2 GPU implementation**

Exploratory attempts were also made to design and implement a GPU algorithm for incremental PageRank. SpMV on the GPU is a very popular research area [2]. However, to the best of our knowledge, there has not been a detailed study of SpMSpV on the GPU. Yang et al [13] do come up with a preliminary algorithm for SpMSpV, which concentrates primarily on the Breadth First Search subroutine. The algorithm devised as part of this thesis used an approach similar to [13] as a basis for our implementation of the SpMSpV algorithm, and is detailed in Algorithm 6.
The most time consuming step in the above algorithm is the SortPairs step on line 4. Hence, 2 types of sorting techniques were experimented with - Merge Sort and Radix Sort. The SpMSpV algorithm was then evaluated on a GTX Titan machine and compared with ModernGPU’s [8] SpMV subroutine. The MergeSort and SegReduce methods were implemented using ModernGPU [8], while Radix Sort was implemented using CUB [5]. As can be seen in Figure 5.4, the performance of SpMSpV using both merge sort and radix sort are poorer than ModernGPU’s optimized SpMV. In general though, radix sort performs better than Merge Sort. This can be attributed to the fact the the indexes that need to be sorted always lie between [0,numRows). As the scope of the GPU algorithm was primarily exploratory in nature, a deeper analysis of the bottlenecks of the SpMSpV implementation was not performed.
Algorithm 6: SpMSpV used to calculate Incremental PageRank on the GPU

**Input**: Sparse Matrix A, Sparse Vector V  
**Output**: Resultant Sparse Vector V

1. procedure SpMSpV (A, V)
   
   /* Gathers a list of indexes in the output vector. Size of ind is the total number of flops */
   2. ind ← Gather(A, V.indexesSparse)
   
   /* Gathers a list of piece-wise values in the output vector. Size of GVal is the total number of flops */
   3. GVal ← Gather(A, ind)
   4. SortPairs(ind, GVal)
   5. for each $j \in ind$ in parallel do
      6. flag[$j$] ← 1
      7. val[$j$] ← GVal[$j$] * valsDense[$j$]
      8. if ind[$j$] = ind[$j-1$] then
         9. flag[$j$] ← 0
      end
   end
   10. V.valsSparse ← SegReduce(val, flag)
   11. return V

---

Figure 5.4: GFLOP/s of SpMV by ModernGPU vs SpMSpV with Merge Sort vs SpMSpV with Radix Sort for input vector with density 10%
Almost 17 years after it was first developed, the PageRank algorithm is still an extremely important metric to measure the quality of a webpage. While it is not the only algorithm that Google uses for web search, it was the first developed and one need look no further than the field of Search Engine Optimization - a field that was spawned out of this algorithm - to judge its pervasiveness. A corollary to the notion of PageRank’s importance, in combination with the fickle nature of web and social network structure is that updating PageRank scores in a constantly changing graph becomes of crucial significance in order for it to remain accurate.

The primary contributions of this thesis was the formulation of this incremental PageRank algorithm in terms of Sparse Matrix Sparse Vector operations as well as a comprehensive analysis of the Sparse Matrix Sparse Vector multiplication operation. The SpMSpV approach was fortified by making smart design choices and algorithmic optimizations such as using a dense-cum-sparse representation for the Sparse Vector, CSC for the Sparse Matrix and updating the vector in-place. Our experimental analysis also shows that in many common power law graphs, SpMSpV outperforms SpMV for vector sparsity densities of upto 35%. Using this heuristic, we implemented a PageRank algorithm that switches between SpMSpV and SpMV in accordance with the vector sparsity density. In addition to these optimizations, we implement a smart
thresholding technique during the SpMSpV operation to ensure that the sparsity density of the resultant vector remains low, to calculate approximate PageRank. High NDCG@K (for K=5000,10000) scores corroborate the efficacy of the approximation techniques. A thorough analysis of it’s efficacy and efficiency was performed on single core, multicore and the GPU, and promising results were obtained.

There are several avenues that can be further explored to drive home the usefulness of the SpMSpV subroutine. A robust analysis of the datasets can provide some insights into specific situations where SpMSpV will outperform SpMV for single core, multicore and GPU architectures. In addition, implementing the SpMSpV subroutine on the WebGraph framework could be a useful endeavor. Overall, in conclusion, it can be said that the SpMSpV operation is a very promising subroutine.
Appendix A: Running Incremental PageRank

To build and run Incremental PageRank, download the codebase and follow the steps below, assuming \$IPR\_HOME is the home directory of the Incremental PageRank code base:

Note that the same code has 2 functions. Firstly is to perform Incremental PageRank, and second is to benchmark SpMV vs SpMSpV vs SpMSpv-Multicore/GPU results. The specific function can be changed by setting/un-setting the DO\_PAGERANK and DO\_BENCHMARK macro flags in the incr\_pagerank.cc file.

1. Compile nlibs, the library that is used to handle the input matrices

   (a) Ensure \$IPR\_HOME/Makefile.in has appropriate values for the CUBHOME and CUSPHOME environment variables, and \$IPR\_HOME/config.mk has appropriate values for all its flags.

   (b) cd \$IPR\_HOME/nlibs/

   (c) make

2. Compile the Incremental PageRank code

   (a) cd \$IPR\_HOME/Multicore (or Serial\_v2 or GPU, based on your requirement)
(b) make

3. Run the code

(a) cd $IPR_HOME/Multicore (or Serial_v2 or GPU, based on your requirement)

(b) ./run.sh

Note that ./run.sh contains 2 parameters that can be changed, as required. One is the value of OMP_NUM_THREADS (only in case of Multicore) and the other is the list of input matrices.

Also note that after running, a folder is generated for each input file. Inside each folder, there are several files generated by the program. For instance, the <input_matrix>.mtx.orig.outbenchmark.csv contains SpMV-SpMSpV benchmark results if the DO_BENCHMARK flag is set in the code.
Bibliography


