2008

A computational study of sparse matrix storage schemes

Haque, Sardar Anisul

Lethbridge, Alta. : University of Lethbridge, Deptartment of Mathematics and Computer Science, 2008

A COMPUTATIONAL STUDY OF SPARSE MATRIX STORAGE SCHEMES

SARDAR ANISUL HAQUE
Bachelor of Science, Islamic University of Technology, 2002

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

I dedicate this thesis to my parents.

Abstract

The efficiency of linear algebra operations for sparse matrices on modern high performance computing system is often constrained by the available memory bandwidth. We are interested in sparse matrices whose sparsity pattern is unknown. In this thesis, we study the efficiency of major storage schemes of sparse matrices during multiplication with dense vector. A proper reordering of columns or rows usually results in reduced memory traffic due to the improved data reuse. This thesis also proposes an efficient column ordering algorithm based on binary reflected gray code. Computational experiments show that this ordering results in increased performance in computing the product of a sparse matrix with a dense vector.

Acknowledgments

I express my deep acknowledgment and profound sense of gratitude to my supervisor Dr. Shahadat Hossain, Associate Professor, University of Lethbridge for his inspiring guidance, helpful suggestions and persistent encouragement as well as close and constant supervision throughout the period of my Masters Degree.

I would also like to thank my M.Sc. co-supervisor Dr. Daya Gaur, Associate Professor, University of Lethbridge for his guidance and suggestion.

I would also like to thank my M.Sc. supervisory committee member Dr. Saurya Das, Associate Professor, University of Lethbridge for his guidance and suggestion.

I am grateful to Dr. Shahadat Hossain, Dr. Daya Gaur and the School of Graduate Studies for the financial assistantships. I thank all the staff, and my colleagues at the University of Lethbridge for their helpful nature and co-operation.

I am very much thankful to my family and fellow graduate students Shafiq Joty, Salimur Choudhury, Mohammad Islam, Mahmudul Hasan and Minhaz Zibran for the continuous encouragement that helped me to complete this thesis.

I would also like to thank my wife, Tanzia Rouf Tuie for her support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Sparse matrix-vector multiplication ($y = Ax$) is an important kernel in scientific computing. For example, in the conjugate gradient method (iterative solver for system of linear equations) the main computational step in each iteration is to calculate the product Ax, where A is a large and sparse matrix and x is a dense vector. The performance of such a solver depends on the efficient implementation of sparse matrix-vector multiplication. Though the total number of arithmetic operations (involving nonzero entries only) to compute Ax is fixed, reducing the probability of cache misses per operation is still a challenging area of research in modern superscalar architecture. Typically, it involves reordering of columns or rows so that we can access the elements of both x and y in a regular way to improve temporal and spatial locality. This preprocessing is done once and its cost is amortized by repeated multiplications (as in, for example, conjugate gradient type algorithms). The efficiency of sparse matrix-vector multiplication on modern high performance computing system is often constrained by the available memory bandwidth [12]. Large volume of load operations from memory compared to the floating point operations, indirect access to the data, and loop overhead are perceived as main computational challenges in efficient implementation of sparse matrix operations [34].

Data structures used for storing sparse matrices have an array for storing its elements along with some other auxiliary arrays. The purpose of these auxiliary arrays is to keep the

information of row and column indices for each nonzero element. Different data structures use different methods for keeping row and column index information. So, the code for sparse matrix-vector multiplication needs to be modified accordingly.

Irregular access of the elements in vector x may cause a large number of cache misses. Each of the nonzeros in A is used only once during multiplication. So, we want to reorder A in such a way that the nonzeros of each row are consecutive. By this, we can improve spatial locality in accessing x. Furthermore, reusing the elements of x improves temporal locality. We can achieve this if the nonzeros along all the columns are consecutive.

Below, we describe conjugate gradient method as an example, where efficient implementation of sparse matrix-vector multiplication is crucial for the overall performance of the method. We then describe the contributions of this thesis, followed by the thesis outline.

1.1 Conjugate Gradient Method

Hyperlink matrix P is a sparse matrix generated by the hyperlink structure of the world wide web. Here P is a square matrix and its dimension is equal to n_p, the number of *pages* in a web. P is defined as follows

$P_{ij} = 1/|O_i|$, if there exists a hyperlink from page i to page j, and 0 otherwise. The scalar $|O_i|$ is the number of outlinks from page i.

The sparse linear system formulation of the *Google PageRank problem* [22] is

$x^T A = b^T$, where $A = (I - \alpha P)$ and $0 < \alpha < 1$.

Here b^T is the *personalization* or *teleportation* vector and x^T is the unknown *PageRank* vector. An efficient way to solve this exteremely large and sparse linear system is to apply a conjugate gradient type method such as GMRES [31].

The conjugate gradient method is an iterative method to obtain numerical solution of system of linear equations $Ax = b$. Here, A is a symmetric and positive definite matrix

and it is usually a large sparse matrix. This method iteratively produces an approximate solution $x^{(i)}$ for the system. The equation for updating $x^{(i)}$ is: $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$, where α_i is a scalar, and $p^{(i)}$ is the search direction vector, respectively. The conjugate gradient algorithm is given in Algorithm 1.

Algorithm 1 Conjugate gradient algorithm [31].

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$;
for $i \leftarrow 1, 2, \ldots$ **do**
    $\rho_{i-1} = r^{(i-1)^T} r^{(i-1)}$;
    if $i = 1$ **then**
        $p^{(1)} = r^{(0)}$;
    else
        $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$;
        $p^{(i)} = r^{(i-1)} + \beta_{i-1} p^{(i-1)}$;
    end if
    $q^{(i)} = Ap^{(i)}$;
    $\alpha_i = \rho_{i-1} / p^{(i)^T} q^{(i)}$;
    $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$;
    $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$;
    check convergence; continue if necessary
end for

In this iterative process, A remains unchanged and we need to multiply it with the search direction vector $p^{(i)}$. This sparse matrix-vector multiplication ($Ap^{(i)}$) is the most computationally expensive step in each iteration. Also the method may require a good number of iterations before convergence. The running time of this method is proportional to the computing time of sparse matrix-vector multiplication ($Ap^{(i)}$). One good way of making this method efficient is to preprocess the sparse matrix A in such a way that it can be multiplied with a vector efficiently. Though the computational cost of preprocessing A can be high, this cost can be amortized by a huge number of repeated multiplications.

1.2 Contributions

The main scientific contributions of this thesis are as follows:

We propose one new column ordering algorithm for improving the performance of sparse matrix-vector multiplication. We propose column ordering algorithm based on *binary reflected gray code* for sparse matrices, which exhibits good data locality during sparse matrix-vector multiplication. To the best of our knowledge our proposal is the first to consider gray codes for column ordering in sparse matrix-vector multiplication. We call it *binary reflected gray code ordering* or BRGC ordering algorithm.

We implement three other column ordering algorithms from literature: *column intersection ordering*, *local improvement ordering* and *similarity ordering*. *column intersection ordering* and *local improvement ordering* are implemented based on the constructive and improvement heuristics for *TSP problem* described in [29] respectively. *Similarity ordering* is implemented based on a *distance measure* function described in [16].

The common objective of *column intersection ordering*, *local improvement ordering*, and *similarity ordering* algorithms is to permute the columns of a sparse matrix A in such a way that the nonzeros of each row are placed consecutively as much as possible. Thus the spatial locality in accessing x is improved resulting performance improvement in sparse matrix-vector multiplication (Ax). But these column ordering algorithms do not try to improve the temporal locality in accessing x. *Binary reflected gray code ordering* improves both temporal and spatial locality of x.

For our experiment, we implement three storage schemes for sparse matrices from literature: *compressed row storage*, *fixed-size block storage*, and *block compressed row storage*.

We carry on our experiments on three different computing platforms. We take test matrices from [5] for our experiment. Each of our test matrices is preprocessed by *column intersection ordering*, *similarity ordering*, *local improvement ordering*, and *binary reflected gray code ordering* algorithms separately.

The performances of sparse matrix-vector multiplication on three different computing platforms by *compressed row storage*, *fixed-size block storage*, and *block compressed row*

storage schemes are reported. We also show the performance comparison among these four column ordering algorithms for all three storage schemes on each platform separately. We also provide an analysis of the performance comparison for each platform.

Based on our experimental results, we find FSB storage scheme performs the best among the storage schemes on the three computing platforms. We also find that BRGC ordering improves both the spatial and the temporal locality in accessing the elements of x during sparse matrix-vector multiplication. This ordering performs the best on two of our computing platforms.

1.3 Thesis Outline

This thesis contains six chapters including this introductory chapter. Below we provide a short description of the remaining chapters:

In Chapter 2, we present background materials relevant to our thesis. Our discussion includes a brief description of computer system memory hierarchy, and computational complexity theory.

Chapter 3 contains description of storage schemes for sparse matrices from literature. We also present two new storage schemes for sparse matrices. The discussion on the implementation of these storage schemes is illustrated by the sparse matrix-vector multiplication code.

In Chapter 4, we first present some of the well-known column ordering algorithms for sparse matrices from literature followed by the description of our proposed column ordering algorithms. We provide complexity analysis of these ordering algorithms.

Chapter 5 presents experimental results that demonstrate the performance of column ordering algorithms described in Chapter 4 on different storage schemes of sparse matrices. All of the experiments are carried on three different computing platforms. We feature the

architectural specifications of these computing platforms. We also describe the method of performance analysis.

Finally, in Chapter 6, we provide concluding remarks and direction for future research in this area.

Chapter 2

Background

This chapter contains background material on memory organization of modern computer systems, and an introduction to computational complexity theory.

2.1 Hierarchical Memory Systems

The description of memory organization of modern computer systems of this section is from [14] [26] [15]. Main memory and cache memory are two important memory components in a computer system. A cache is a small and fast memory component located close to the microprocessor. *Latency* of a memory component and *memory bandwidth* are two related issues for understanding the importance of cache. *Latency* of a memory component is the elapsed time between initiating a request for a data until it is retrieved. *Memory bandwidth* refers to the number of bytes that can be read or transferred from a memory component by processor in one unit of time. As a computer system has memory components with different speeds, latency and bandwidth differ for different components accordingly. Main memory is slower than the microprocessor speed. If a microprocessor has to rely on main memory alone for its execution then it is idle while transfer of data from main memory takes place. Cache plays a vital role in this scenario. Microprocessors can access cache much faster than main memory. Cache keeps a copy of most frequently used data and supply it to microprocessor whenever necessary.

Cache memory can be classified into two groups:

1. Data Cache.

2. Instruction Cache

We limit our discussion to data cache only. Whenever microprocessor requires any data from memory, it looks into cache for a copy of that data. A *hit* in cache happens if the microprocessor gets the requested data from the cache. Otherwise it is called a *miss*. The effectiveness of a cache system is called *hit rate* which is measured by the ratio of total number of *hits* and the total number of access in cache.

In a hierarchical memory organization, cache and main memory are treated as higher level and lower level memory, respectively. A memory component in higher level is faster, smaller and more expensive than those of a lower level. In modern computer system there can be more than one level of caches called Level 1 (L1), Level 2 (L2) and so on. In a multi level cache system, Level(i) cache is treated as a higher level memory component than any cache at level$(i+1)$. There can be more than one cache in one level. For example, one for data and the other for instruction. Sometimes level(1) cache exists on the microprocessor chip. A typical block diagram of hierarchical memory system consisting of two levels of cache is shown in Figure 2.1.

2.1.1 Principle of Locality

The principle of locality states that most programs do not access their code and data uniformly. There are mainly two types of locality:

1. Spatial locality: It refers to the observation that most programs tend to access data sequentially.

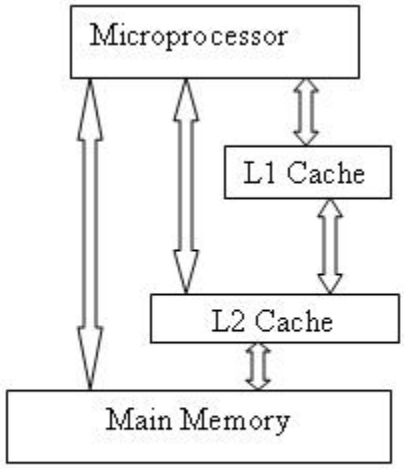

Figure 2.1: A typical block diagram of hierarchical memory systems with two levels of cache. The arrow signs represent data flow among different memory components and microprocessor.

2. Temporal locality: It refers to the observation that most programs tend to access data that was accessed previously.

When a *miss* occurs, a data block of consecutive memory locations consisting the required data from main memory or another cache is copied into the cache. This technique of copying a data block of consecutive memory locations follows the principle of spatial locality. The size of this block of consecutive memory is called a *cache line*. When there is no place for a new data block in cache, it has to replace any existing data block from the cache. This is called *replacement policy*. When a block of memory is copied into cache, it remains there until any replacement occurs in its place.

2.1.2 Cache Miss

We can categorize all cache misses into the following three types:

1. Compulsory misses: It happens when a data block is accessed for the first time.

2. Capacity misses: This type of miss happens because of the small size of cache. For example, a cache may not contain all the data blocks that a program needs during execution.

3. Conflict misses: This type of miss occurs when microprocessor is looking for a data block in cache which was replaced earlier from the cache (according to some replacement policy).

2.1.3 Mapping Function and Replacement Policy

Let, *CacheSize*, *CacheLine*, *MemorySize* be the size of the cache, cache line (data block) and memory respectively in bytes. Then the number of data blocks, *CacheBlock*, in the cache is equal to $CacheSize/CacheLine$. Each data block in main memory has an address *BlockAddressInMemory*. Again, a data block will get an address *BlockAddressInCache*, whenever it is copied into cache. A mapping function defines where a data block from main memory is copied in cache. Least recently used (LRU) and random replacement(RR) are two broadly used replacement policies.

Cache can be classified into three main categories based on the mapping function used: direct mapped cache, fully associative cache and set associative cache. The description of these three categories along with their corresponding replacement policies are given below.

1. Direct mapped cache: If each data block from main memory has one fixed place in cache then the cache is called direct mapped. As the cache size is smaller than that of main memory, many data blocks have same place in the cache. As a result the replacement policy is very simple. It simply discard the data block from the cache (if it contains any data). Direct mapped, the mapping function of this type of cache is usually expressed as:

   $BlockAddressInCache = BlockAddressInMemory \bmod CacheBlock$.

2. Fully associative cache: If a data block from main memory can be placed anywhere in the cache then the cache is called fully assosiative. In this type of cache, replacement any data block from cache is required only when the cache is full. Both LRU and RR can be used as the replacement policy for this type of cache.

3. N-way set associative cache: An N-way set associative cache is divided into a number of sets. The size of each set is equal to $N \cdot cacheLineSize$, where *cacheLineSize* is the size of the cache line. The total number of sets $TotalSets$ is equal to $CacheBlock/N$. A data block from main memory can be placed in any of the N places of a fixed set $SelectedSet$.

   $SelectedSet = BlockAddressInMemory \bmod TotalSets$.

   So replacement is required when all the cache lines in that set is full. Both LRU and RR can be used as the replacement policy for this type of cache.

If any data is modified by the execution of microprocessor and if that data is in cache then it must be modified accordingly in cache. In some architectures, this update is done in cache only while in the others updates are performed in both cache and main memory.

2.2 Computational Complexity

A *polynomial-time* algorithm can be defined to be one whose time complexity function is $O(p(k))$ for some polynomial function p of k, where k is denoted the *input length* [4]. Problems that are solvable by polynomial-time algorithm are considered as being *tractable*.

A *decision problem* is one whose answer is either "yes" or "no". We call an instance of a decision problem as *positive instance* (*negative instance*) if it has "yes" ("no") answer.

A decision problem Π is in class P if it can be solved on a *deterministic Turing machine* in polynomial time. A decision problem Π is in class NP if it can be solved on a *nondeterministic Turing machine* in polynomial time [9].

If we can transform the instances of a decision problem to the instances of another decision problem such that positive (negative) instances of the first decision problem are mapped to positive (negative) instances of the other decision problem in polynomial time then we call this a *polynomial time reduction*. Suppose a problem Π is polynomial time reducible to another problem Π' then we denote it as $\Pi \leq_p \Pi'$.

A decision problem Π is in class $NP-complete$ if

1. $\Pi \in NP$, and

2. $\Pi' \leq_p \Pi$ for every $\Pi' \in NP$ [4].

A *combinatorial optimization problem* is either a "minimization problem" or a "maximization problem". For each instance I of a *combinatorial optimization problem*, there exists a finite set $S(I)$ of candidate solutions for I. A function f is called a "solution value" for each candidate solution if it assigns to each instance and each candidate solution a rational number. In a minimization (maximization) problem, an optimal solution for an instance I is a candidate solution σ_* such that for all possible candidate solution, σ_* has the minimum (maximum) solution value.

The optimization version of the decision problems in the class NP-Complete belong to the class NP-hard i.e. a problem is considered as hard as NP-Complete. The class of NP-Complete and NP-Hard are regarded as *intractable* because problems in these classes have no known polynomial time algorithms.

Chapter 3

Storage Scheme for Sparse Matrices

There is no definite relationship between sparse matrix dimensions and the number of nonzeros. J. H. Wilkinson gave an informal working definition of a sparse matrix as *"any matrix with enough zeros that it pays to take advantage of them"* [11].

In the following section, we will describe some storage schemes for sparse matrices. We use the sparse matrix A given in Figure 3.1 for describing different storage schemes. In this thesis m, n, and nnz represent the row dimension, the column dimension, and the number of nonzeros of sparse matrix A, respectively.

For simplicity we consider nonzeros of sparse matrix A are stored as *double* data types and elements of all other auxiliary data structures are stored as *integer* data types. We also assume that eight *bytes* and four *bytes* are required to store a double and an integer data respectively.

3.1 Different Storage Schemes

A banded matrix is a matrix that has all its nonzeros near the diagonal. If A is a banded matrix and a_{ij} is a nonzero of A then $i-m_l \leq j \leq i+m_u$, where m_l and m_u are two nonnegative integers. The number m_l+m_u+1 is called the bandwidth of A [31]. Sparse matrix with regular sparsity pattern can be represented by minimum storage. For example, a banded sparse matrix (with small bandwidth) can be stored by a number of arrays. Each of the

$$A = \begin{pmatrix} 0 & 0 & a_{02} & a_{03} & 0 \\ a_{10} & 0 & 0 & 0 & a_{14} \\ 0 & a_{21} & 0 & a_{23} & 0 \\ a_{30} & 0 & a_{32} & 0 & a_{34} \\ 0 & 0 & a_{42} & a_{43} & 0 \end{pmatrix}$$

Figure 3.1: Sparse matrix A.

arrays represents a diagonal. On the other hand, a matrix with irregular sparsity pattern needs auxiliary storage. We assume no specific a priori knowledge of sparsity pattern for this thesis.

Compressed row storage (CRS), *compressed column storage (CCS)*, *fixed-size block storage (FSB)*, *block compressed row storage (BCRS)*, and *jagged diagonal storage (JDS)* are some of the well-known data structures for sparse matrices with no particular sparsity pattern assumed [1]. Among these data structures *JDS* is suitable for vector processors [8]. Hossain proposed *compressed column block storage (CCBS)* [18] and *bi-directional jagged diagonal storage (Bi-JDS)* [17] data structures for superscalar architecture and vector processors respectively. *Compressed diagonal storage (CDS)* [13] is another sparse data structure, which is suitable for banded matrices.

3.1.1 Compressed Row Storage (CRS)

This is the most common storage scheme for sparse matrices [1]. Three arrays are used in this storage scheme to store sparse matrix A:

1. $value$: for storing the nonzeros of A row-by-row.

2. $colind$: for storing the column index of each nonzero.

3. $rowptr$: for storing the index of the first nonzero of each row in $value$ array.

Table 3.1: CRS data structure for sparse matrix A.

value	a_{02}	a_{03}	a_{10}	a_{14}	a_{21}	a_{23}	a_{30}	a_{32}	a_{34}	a_{42}	a_{43}
colind	2	3	0	4	1	3	0	2	4	2	3
rowptr	0	2	4	6	9	11					

The data structures to store the example matrix A under this scheme is shown in table 3.1. Sample code for computing $y = Ax$ under this storage scheme is given in Algorithm 2. The memory requirement of this scheme is $8nnz + 4(nnz + m + 1)$ bytes.

Algorithm 2 Sparse matrix vector multiplication for CRS scheme [1].

```
1: for i ← 0 to m − 1 do
2:     for k ← rowptr[i] to rowptr[i + 1] − 1 do
3:         j ← colind[k];
4:         y[i] += value[k] ∗ x[j];
5:     end for
6: end for
```

According to the sparse matrix-vector multiplication code under this storage scheme (in Algorithm 2), it requires a load operation (for loading column indices) and an indirect memory access (for accessing elements in x) for each scalar multiply-add operation. So, it is expected that the performance of the sparse matrix-vector multiplication is restricted by these operations. If the nonzeros in each row of A have very sparse column indices, the access to the elements in x is highly irregular which results in poor spatial locality. If the nonzeros along all the columns are consecutive then sparse matrix-vector multiplication code can reuse elements of x, thus improving temporal locality of x.

Loop unroll and jam are two code optimization techniques to reduce loop overhead, memory load operations and increase register exploitation. In loop unroll the computations in more than one iterations of a loop are merged in one iteration. And in loop jam two or more loops are merged in a single loop. *Row length* of a row i is defined as the number of nonzeros in row i. Mellor-Crummey and Garvin in [24] proposed *Length-grouped CSR (L-CSR)* scheme for sparse matrices. In this storage scheme the nonzeros of rows having

Table 3.2: CCS data structure for sparse matrix A.

value	a_{10}	a_{30}	a_{21}	a_{02}	a_{32}	a_{42}	a_{03}	a_{23}	a_{43}	a_{14}	a_{34}
rowind	1	3	2	0	3	4	0	2	4	1	3
colptr	0	2	3	6	9	11					

equal row length are kept together in an array. Like CRS, it also has another array to store the column indices of corresponding nonzeros. It also keep the row lengths and row indices in two other arrays. The multiplication code for this storage scheme is same as CRS except we can use loop unroll and jam. L-CSR scheme is best suited for sparse matrices that have small and predictable row lengths.

3.1.2 Compressed Column Storage (CCS)

This scheme is same as *CRS* except that the nonzeros are stored column-by-column [1]. Like *CRS*, three arrays are used in this storage scheme to store sparse matrix A:

1. *value*: for storing the nonzeros of A column-by-column.

2. *rowind*: for storing the row index of each nonzero.

3. $colptr$: for storing the index of the first nonzero of each column in *value* array.

The data structures to store the example matrix A under this storage scheme is shown in table 3.2. Sample code for computing $y = Ax$ under this storage scheme is given in Algorithm 3. The memory requirement of this scheme is $8nnz + 4(nnz + n + 1)$ bytes.

Algorithm 3 Sparse matrix vector multiplication for CCS scheme [1].

```
1: for i ← 0 to n − 1 do
2:     for k ← colptr[i] to colptr[i + 1] − 1 do
3:         j ← rowind[k];
4:         y[j] += value[k] ∗ x[i];
5:     end for
6: end for
```

The sparse matrix-vector multiplication code under this storage scheme (Algorithm 3), it requires a load operation (for loading row indices) and an indirect memory access (for accessing elements in y) for each scalar multiply-add operation. So, it is expected that the performance of the sparse matrix-vector multiplication is restricted by these operations. If the nonzeros in each column of A have very sparse row indices, the access of y is highly irregular which results in poor spatial locality. If the nonzeros along all the rows are consecutive then sparse matrix-vector multiplication code can reuse elements of y, thus improving temporal locality of y.

3.1.3 Fixed-size Block Storage Scheme (FSB)

Toledo in [34] proposed this storage scheme for sparse matrices. We define a *nonzero block* as a sequence of $l \geq 1$ contiguous nonzero elements in a row. For the purpose of demonstration, we will describe block storage scheme of fixed size $l = 2$. According to this scheme, the given sparse matrix A is expressed as a sum of two matrices A_1 and A_2; A_1 stores all the nonzero blocks of size 2 and A_2 stores the rest (in *CRS* scheme). If a sparse matrix is stored by this storage scheme then we can use loop unrolling during multiplication of the nonzeros in A_1 with x. By this we can reduce the number of load instruction during computing Ax [34]. We will denote this storage scheme by *FSBl*, where the last character l represents the length of the nonzero block. For example, *FSB2* represents fixed-size block storage scheme of length 2.

The data structure for A_1 is similar to *CRS* with one difference: the column index of the first element of each nonzero block is stored. Like *CRS* three arrays are used to store A_2:

1. $value$: for storing the nonzeros of A_2 row-by-row.
2. $colind$: for storing the column index of each nonzero of $A2$.
3. $rowptr$: for storing the index of the first nonzero of each row in $value$ array.

$$A_1 = \begin{pmatrix} 0 & 0 & a_{02} & a_{03} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a_{42} & a_{43} & 0 \end{pmatrix}$$

Figure 3.2: Sparse matrix A_1 for FSB2.

$$A_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ a_{10} & 0 & 0 & 0 & a_{14} \\ 0 & a_{21} & 0 & a_{23} & 0 \\ a_{30} & 0 & a_{32} & 0 & a_{34} \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3.3: Sparse matrix A_2 for FSB2.

We need three more arrays to store A_1:

1. *valueBl*: for storing the nonzeros of A_1 row-by-row.

2. *colindBl*: for storing the column index of the first nonzero of each nonzero block in A_1.

3. *rowptrBl*: for storing the index of the first nonzero of each row of A_1 in *valueBl* array.

The A_1 and A_2 matrices of the example sparse matrix A are given in Figure 3.2 and Figure 3.3 respectively. Tables 3.3 and 3.4 display the data structures under this scheme. Sample code for computing of $y = Ax$ under this scheme is given in Algorithm 4.

Denote by β_p the number of nonzero blocks of size p. Let *maxNonzeroBlock* be the size of the largest nonzero block in matrix A in a given column order. The memory requirement of this scheme is $8nnz + 4(2m + 2 + \sum_{r=1}^{maxNonzeroBlock}((\lfloor r/l \rfloor + r \bmod l)\beta_r))$ bytes.

To improve data locality in accessing x and reduce loop overhead, we multiply A_1 and A_2 with x in one loop as shown in Algorithm 4. The sparse matrix-vector multiplication code under this storage scheme (Algorithm 4) requires a load operation (for loading column

Table 3.3: FSB2 data structure for sparse matrix A_1.

valueBl	a_{02}	a_{03}	a_{42}	a_{43}		
cloindBl	2	2				
rowptrBl	0	1	1	1	1	2

Table 3.4: FSB2 data structure for sparse matrix A_2.

value	a_{10}	a_{14}	a_{21}	a_{23}	a_{30}	a_{32}	a_{34}
colind	0	4	1	3	0	2	4
rowptr	0	0	2	4	7	7	

Algorithm 4 Sparse matrix vector multiplication for FSB2 scheme.

1: **for** $i \leftarrow 0$ to $m-1$ **do**
2: **for** $k \leftarrow rowptrBl[i]$ to $rowptrBl[i+1]-1$ **do**
3: $j \leftarrow colindBl[k]$;
4: $l \leftarrow 2*k$;
5: $y[i] += valueBl[l]*x[j]$;
6: $y[i] += valueBl[l+1]*x[j+1]$;
7: **end for**
8: **for** $k \leftarrow rowptr[i]$ to $rowptr[i+1]-1$ **do**
9: $j \leftarrow colind[k]$;
10: $y[i] += value[k]*x[j]$;
11: **end for**
12: **end for**

indices) and an indirect memory access (for accessing elements in x) for l scalar multiply-add operations for computing A_1x. Furthermore, it requires a load operation (for loading column indices) and an indirect memory access (for accessing elements in x) for a scalar multiply-add operation in computing A_2x. Again if the nonzeros along all the columns of A are consecutive then sparse matrix-vector multiplication code can reuse elements of x, thus improving temporal locality of x.

A $(k \times l)$ *block* of a sparse matrix A is a submatrix of A of dimension $(k \times l)$, where all the elements in the submatrix are nonzeros. A block of (1×2) is same as a nonzero block of length 2. In [34], sparse matrix A is stored as a sum of differently blocked matrices. For example, one stores all blocks of (2×2), one stores all blocks of (1×2), and the third one with the remaining elements.

A $(k \times l)$ *tile block starting at position* (i, j) of a sparse matrix A is a submatrix of A of dimension $(k \times l)$, where at least one element in the submatrix is nonzero, where $0 \leq i < m$ and $0 \leq j < n$. Vuduc in [35] and Im, Yelick and Vuduc in [20] proposed to store A by a number of *tile blocks* of $(k \times l)$, where $i \bmod k = j \bmod l = 0$ for all tile blocks. Buttari, Eijkhout, Langou and Filippone in [2] proposed to store A by a number of *tile blocks* of $(k \times l)$, where $i \bmod k = 0$ for all tile blocks. Vuduc and Moon in [36] proposed to store A into a sum, $A = A_1 + A_2 + \ldots + A_s$, where each A_i stores tile blocks of a particular dimension. The choice of the dimensions of tile blocks in any of these techniques depend on both the computing platform and the type of sparse matrices. In [2], a computational method for finding appropriate k and l for a computing platform is suggested. Im in [19] and Vuduc in [35] proposed to apply register blocking and cache blocking in sparse matrix-vector multiplication if the sparse matrix is stored in a fixed dimension *blocks* or *tile blocks*. The applicability of cache blocking in computing Ax is described in [25].

Table 3.5: BCRS data structure of A.

value	a_{02}	a_{03}	a_{10}	a_{14}	a_{21}	a_{23}	a_{30}	a_{32}	a_{34}	a_{42}	a_{43}
colind	2	0	4	1	3	0	2	4	2		
nzptr	0	2	3	4	5	6	7	8	9	11	
rowptr	0	1	3	5	8	9					

3.1.4 Block Compressed Row Storage (BCRS)

Pinar and Heath in [29] proposed this storage scheme. In this storage scheme, the multiplications of the nonzeros of each nonzero block are done in the inner loop. We need one load instruction to load the column index of the first nonzero of each nonzero block. So the total number of load instructions is reduced. As a sparse matrix can have nonzero blocks of different sizes, we can not use loop unrolling in the code for computing Ax under this storage scheme. Four arrays are required to represent a sparse matrix in BCRS scheme:

1. *value*: for storing the nonzeros of A row by row.
2. *colind*: the column index of first element of each nonzero block.
3. *nzptr*: the index of the first nonzero of each nonzero block in *value* array.
4. *rowptr*: the index of first nonzero block in each row in *nzptr* [29].

The data structure in *BCRS* scheme is illustrated in Table 3.5 and sample code for computing $y = Ax$ under this scheme is given in Algorithm 5. Let β be the total number of nonzero blocks in A then memory requirement of this scheme is $8nnz + 4(2\beta + m + 2)$ bytes.

The sparse matrix-vector multiplication code under this storage scheme (Algorithm 5) requires a load operation (for loading the column indices of the first nonzeros of all nonzero blocks) and an indirect memory access (for accessing elements in x) for l multiply-add operations, where l represents the length of a nonzero block. Fewer number of nonzero

Algorithm 5 Sparse matrix vector multiplication for BCRS scheme [29].

```
1: for i ← 0 to m − 1 do
2:     for j ← rowPtr[i] to rowPtr[i + 1] − 1 do
3:         startcol ← colind[j];
4:         t ← 0;
5:         for k ← nzPtr[j] to nzPtr[j + 1] − 1 do
6:             y[i] += value[k] ∗ x[startcol + t];
7:             t ← t + 1;
8:         end for
9:     end for
10: end for
```

blocks means less loop overhead per multiplication. However, the problem of ordering columns of A to get minimum number of nonzero blocks is in NP-Hard [29]. Again if the nonzeros along all the columns are consecutive then sparse matrix-vector multiplication code can reuse elements of x, thus improving temporal locality of x.

3.1.5 Compressed Column Block Storage (CCBS)

Hossain [18] proposed this storage scheme. Three arrays ($value$, $brind$, and $brptr$) and one integer ($ncol$) are used in this storage scheme to store sparse marix A.

Two rows of a matrix are said to be *orthogonal* if they do not both have nonzero elements in the same column position. In this scheme, first the rows of A are partitioned into a number ($ncol$) of row groups such that the rows in each group are orthogonal. Then the nonzeros of each orthogonal row group are "packed" in an array of size n such that a nonzero a_{ij} of A is placed at j^{th} index of that array. All entries of that array may not be occupied by nonzeros. So we will fill the empty places by zeros. Then these arrays are merged one after another and stored in $value$ array. So the size of $value$ is $(n \cdot ncol)$.

Let $value[i]$ and $value[j]$ be two nonzeros of array $value$ from row r and s respectively such that $j > i$ and all elements in $\{value[i+1], \ldots, value[j-1]\}$ are zeros. We treat these zeros as elements of either row r or s. If $value[i]$ and $value[j]$ are elements from

Table 3.6: CCBS data structure of A considering orthogonal row groups: {{0, 1} {2, 3} {4}}.

value	a_{10}	0	a_{02}	a_{03}	a_{14}	a_{30}	a_{21}	a_{32}	a_{23}	a_{34}	0	0	a_{42}	a_{43}	0
brind	1	0	1	3	2	3	2	3	4						
brptr	0	2	4	5	6	7	8	9	10	15					

the same row r then we treat these zeros as elements of row r. An *elementary block* of length k in *value* array is defined as a set of contiguous elements (both zeros and nonzeros) $\{value[i],\ldots,value[i+k-1]\}$ such that all elements of $\{value[i],\ldots,value[i+k-1]\}$ come from the same row r of A and both $value[i-1]$ and $value[i+k]$ are from other rows.

The row indices of all elementary blocks are stored in *brind* array. The indices of the first elements of all elementary blocks in *value* array are stored in *brptr* array [18]. Sometimes, total number of elementary blocks in *value* can be less than total number of nonzero blocks in A. Because two or more nonzero blocks separated by some zeros of a row in A can be merged as an elementary block in *value*.

Algorithm 6 Sparse matrix vector multiplication for CCBS scheme [18].

```
1:  blk ← 0;
2:  for l ← 0 to ncol − 1 do
3:      j ← 0;
4:      while j ≤ n − 1 do
5:          i ← brind[blk];
6:          for k ← brptr[blk] to brptr[blk + 1] − 1 do
7:              y[i]+ = value[k] ∗ x[j];
8:              j ← j + 1;
9:          end for
10:         blk ← blk + 1;
11:     end while
12: end for
```

Partitioning the rows to a minimum number of orthogonal row groups is NP-hard problem [3]. Ideally, we would prefer $ncol \approx \lceil nnz/n \rceil$, though *ncol* can not be less than $MAX(\rho_0,\ldots,\rho_{n-1})$, where ρ_i is the number of nonzeros in column i. In practice, *ncol* is often so big that it is impractical to store A in *value* array under this storage scheme in

computer memory.

Algorithm 7 Orthogonal row partitioning of A for CCBS scheme.

```
1:  B is a row permuted matrix of A;
2:  {r_0, r_1, ... r_{m-1}} are the rows of B such that size[i] ≥ size[i+1];
3:                                          ▷ size[i] is the number of nonzeros in r_i
4:  flag is an array of size m. Initially all elements of flag are zero;
5:  mask is an array of size n. Initially all elements of mask are zero;
6:  rowGroups is a two dimensional array;
7:                              ▷ rowGroups[i] stores the row indices of one orthogonal row group i
8:  processRow = 0;
9:  ncol = 0;
10: l = 0;
11: while processRow < m do
12:     j = −1;
13:     for i ← 0 to m − 1 do
14:         if flag[i] = 0 and r_i is orthogonal with mask then
15:             j = i;
16:             break;
17:         end if
18:     end for
19:     if j > −1 then
20:         copy the nonzeros of r_j in mask such that a nonzero b_jk of r_i is copied in mask[k];
21:                                                          ▷ b_jk is a nonzero of B
22:         rowGroups[ncol][l] = j;
23:         processRow = processRow + 1;
24:         l = l + 1;
25:         flag[j] = 1;
26:     else
27:         ncol = ncol + 1;
28:         l = 0;
29:         update all entries of mask as zero;
30:     end if
31: end while
```

Let $\{r_0, r_1, \ldots r_{m-1}\}$ be the rows of A and $\{r_0^j, \ldots r_k^j\}$ be the rows of a orthogonal row group j. We choose to append this orthogonal row group by inserting row r_i in it if

1. r_i is orthogonal to all rows in the current orthogonal row group $\{r_0^j, \ldots r_k^j\}$.

2. r_i is not in any other orthogonal row group.

3. there is no rows r_p (which is not in any orthogonal row group) such that the number of nonzeros in r_p is greater than that of r_i.

This procedure is shown in Algorithm 7. On termination of this algorithm $rowGroups[i]$ stores the row indices of an orthogonal row group i. After orthogonal row partitioning of A, we permute rows of A in such a way that the rows in a orthogonal row group are placed together.

The data structure of A in *CCBS* scheme is illustrated in Table 3.6 and sample code for computing $y = Ax$ under this storage scheme is given in Algorithm 6. The memory requirement of this scheme is $8n \cdot ncol + 4(2\beta' + 2)$ bytes (where β' is the total number of elementary blocks in *value*).

Under this storage scheme (Algorithm 6) access to both x and *value* are regular; while the accesses to y are not regular. But the temporal locality of y is improved because the rows of each orthogonal row group of A are together.

3.1.6 Compressed Column Block Storage With Compreesed Row Storage (CCB-SWCRS)

This storage scheme is composed of *CCBS* and *CRS*. As mentioned in the description of *CCBS* scheme that the number of orthogonal row groups in *CCBS* is often so big that it is impractical to store A in *value* array in computer memory. So we combine *CCBS* and *CRS* schemes to store sparse matrix A in this storage scheme. In this storage scheme A is expressed as a sum of two matrices A_1 and A_2.

Let $ORG = \{org_0, org_1, \ldots org_{ncol-1}\}$ be the orthogonal row groups of A found by the heuristic in Algorithm 7 and let θ_i be the number of nonzeros in the orthogonal row group org_i. Let *threshold* be a real number between 0 and 1. We can partition these orthogonal row groups into two categories $ORG1$ and $ORG2$:

$$A_1 = \begin{pmatrix} 0 & a_{21} & 0 & a_{23} & 0 \\ a_{30} & 0 & a_{32} & 0 & a_{34} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3.4: Sparse matrix A_1 for *CCBSWCRS*.

$$A_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a_{02} & a_{03} & 0 \\ a_{10} & 0 & 0 & 0 & a_{14} \\ 0 & 0 & a_{42} & a_{43} & 0 \end{pmatrix}$$

Figure 3.5: Sparse matrix A_2 for *CCBSWCRS*.

1. $ORG1 = \{org_j : \theta_j/n > threshold\}$.

2. $ORG2 = \{org_k : \theta_k/n \leq threshold\}$.

We fix 0.90 as the value of *threshold*. We permute the rows of A in such a way that the rows in $ORG1$ are placed first. And the rows in a orthogonal row group of $ORG1$ are placed consecutively. A_1 stores all the rows in $ORG1$ and A_2 stores the rest rows.

For example, the given sparse matrix A in figure3.1 has three orthogonal row groups found by the heuristic in Algorithm 7: $ORG = \{org_0, org_1, org_2\}$, where $org_0 = \{0,1\}$, $org_1 = \{2,3\}$ and $org_2 = \{4\}$. And $ORG1 = \{org_1\}$ if $threshhold = 0.9$. So matrix A_1 has row 2 and 3. Figure 3.4 and 3.5 show A_1 and A_2 respectively.

Matrix A_1 is stored in *CCBS* scheme. Let $\beta^{1'}$ and $ncol_1$ be the number of elementary blocks and the number of orthogonal row groups in A_1. Matrix A_2 is stored in *CRS* scheme. Let nnz_2 be the total number of nonzeros in A_2. So the memory requirement of this scheme is $8(n \cdot ncol_1 + nnz_2) + 4(2\beta^{1'} + m + nnz_2 + 3)$ bytes.

Sparse matrix-vector multiplication A_1x and A_2x are done separately using Algorithm 6 and 2 respectively.

3.1.7 Modified Compressed Column Block Storage (MCCBS)

This storage scheme is same as CCBS scheme except the rows are not partitioned into a number of orthogonal row groups. Three arrays (*value*, *brind*, and $brptr$) and one integer ($ncol'$) are used in this storage scheme to store sparse marix A.

Let $CR_0, CR_1, \ldots, CR_{ncol'-1}$ be arrays of length n each. We will call these arrays as *compressed rows*. Each of these compressed rows is filled by nonzeros of the sparse matrix such that,

1. a nonzero a_{ij} is in one compressed row.
2. if a nonzero a_{ij} is in compressed row CR_k then it must be in $CR_k[j]$.
3. if $\{a_{ij}, a_{ij+1}, \ldots a_{ij+l-1},\}$ is a nonzero block of length l in A then all nonzeros of this nonzero block are in the same compressed row.
4. all nonzeros in A are placed in the compressed rows.

All entries of compressed rows may not be occupied by nonzeros. So we fill the empty places by zeros. These compressed rows are merged one after another and stored in *value*. So the size of the *value* is $(n \cdot ncol')$.

The row indices of all elementary blocks are stored in *brind* array. The indices of the first elements of all elementary blocks in *value* array are stored in $brptr$ array. Sometimes, total number of elementary blocks in *value* can be less than total number of nonzero blocks in A. Because two or more nonzero blocks of a row separated by some zeros in A can be merged as an elementary block in *value*.

Let R_j be the set of rows such that all rows in R_j contribute at least one nonzero block in compressed row CR_i. We insert a new nonzero block in CR_i according to the following rules:

Table 3.7: MCCBS data structure of A.

value	a_{10}	a_{21}	a_{02}	a_{03}	a_{14}	a_{30}	0	a_{32}	a_{23}	a_{34}	0	0	a_{42}	a_{43}	0
brind	1	2	0	1	3	2	3	4							
brptr	0	1	2	4	5	8	9	10	15						

1. choose the largest nonzero block b_s (which is not in any compressed row) from any row of R_j.

2. if no such b_s found, choose the largest nonzero block b_q (which is not in any compressed row) from any other row of A.

The first rule of choosing nonzero block for a compressed row is preferred to improve the temporal locality of y during sparse matrix-vector multiplication under this scheme. The expected advantage of this scheme over *CCBS* is to have fewer zeros in *value* array. But in practice, $ncol'$ is often so big that it is impractical to store A in *value* array under this storage scheme in computer memory.

The multiplication algorithm of this scheme is same as *CCBS*. The memory requirement of this scheme is $8n \cdot ncol' + 4(2\beta' + 2)$ bytes (where β' is the total number of elementary blocks in *value*). The data structure in *MCCBS* scheme is illustrated in Table 3.7.

3.2 Summary of Storage Requirements for Sparse Storage Schemes

CCBSWCRS and MCCBS schemes are new data structures for sparse matrix. The storage requirement of each storage scheme described in this chapter is given in Table 3.8. The desciption of symbols used in this table is given below.

- β_r is the number of nonzero blocks of size r.

- *maxNonzeroBlock* is the size of the largest nonzero block in A.

- β is the total number of nonzero blocks in A.

Table 3.8: Storage Requirements for Sparse Storage Schemes.

Storage Scheme Name	Storage Requirement (bytes)
CRS	$8nnz + 4(nnz + m + 1)$
CCS	$8nnz + 4(nnz + n + 1)$
FSBl	$8nnz + 4(2m + 2 + \sum_{r=1}^{maxNonzeroBlock}((\lfloor r/l \rfloor + r \bmod l)\beta_r))$
BCRS	$8nnz + 4(2\beta + m + 2)$
CCBS	$8n \cdot ncol + 4(2\beta' + 2)$
CCBSWCRS	$8(n \cdot ncol_1 + nnz_2) + 4(2\beta^{1'} + m + nnz_2 + 3)$
MCCBS	$8n \cdot ncol' + 4(2\beta' + 2)$

- β' is the total number of elementary blocks in A.
- $ncol$ is the number of orthogonal row groups.
- $ncol_1$ is the number of orthogonal row groups in A_1 of CCBSWCRS.
- nnz_2 is the number of nonzeros in A_2 of CCBSWCRS.
- $\beta^{1'}$ is the total number of elementary blocks in A_1 of CCBSWCRS.
- $ncol'$ is the number of compressed rows of A.

The input matrices from [5] are given in coordinate format. We convert all these matrices into different storage schemes. The storage requirements for CRS and CCS schemes are same for square matrices. If a sparse matrix has fewer nonzero blocks then BCRS scheme requires fewer number of bytes than that of CRS. The memory requirement for FSBl schemes depends on both l and β_r.

Chapter 4

Column Ordering Algorithms

In this chapter we describe four column ordering algorithms: *column intersection ordering*, *local improvement ordering*, *similarity ordering*, and *binary reflected gray code ordering*. *Column intersection ordering*, *local improvement ordering* and *similarity ordering* algorithms are taken from literature. We propose *binary reflected gray code ordering*. We use the sparse matrix A shown in Figure 4.1 for describing column ordering algorithms.

Columns j and l of matrix A are said to *intersect* if there is a row i such that $a_{ij} \neq 0$ and $a_{il} \neq 0$. The *weight of intersection* of any two columns j and l, denoted by w_{jl}, is the number of rows in which they intersect. We define the *column ordering problem* as follows.

Given an $m \times n$ sparse matrix A, find a permutation of columns that minimizes β*, where* β *is the total number of nonzero blocks in A.*

The following result (Proposition 1) is observed in [29].

$$A = \begin{pmatrix} 0 & 0 & 0 & a_{03} & 0 \\ a_{10} & 0 & 0 & 0 & a_{14} \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ a_{30} & 0 & 0 & 0 & a_{34} \\ 0 & 0 & 0 & a_{43} & 0 \\ a_{50} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{63} & a_{64} \end{pmatrix}$$

Figure 4.1: Sparse matrix A.

Proposition 1. *For any column permutation* π *of an* $m \times n$ *matrix A,* $\sum_{j=0}^{n-2} w_{j,j+1} + \beta = nnz$*, where* β *is the total number of nonzero blocks in the matrix A resulting from applying* π *on the columns of A.*

Proof. A nonzero block of size l contributes $l-1$ to the total weight of intersection ($\sum_{j=0}^{n-2} w_{j,j+1}$). Now if we sum the contributions of all nonzero blocks to the total weight of intersection and β it will be equal to *nnz*. □

As $\sum_{j=0}^{n-2} w_{j,j+1} + \beta = nnz$ and *nnz* is a constant for a sparse matrix, we can rewrite the column ordering problem as follows.

Given an $m \times n$ *sparse matrix A, find a permutation of columns that maximizes the sum* $\sum_{j=0}^{n-2} w_{j,j+1}$.

In [29] it is shown that the above problem is NP-Hard. By Proposition 1 we can say that nonzero block minimization problem is also in NP-Hard. *Column intersection ordering*, *local improvement ordering*, and *similarity ordering* are three heuristic algorithms to solve column ordering problem. *Column intersection ordering* and *local improvement ordering* algorithms are given in [29]. *Similarity ordering* is implemented based on a *distance measure* function described in [16]. We develop *binary reflected gray code ordering* algorithm for solving the column ordering problem.

The common objective of *column intersection ordering*, *local improvement ordering*, and *similarity ordering* algorithms is to permute the columns of a sparse matrix A in such a way that the number of nonzero blocks is minimized. Thus the spatial locality of x is expected to be improved if we access A row wise in computing Ax. To improve the temporal locality of x in this case we need to permute the rows of A in such a way that the nonzeros of each column are consecutive [28]. We show that *binary reflected gray code ordering* improves both temporal and spatial locality of x.

$$A = \begin{pmatrix} 0 & 0 & a_{03} & 0 & 0 \\ a_{10} & a_{14} & 0 & 0 & 0 \\ 0 & 0 & a_{23} & a_{21} & a_{22} \\ a_{30} & a_{34} & 0 & 0 & 0 \\ 0 & 0 & a_{43} & 0 & 0 \\ a_{50} & 0 & 0 & 0 & 0 \\ 0 & a_{64} & a_{63} & 0 & 0 \end{pmatrix}$$

Figure 4.2: Given sparse matrix A after column intersection ordering is performed.

4.1 Column Intersection Ordering

Algorithm 8 describes the *column intersection ordering* algorithm.

Algorithm 8 Column intersection ordering algorithm [29].

1: integer array P of size n. Initially all elements of P are zero;
2: $i \leftarrow 0$;
3: set $C = \{c_0, c_1, \ldots, c_{n-1}\}$, represents column indices of A
4: ▷ w_{ij} is the number of intersection between column i and j
5: $P[i] \leftarrow c_0$;
6: **while** $i < n-1$ **do**
7: find a column $c_j \notin P$ such that $w_{p[i]c_j}$ is largest;
8: $i \leftarrow i+1$;
9: $P[i] \leftarrow c_j$;
10: **end while**

Upon termination the array P holds the order of the columns found by Algorithm 8. In Figure 4.2, the sparse matrix A is shown after column intersection ordering is computed.

Let π be the column permutation found by Algorithm 8 for a given sparse matrix A. In the worst case, we need to access $n-1$ times the nonzeros of each column $\pi[i]$ in this algorithm. So, the running time of this algorithm is $O(nnz \cdot (n-1))$.

4.2 Local Improvement Ordering

We call the given ordering of A as *natural ordering*. *Local improvement ordering* improves the sum of weight of intersections in *natural ordering* iteratively. In this algorithm we first

partition the columns of A into a number of lists $C_0, \ldots, C_p$ such that,

- Each column of A belongs to one and only one list.
- Let c_i^j and c_i^{j+1} be two consecutive columns in C_i. Then these two columns are also consecutive in *natural ordering*.
- Let c_i^{ζ} be the last column of C_i and c_{i+1}^0 be the first column of C_{i+1}. Then these two columns are consecutive in *natural ordering*. So c_0^0 is the 0^{th} column in *natural ordering* and $(n-1)^{th}$ column in *natural ordering* is the last column of C_p.
- c_i^j and c_i^{j+1} are two consecutive columns in C_i of $w_{c_i^j c_i^{j+1}} \geq \rho_{c_i^j} \cdot \delta$, where $\rho_{c_i^j}$ is the number of nonzeros in column c_i^j and δ is a real number between 0 and 1. We fix 0.90 as the value of δ.
- c_i^{ζ} and c_{i+1}^0 be the last and first columns of C_i and C_{i+1} respectively then $w_{c_i^{\zeta} c_{i+1}^0} < \rho_{c_i^{\zeta}} \cdot \delta$.

For example, the columns of the given sparse matrix A in Figure 4.1 can be partitioned into four lists: $C_0 = [0], C_1 = [1,2], C_2 = [3]$, and $C_3 = [4]$ according to the method described above.

We can compute $C_0, \ldots, C_p$ in $O(2nnz)$ time by calculating the weight of intersection between consecutive columns of A. The inputs to the *local improvement ordering* algorithm are the lists $C_0, \ldots, C_p$. On termination the algorithm gives a column order π of A such that,

- If c_i^j and c_i^{j+1} are two consecutive columns in C_i then these two columns are also consecutive in π.
- Let $\pi[i]$ be the last column of C_j. Then $\pi[i+1]$ is the first column of any C_k such that $w_{\pi[i]\pi[i+1]}$ is maximum.

Algorithm 9 Local improvement ordering algorithm [29].

P is an integer array of size n. Initially all elements of P are zero;
C is a two dimensional vector
▷ $C[i]$ stores the column indices of list C_i
▷ $length(C[i])$ is the number of column indices in $C[i]$
$flag$ is an array of size $p+1$. Initially all elements of $flag$ are zero;

for $i \leftarrow 1$ to $length(C[0])$ **do**
    $P[i-1] \leftarrow C[0][i-1]$;
end for
$flag[0] \leftarrow 1$;
$j \leftarrow length(C[0])$;
$b \leftarrow 1$;
while $b < p$ **do**
    find a $C[l]$ such that $flag[l] = 0$ and $w_{P[j-1]C[l][0]}$ is largest;
    for $i \leftarrow 1$ to $length(C[l])$ **do**
        $P[j] \leftarrow C[l][i-1]$;
        $j \leftarrow j+1$;
    end for
    $flag[l] \leftarrow 1$;
    $b \leftarrow b+1$;
end while

$$A = \begin{pmatrix} 0 & 0 & a_{03} & 0 & 0 \\ a_{10} & a_{14} & 0 & 0 & 0 \\ 0 & 0 & a_{23} & a_{21} & a_{22} \\ a_{30} & a_{34} & 0 & 0 & 0 \\ 0 & 0 & a_{43} & 0 & 0 \\ a_{50} & 0 & 0 & 0 & 0 \\ 0 & a_{64} & a_{63} & 0 & 0 \end{pmatrix}$$

Figure 4.3: Given sparse matrix A after local improvement ordering is performed.

Algorithm 9 describes *local improvement ordering*. On termination the array P holds the order of the columns found by the algorithm. Figure 4.3 shows the given sparse matrix A after *local improvement ordering* is performed.

We can order the lists $C_0,\ldots,C_p$ as $C'_0,\ldots,C'_p$ such that the column indices in C'_i come before the column indices of C'_{i+1} in π. In Algorithm 9, we need to access i and $p-1-i$ times the nonzeros of $c_i^{0'}$ (the first column of C'_i) and the nonzeros of $c_i^{\zeta'}$ (the last column of C'_i) respectively. So the running time of this algorithm is $O(2nnz+\sum_{j=1}^{p-1} j\cdot\rho_{c_j^{0'}}+\sum_{j=0}^{p-1}(p-1-j)\cdot\rho_{c_j^{\zeta'}})$. The running time of this algorithm depends on *natural ordering*. As column ordering algorithm is used as a preprocessing step, we are interested in computationally efficient heuristics.

4.3 Similarity Ordering

The algorithm for *similarity ordering* is similar to the algorithm for *column intersection ordering* except that we take into account both zeros and nonzeros in determining the weight of intersection between two columns. In this column ordering algorithm, the weight of intersection between two columns i and j is the number of rows in which both of them have either zero or nonzero. The running time is also the same as that of *column intersection ordering*. *Similarity ordering* algorithm tries to keep similar type of columns consecutively. In [16], the total weight of intersection (considering both zeros and nonzeros) of a sparse matrix is used to measure the data locality of A.

To improve the running time of this algorithm in practice we sort the columns of A in ascending order of ρ_i (where ρ_i is the number of nonzeros in column i) before applying *similarity ordering* algorithm. As $\rho_i \ll m$ for all i, we can sort the columns of A in ascending order of ρ_i in $O(n)$ time by counting sort algorithm. The reason for sorting the columns of A is as follows.

$$A = \begin{pmatrix} 0 & 0 & a_{03} & 0 & 0 \\ 0 & 0 & 0 & a_{14} & a_{10} \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & 0 & 0 & a_{34} & a_{30} \\ 0 & 0 & a_{43} & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{50} \\ 0 & 0 & a_{63} & a_{64} & 0 \end{pmatrix}$$

Figure 4.4: After similarity ordering of sparse matrix A.

Let π be the column permutation found by this ordering algorithm for a given sparse matrix A. The upper and lower bound of the weight of intersection between two columns i and j according to the method for determining weight of intersection used in this ordering algorithm are $(m-|(\rho_i-\rho_j)|)$ and $(m-\rho_i-\rho_j)$ respectively. So, in searching the candidate column $\pi[i+1]$, it is sufficient to search columns sequentially until we get a column b such that $\rho_b = 3\rho_{\pi[i]}$ provided that the columns of input matrix A is sorted as stated and there is at least one column c such that $\rho_c = \rho_{\pi[i]}$ and $c \notin \{\pi[0],\ldots,\pi[i]\}$.

In Figure 4.4, the given sparse matrix A is shown after similarity ordering is performed.

4.4 Binary Reflected Gray Code Ordering

Let π be the column permutation found by *column intersection ordering* or *local improvement ordering* or *similarity ordering* algorithm. Then column $\pi[i+1]$ in the above ordering algorithms is found by looking at the nonzeros of column $\pi[i]$. But the data locality of A should be evaluated over more than pairs of columns [16]. We develop a new column ordering algorithm based on *binary reflected gray code* for sparse matrices. We will call this column ordering algorithm as *binary reflected gray code* or BRGC ordering. To the best of our knowledge, this algorithm is new for the column ordering problem of sparse matrices. BRGC ordering algorithm tries to keep columns having same pattern consecutive in order to improve both spatial and temporal locality of x during computing Ax.

$$B = \begin{pmatrix} b_{00} & b_{01} & 0 & 0 & b_{04} & b_{05} \\ b_{10} & 0 & 0 & b_{13} & b_{14} & 0 \\ 0 & b_{21} & b_{22} & 0 & b_{24} & 0 \end{pmatrix}$$

Figure 4.5: Sparse matrix B.

A *gray code* is an ordering of the 2^p binary strings of length p such that any two consecutive binary strings in the ordering differ only in exactly one bit. A *binary reflected gray code* [21] is a gray code denoted by G^p as follows: $G^p = [0G_0^{p-1}, \ldots, 0G_{2^{p-1}-1}^{p-1}, 1G_{2^{p-1}-1}^{p-1}, \ldots, 1G_0^{p-1}]$, where G_i^p is the i_{th} binary string of G^p. For example $G^3 = [000, 001, 011, 010, 110, 111, 101, 100]$. *Rank* of a binary string G_q^p in a binary reflected gray code G^p is the position of G_q^p in G^p. For example the rank of 011 and 010 in G^3 is 2 and 3 respectively.

An ordering of q binary strings of length p $\{G_{i_1}^p, G_{i_2}^p, \ldots, G_{i_q}^p\}$ in descending order according to binary reflected gray code ranking means that $G_{i_j}^p$ comes before $G_{i_k}^p$ if the rank of $G_{i_j}^p$ is greater than that of $G_{i_k}^p$. For example the order of $\{001, 010, 111, 100\}$ in descending order according to binary reflected gray code ranking is $(100, 111, 010, 001)$.

In BRGC ordering of columns each column of a sparse matrix A is treated as a binary string of length m considering each nonzero in A as 1. The most significant bits of such columns are the bits from row 0 of A. For example, corresponding to the matrix of Figure 4.5, there are 6 binary strings of length 3: $\{110, 101, 001, 010, 111, 100\}$. The corresponding $0-1$ matrix is shown in Figure 4.6. Now the sorted binary strings in descending order according to binary reflected gray code ranking is: $\{100, 101, 111, 110, 010, 001\}$. BRGC ordering of A sorts the columns in descending order according to binary reflected gray code ranking. Instead of reducing the number of blocks, it places the similar type of columns contiguously. Thus, we can expect to have good data locality in the column reordered matrix. A procedure for computing gray code order for the columns is depicted in Algorithm 10. On output the array P holds the binary reflected gray code ordering of the columns.

$$B = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 4.6: Sparse matrix B where all nonzeros are viewed as 1s.

$$B = \begin{pmatrix} b_{05} & b_{01} & b_{04} & b_{00} & 0 & 0 \\ 0 & 0 & b_{14} & b_{10} & b_{13} & 0 \\ 0 & b_{21} & b_{24} & 0 & 0 & b_{22} \end{pmatrix}$$

Figure 4.7: After BRGC ordering of matrix B.

In Figure 4.7 and 4.8 matrices B and A are shown after BRGC ordering.

4.4.1 Correctness of *BRGC* Ordering Algorithm

Given an $m \times n$ matrix A, we consider each nonzero of A as 1. So each column of A can be treated as a binary string of length m. The most significant bit of such a binary string is the bit from row 0.

Proposition 2. *Algorithm 10 order the columns of A in descending order according to their binary reflected gray code rankings.*

Proof. The sparsity pattern of a column c^j of A can be represented by a binary vector $b^j = [b^j_{m-1} \dots b^j_0]$, where $b^j_i = a_{(m-1-i)j}$. Let b^j represents the ordered set $T^j \subseteq \{1, \dots, m\}$ such that the elements in T^j are in ascending order and if $b^j_i = 1$ then $m - i \in T^j$ and

$$A = \begin{pmatrix} a_{03} & 0 & 0 & 0 & 0 \\ 0 & a_{10} & a_{14} & 0 & 0 \\ a_{23} & 0 & 0 & a_{21} & a_{22} \\ 0 & a_{30} & a_{34} & 0 & 0 \\ a_{43} & 0 & 0 & 0 & 0 \\ 0 & a_{50} & 0 & 0 & 0 \\ a_{63} & 0 & a_{64} & 0 & 0 \end{pmatrix}$$

Figure 4.8: After BRGC ordering of matrix A.

Algorithm 10 BRGC ordering algorithm.

```
1: global integer array P;
2: global integer i ← 0;
3: C = {c_0, c_1, ..., c_{n-1}};                    ▷ c_i denotes column i of A
4: rowIndex = 0;
5: sign = +1;
6: procedure BRGC(C, rowIndex, sign )
7:     if C has only one element OR rowIndex = m then
8:         for j ← 0 to |C| − 1 do
9:             P[i] ← C[j];
10:            i ← i + 1;
11:        end for
12:        return
13:    end if
14:    partition C into two disjoint sets C_1 and C_2 such that C_1 contains all column indices that
       have nonzeros in rowIndex and C_2 contains the other indices;
15:    if sign = +1 then
16:        BRGC(C_1, rowIndex + 1, −1);
17:        BRGC(C_2, rowIndex + 1, +1);
18:    else
19:        BRGC(C_2, rowIndex + 1, −1);
20:        BRGC(C_1, rowIndex + 1, +1);
21:    end if
22: end procedure
```

$m-i \notin T^j$ otherwise. Algorithm 11 (Algorithm 2.4 of page 41 in [21]) computes the rank of a binary string in binary reflected gray code ordering.

Let b^l and b^k be two binary vectors corresponding to columns c^l and c^k of A respectively. The corresponding ordered sets of b^l and b^k are T^l and T^k respectively. Let $T_i^l = T_i^k$ for $i = 0, \ldots, s-1$ and $T_s^l < T_s^k$. According to the Algorithm 11, the rank of b^l is greater than that of b^k if s is even. Otherwise the rank of b^k is greater than b^l.

Consider $a_{il} = a_{ik}$ for $i = 0, \ldots, h-1$ and $a_{hl} \neq a_{hk}$. As $T_s^l < T_s^k$, a_{hl} is a nonzero. Both c^l and c^k have even (odd) number of nonzeros in their $i = 0, \ldots, h-1$ row positions if we consider s is even (odd). Algorithm 10 does not change the value of *sign* for C_2 set. The *sign* change occurs in only for C_1. Both c^l and c^k are in the same set of columns when $rowIndex = h-1$ with $sign = +1$ if s is even and $sign = -1$ otherwise. As a_{hl} is a nonzero and the algorithm partitions the columns in depth first search manner, c^l (c^k) enters into P array before c^k (c^l) if s is even (odd).

Thus we prove that the Algorithm 10 orders the columns of a matrix in descending order according to their binary reflected gray code rankings. □

Algorithm 11 The ranking algorithm for binary reflected gray code [21].

```
 1: procedure RANKINGBINREFGRAYCODE(m, T^j )
 2:     r = 0;
 3:     d = 0;
 4:     for i ← m − 1 downto 0 do
 5:         if m − i ∈ T^j then
 6:             d = 1 − d;
 7:         end if
 8:         if d = 1 then
 9:             r = r + 2^i;
10:         end if
11:     end for
12:     return(r);
13: end procedure
```

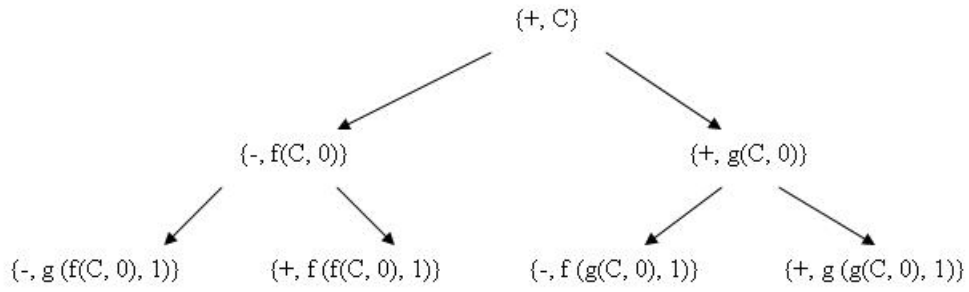

Figure 4.9: Recursive tree produced by BRGC ordering algorithm.

4.4.2 Complexity Analysis of *BRGC* Ordering Algorithm

At any leaf of the recursive tree produced by Algorithm 10, either $rowIndex = m$ or $|C| \leq 1$. In each recursive call, *rowIndex* is increased by 1. Its running time is given by the following recurrence relation:

$T(n) = T(n_1) + T(n_2) + O(n)$, where $n_1 + n_2 \leq n$.

So, the complexity of this algorithm is the same as quicksort. Its running time is $O(n^2)$ and $O(nlogn)$ in worst case and in average case respectively [4].

4.4.3 On the Implementation of *BRGC* Ordering Algorithm

Consider the recursion tree in Figure 4.9 generated by Algorithm 10. Here C is the set of columns. $f(C,i) \subset C$ such that all the columns $c \in f(C,i)$ have a nonzero in row i. And $g(C,i) \subset C$ such that all the columns $c \in g(C,i)$ have a zero in row i.

A column has rank of (i, j) if its i^{th} nonzero from top (row 0) is in row j. For example, sparse matrix B in Figure 4.6, the first column has ranks of $(1,0)$ and $(2,1)$. The nodes in the recursion tree are visited in depth first order and columns having rank $(1,0)$ are partitioned first at the root of the recursive tree. The same partitioning is done in the right subtree, the right subtree of the right subtree and so on. So, it is clear that all columns

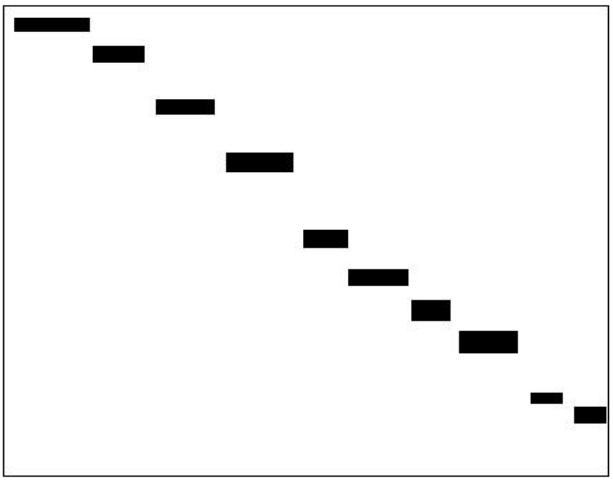

Figure 4.10: View of a sparse matrix after *BRGC* ordering, where the nonzeros participating in all columns having rank of $(1,i)$ are shown in colored rectangles.

having rank of $(1,i)$ are occured before columns having rank $(1,j)$ in P, where $j > i$. Expected view of a sparse matrix after this high level partition is given in Figure 4.10.

Among the set of columns having rank of $(1,i)$, columns having rank of $(2,j)$ are placed after columns having rank of $(2,k)$ in P, where $k > j$. Because, if the input column set C of the algorithm has rank of $(1,i)$ then it calls with $sign = -1$ and $g(C,i+1)$ first. Expected view of columns having rank of $(1,i)$ is given in Figure 4.11.

Column ordering is a preprocessing task of sparse matrix-vector multiplication. So, we should emphasis on its efficient implementation. We can improve the running time of the Algorithm 10 in the following ways:

1. If a recursive node has column set C where $|C|$ is small then we can compute the order of the columns in C explicitly according to the basic rule without any further recursive calls.

2. Instead of calling the recursive call with C_2 and $rowIndex+1$, it can call with C_2 and $rowIndex+i$, if none of the column $c \in C_2$ have any nonzero in its $rowIndex+1$,

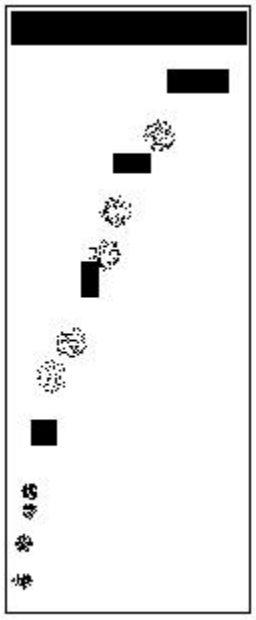

Figure 4.11: Expected view of columns having rank of $(1,i)$.

$rowIndex+2$, .. and $rowIndex+i-1$ positions.

We can also apply the following heuristics to speed up more for very big sparse matrices:

1. If $|C|$ is very small then we can push the columns indices of C in P without making any recursive calls from that node.

2. Let nnz_{CU} denotes the total number of nonzeros in the columns of C at their $\{0,1,\ldots,$ $rowIndex\}$ row indices. And let nnz_{CL} denotes the total number of nonzeros in the columns of C at their $\{rowIndex+1, rowIndex+2, \ldots m-1\}$ row indices. We can push the columns indices of C in P without making any recursive calls from that node if $(nnz_{CU}/(nnz_{CU}+nnz_{CL})) \geq fraction$, where $fraction$ is a real value. We choose the value of $fraction$ as 0.8.

If we apply any of the heuristics described above, we can not expect to get a real binary reflected gray code sorting of the columns. The resultant column order is an approximate binary reflected gray code ordering. We apply these heuristics to reduce the computational time for our test matrices significantly.

4.4.4 Number of Nonzero Blocks and *BRGC* Ordering

In *BRGC* ordering, we have n binary strings of length m. The total number of binary string of length m is 2^m. As $2^m \gg n$, we can not expect to get two columns (binary strings) having difference in one row position (bit) placed consecutively after this ordering. As a result, this ordering does not guarantee optimal number of nonzero blocks for a sparse matrix. Though we found this ordering competitive with *column intersection ordering*, *local improvement ordering* and *similarity ordering* considering the number of nonzero blocks.

In Figure 4.12 and 4.13, the sparsity pattern of sparse matrices bcsstk35 and cavity26 (from [5]) are shown. We also show the sparsity pattern of these matrices after BRGC ordering. We use $spy()$ function of matlab [23] to generate the figures showing sparsity pattern of these matrices after BGRC ordering.

4.4.5 BRGC Ordering of a Random Column Permuted Sparse Matrix

One important characteristic of *BRGC* ordering is that even if the columns of the input matrix is randomly permuted, it will produce the same output. If c^i and c^j are two columns of A and $i > j$ (in the given order) such that $\rho_{c^i} = \rho_{c^j} = w_{c^i c^j}$ then column c^j will come before c^i after BRGC ordering of A. Furthermore, BRGC ordering does not change the sparsity structure of a banded matrix much. If A is a column permuted banded matrix, then A will be a banded matrix again after BRGC ordering. Because in banded matrix there is only one column having rank of $(1,i)$ (where $i > 0$) and column having rank of $(1,i)$ comes before column having rank of $(1,j)$, where $j > i$. BRGC ordering algorithm also order the

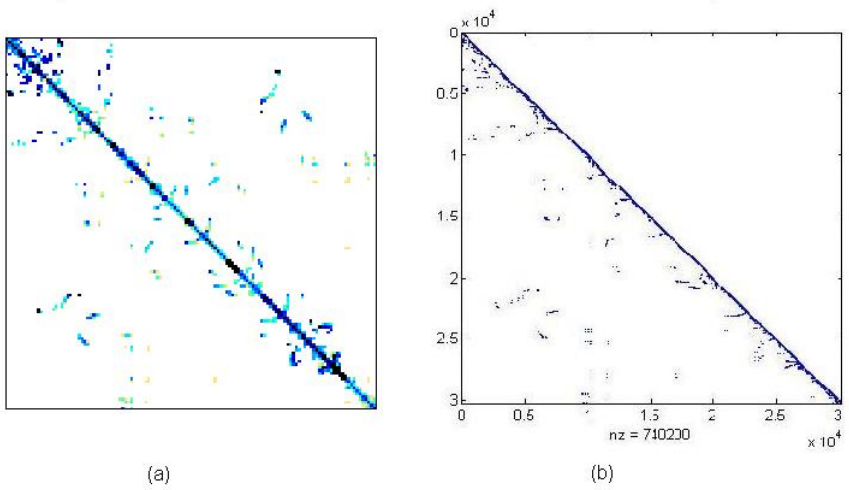

Figure 4.12: (a) spy() view of matrix bcsstk35 [5], (b) spy() view of matrix bcsstk35 after BRGC ordering.

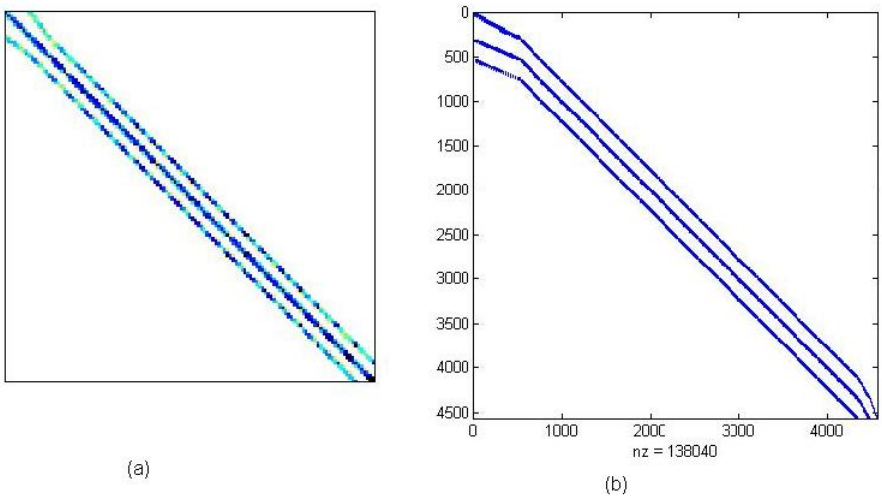

Figure 4.13: (a) spy() view of matrix cavity26 [5], (b) spy() view of matrix cavity26 after BRGC ordering.

columns of a banded matrix in the same manner except columns having rank of $(1,0)$.

4.4.6 Cache Misses in Computing Ax After BRGC Ordering

In this section, we discuss cache misses for the sparse matrix-vector multiplication code (where the sparse matrix is ordered by BRGC ordering) on N-way set associative cache. Note that, a direct mapped cache can be viewed as a 1-way set associative cache. And now a days, the vast majority of processor caches are direct mapped or N-way set associative [15].

For simplicity, we describe cache misses of sparse matrix-vector multiplication ($y = Ax$) codes for *CRS*, *BCRS* and *FSBl* storage schemes. The sparse matrix-vector multiplication codes for these three storage schemes share the following common properties:

1. There is no conflict cache miss in accessing the nonzeros of sparse matrix A.
2. There is no conflict cache miss in accessing y.
3. All three types of cache misses can be observed in accessing x.

We can classify conflict cache misses in accessing x during computing $y = Ax$ under (*CRS*, *BCRS* and *FSBl*) into three categories:

1. Conflict cache misses due to the replacement of a data block of x by any data block not consisting x or y or nonzeros of A.
2. Conflict cache misses due to the replacement of a data block of x by any data block of y or nonzeros of A.
3. Conflict cache misses due to the replacement of a data block of x by any other data block of x. We will call it *self conflict cache miss* of x.

Temam and Jalby [33] analyzed the number of cache misses of sparse matrix-vector multiplication considering only self conflict cache misses of x. They consider an uniform distribution of nonzeros in sparse matrix. We also discuss only the self conflict cache misses of x in computing Ax. We consider sparse matrix A stored in CRS or BCRS or FSB scheme and A is preprocessed by BRGC ordering algorithm.

The nonzeros in a submatrix ($r \times c$) of A participate with ($\{x[p], x[p+1], \ldots, x[p+c-1]\}$) elements of x in computing Ax for some r, c, and p. We will call such a submatrix a *free rectangle* if any two elements in ($\{x[p], x[p+1], \ldots, x[p+c-1]\}$) do not generate any self conflict cache miss of x for each other during sparse matrix-vector multiplication (Ax). The value of r (*width* of a free rectangle) and c (*length* of a free rectangle) are proportional to the size of the cache. Width of a free rectangle also depends on the distribution of nonzeros in that free rectangle. We expect the length of a free rectangle is proportional to the N of N-way set associative cache. In Figure 4.14 such possible free rectangles of a sparse matrix (if the sparse matrix is preprocessed by BRGC ordering algorithm) are shown by black rectangles. Free rectangles are aligned with the first nonzeros from above of all columns having rank of $(1, i)$, where $0 \leq i \leq m$. There is no nonzero in the region above the free rectangles denoted by *Blank*. The nonzeros below the free rectangles are denoted by cross signs. Self conflict cache misses of x occur only when the elements of x associated with the nonzeros in a free rectangle replace any data block of x from cache that is required again in computing Ax.

If A is preprocessed by BRGC ordering then we can find a number of free rectangles where there is no self conflict cache misses during computing $y = Ax$ if A is stored in CRS or BCRS or FSB scheme.

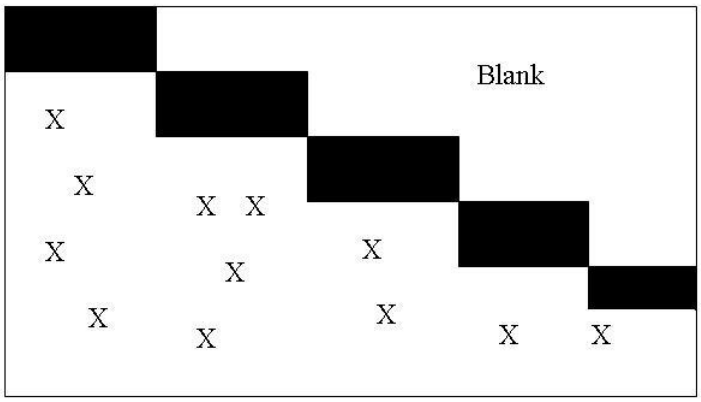

Figure 4.14: Free rectangles in *A* after *BRGC* ordering is performed.

Chapter 5

Experiments

In this chapter we present our experimental results with analyses. We first describe the computing platform specifications on which experiments are carried. We then mention the names of input matrices, chosen storage schemes, and chosen column ordering algorithms followed by a description of evaluation method used. Finally we analyze our experimental results.

5.1 Platform Specifications

We performed all our experiments on three different computing platforms. The technical specifications of these platforms are given below:

1. Compaq: It has AMD Athlon(tm) 64 processor 3500+ (2.2 GHz). It has RAM of size 512 MB. The L2 cache size is 512KB. The cache line size is 64 bytes. The mapping function used for L2 cache is 16-way set associative. The size of both L1 data cache and L1 instruction cache is 64 KB. Both of them have 64 bytes cache line size and use 2-way set associative mapping function [27]. The operating system is linux. The code is written in C++ and compiled by g++ (version number: 4.2.3) with -O option.

2. IBM: It has Intel pentium4 processor (2.8 GHz). It has RAM of size 1GB. The

L2 cache size is 512KB. The cache line size is 64 bytes. The mapping function used for L2 cache is 8-way set associative. The size of L1 data cache is 8 KB. It has 64 bytes cache line size and use 4-way set associative mapping function. The operating system is linux[30]. The code is written in C++ and compiled by g++ (version number: 4.1.2) with -O option.

3. Sun: It has Ultra sparc-IIe processor (550 MHz). It has RAM of size 384 MB. The L2 cache size is 256 KB. The cache line size is 64 bytes. The mapping function used for L2 cache is 4-way set associative and direct mapped. The size of both L1 data cache and L1 instruction cache is 16 KB. Both of them have 64 bytes cache line size and use 2-way set associative mapping function [6]. The operating system is sun solaris 10. The code is written in C++ and compiled by g++ (version number: 3.4.3) with -O option.

5.2 Input Matrices

We choose 26 matrices from [5]. The description of the matrices are given in the Table 5.1. The memory requirement for any of these matrices in any storage scheme of our interest is larger than L2 cache size of all of our computing platforms on which experiments are carried on. In [29] bcsstk35, bcsstk32, bcsstk30, psmigr_1, cavity26, bcsstk29, memplus, and bcsstk28 are used to compare among different column ordering algorithms on different storage schemes of sparse matrices. In [34] bcsstk35, and bcsstk32 are used for the same purpose. In [10], lp_ken_18, lp_cre_d, lp_maros_r7, lp_pds_10, lp_ken_13, lp_fit2d and lp_stocfor3 are used in sparse derivative matrix computation.

5.3 Chosen Storage Scheme for Experiment

We use the following storage schemes for our experiments:

Table 5.1: Input matrices from [5].

MatrixName	m	n	nnz	Application Area
matrix_9	103430	103430	2121550	semiconductor device problem
landmark	71952	2704	1151232	least squares problem
rajat20	86916	86916	605045	circuit simulation problem
bcsstk35	30237	30237	740200	structural problem
bcsstk32	44609	44609	1029655	structural problem
nsct	23003	37563	697738	linear programming problem sequence
bcsstk30	28924	28924	1036208	structural problem
forme21	67748	216350	465294	linear programming problem
psmigr_1	3140	3140	543162	economic problem
lp_ken_18	105127	154699	358171	linear programming problem
blockqp1	60012	60012	340022	optimization problem
Ill_Stokes	20896	20896	191368	computational fluid dynamics problem
cavity26	4562	4562	138187	subsequent computational fluid dynamics problem
bcsstk29	13992	13992	316740	structural problem
memplus	17758	17758	126150	circuit simulation problem
lp_cre_d	8926	73948	246614	linear programming problem
psse0	26722	11028	102432	power network problem
bcsstk28	4410	4410	111717	structural problem
lp_maros_r7	3136	9408	144848	linear programming problem
e18	24617	38602	156466	linear programming problem
baxter	27441	30733	111576	linear programming problem
lp_pds_10	16558	49932	107605	linear programming problem
lp_ken_13	28632	42659	97246	linear programming problem
lp_fit2d	25	10524	129042	linear programming problem
lp_stocfor3	16675	23541	76473	linear programming problem
dictionary28	52652	52652	89038	undirected graph

1. CRS

2. BCRS

3. FSB2.

4. FSB3.

In CCBS and MCCBS, the total number zeros in compressed row groups is very large for most of the input matrices that we decide not to carry on our experiments on these schemes. We do not do any experiments on JDS, TJDS or Bi-JDS, as these schemes are suitable for vector processors. We also do not do any experiment on CDS as it is suitable for banded matrix. As CCS is similar to CRS and CRS is commonly used, we also do not do any experiments on CCS.

In CCBSWCRS scheme a number of rows of a sparse matrix are stored in CCBS scheme. We can not find any set of rows to store in CCBS scheme according to the criteria described in Chapter 3 for most of our chosen input matrices. Furthermore natural row ordering of input matrices are changed in this storage scheme. So in the CRS part of this scheme, the temporal locality of x worsens during sparse matrix-vector multiplication for a number of matrices.

5.4 Notations for Column Ordering Algorithms

The names of the column ordering algorithms along with their notations are given below.

1. Natural Ordering: This is the column ordering given in [5]. We denote this ordering by $O_{natural}$.

2. Column Intersection Ordering: We denote this ordering by $O_{intersection}$.

3. Local Improvement Ordering: We denote this ordering by $O_{improvement}$.

4. Similarity Ordering: We denote this ordering by $O_{similarity}$.

5. Binary Reflected Gray Code Ordering: We denote this ordering by O_{brgc}.

5.5 Evaluation Method

Dolan and Moré [7] proposed performance profiles as a tool for comparing optimization software. We applied their technique for comparing different storage schemes of sparse matrices and column ordering algorithms of sparse matrix-vector multiplication ($SpMxV$). We use CPU time as the performance measure. We do not use average or cumulative total CPU time of $SpMxV$s for performance comparison. Because the CPU time for multiplying a small number of input sparse matrices with vector can dominate the comparison results if this CPU time is considerably big or small. We use $getrusage()$ [32] function for measuring the CPU time for sparse matrix-vector multiplication. CPU time reported here is the time for single sparse matrix-vector multiplication ($y = Ax$) by averaging 1000 sparse matrix-vector multiplications.

First we define three parameters:

1. Computing Platform (PL): $PL = \{sun, compaq, ibm\}$.

2. Storage Scheme (SS) : $SS = \{CRS, BCRS, FSB2, FSB3\}$.

3. Ordering Algorithm (RA): $RA = \{O_{natural}, O_{intersection}, O_{improvement}, O_{similarity}, O_{brgc}\}$.

So, we have 60 (Cartesian product of three sets: PL, SS, RA) different types of $SpMxV$ experiments. We will denote a particular $SpMxV$ by ordered list of three parameters :(pl, ss, ra), where $pl \in PL$, $ss \in SS$ and $ra \in RA$. For example, $SpMxV(compaq, crs, O_{brgc})$ means sparse matrix-vector multiplication on compaq platform, where the input matrix is preprocessed by binary reflected gray code ordering algorithm and the storage is CRS. We

can denote a set of $SpMxV$s by specifying a set in any of these three parameters by ANY. For example, $SpMxV(sun,ANY,ANY)$ denotes all the $SpMxV$s on sun platform.

For each sparse matrix A and $SpMxV(pl,ss,ra)$, we define $t_{A,SpMxV(pl,ss,ra)}$ as the CPU time to multiply A with a dense vector on computing platform pl, where A is preprocessed using ordering algorithm ra and it is stored in ss scheme.

We define $min\{t_{A,SpMxV(pl,ss,ANY)}\}$ as the minimum CPU time to multiply A with a dense vector on computing platform pl where A is stored in ss scheme and A is pre-processed by any column ordering algorithm. For example, $min\{t_{A,SpMxV(pl,ss,ANY)}\}$ be $t_{A,SpMxV(pl,ss,O_{natural})}$. In this case we will call $SpMxV(pl,ss,O_{natural})$ as the best $SpMxV$ among the set of $SpMxV$s defined by $SpMxV(pl,ss,ANY)$ on sparse matrix A.

We define *performance ratio* as $r_{A,SpMxV(pl,ss,ra)} = \frac{t_{A,SpMxV(pl,ss,ra)}}{min\{t_{A,SpMxV(pl,ss,ANY)}\}}$ to compare the performance on input matrix A of $SpMxV(pl,ss,ra)$ with the best performing (in terms of CPU time) $SpMxV$ on computing platform pl and A, where A is preprocessed by any ordering algorithms and it is stored in ss. We can modify the equation for computing performance ratio of a $SpMxV$ according to our objective. For example, if we need to compare with all $SpMxV$s on computing platform pl then we need to change the set of $SpMxV$ in denominator of right hand side of this equation to $SpMxV(pl,ANY,ANY)$.

Finally, the performance of a $SpMxV(pl,ss,ra)$ can be measured by the following cumulative distribution function [7]:

$\rho_{SpMxV(pl,ss,ra)}(\tau) = \frac{1}{|\Gamma|} size\{A \in \Gamma : r_{A,SpMxV(pl,ss,ra)} \leq \tau\}$, where, Γ is the set of input matrices of our interest. And $r_{A,SpMxV(pl,ss,ra)} = \frac{t_{A,SpMxV(pl,ss,ra)}}{min\{t_{A,SpMxV(pl,ss,ANY)}\}}$, if we want to compare $SpMxV(pl,ss,ra)$ with all $SpMxV$s on computing platform pl where input matrices are stored in ss scheme.

If $|\Gamma|$ is suitably large and representative of input matrices that are likely to occur in applications then we can say that $\rho_{SpMxV(pl,ss,ra)}(\tau)$ is the probability that $SpMxV(pl,ss,ra)$ can multiply any sparse matrix with a dense vector within a factor τ of the best performing

SpMxV among a set of *SpMxV* under consideration. So we prefer *SpMxV* with large probability $\rho_{SpMxV(pl,ss,ra)}(\tau)$.

Consider τ' is the minimum value of τ such that all $\rho_{SpMxV(pl,ss,ra)}(\tau')$ values of a set of *SpMxV*s become 1. Then we are interested in the values of $\rho_{SpMxV(pl,ss,ra)}(\tau)$ where $1 \leq \tau \leq \tau'$. We call $SpMxV(pl,ss,ra)$ as the best *SpMxV* among a set of *SpMxV*s if its $\rho_{SpMxV(pl,ss,ra)}(\tau)$ value dominates those of the other *SpMxV*s for most of the values of τ in the range $(1,\tau')$.

In our experiment we do not compare *SpMxV*s among different computing platforms. For each computing platform we first find out the best *SpMxV* for each storage scheme. Then we compare among the best *SpMxV*s of each storage scheme to find out the optimal *SpVxM* for a computing platform.

We show the performance comparison among different column ordering algorithms on a computing platform *pl* where the sparse matrices are in storage scheme *ss*. In the comparison figures each line represents a particular column ordering denoted by the legend. The value of τ is in $X-axis$ and the corresponding $\rho_{SpMxV(pl,ss,ra)}(\tau)$ value is given in $Y-axis$.

5.6 Experiments on Compaq Platform

5.6.1 *SpMxV*s on Compaq Platform with CRS Scheme

A graph showing the performance of $SpMxV(compaq,crs,O_{natural})$, $SpMxV(compaq,crs,O_{intersection})$, $SpMxV(compaq,crs,O_{similarity})$, $SpMxV(compaq,crs,O_{brgc})$ and $SpMxV(compaq,crs,O_{improvement})$ is shown in Figure 5.1. The value of τ' is 3.0. From the figure we can see that $\rho_{SpMxV(compaq,crs,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(compaq,crs,O_{natural})}(\tau)$, $\rho_{SpMxV(compaq,crs,O_{intersection})}(\tau)$, $\rho_{SpMxV(compaq,crs,O_{improvement})}(\tau)$, and $\rho_{SpMxV(compaq,crs,O_{similarity})}(\tau)$ for all values of τ. And $\rho_{SpMxV(compaq,crs,O_{brgc})}(1.2) = 0.96$, which means $SpMxV(compaq,crs,O_{brgc})$

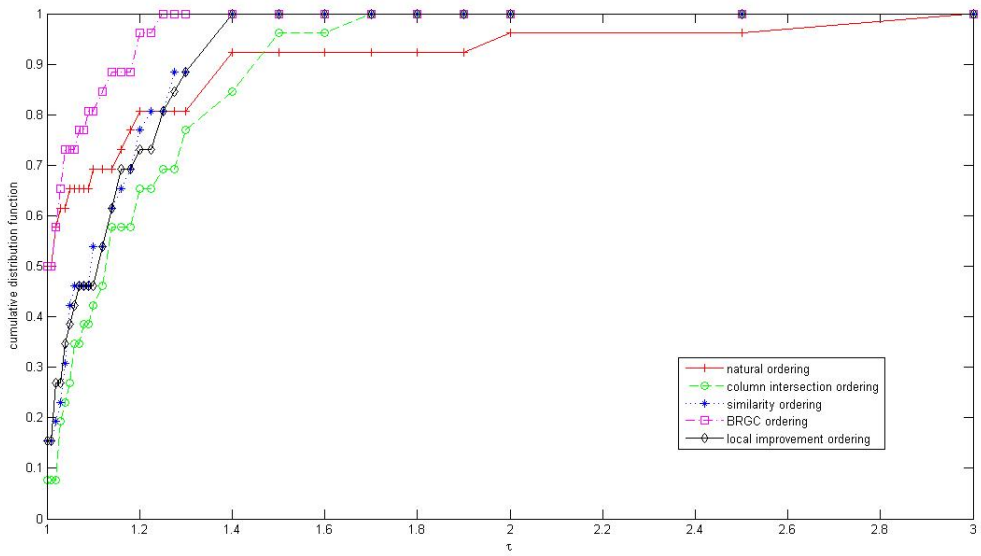

Figure 5.1: Performance of *SpMxV*s on compaq platform with CRS scheme.

can multiply 96% of test matrices with a dense vector within a factor 1.2 of best performing *SpMxV* on compaq platform with CRS scheme. So, we can say that $SpMxV(compaq, crs, O_{brgc})$ performs best among these five *SpMxV*s.

5.6.2 *SpMxV*s on Compaq Platform with BCRS Scheme

A graph showing the performance of $SpMxV(compaq, bcrs, O_{natural})$, $SpMxV(compaq, bcrs, O_{intersection})$, $SpMxV(compaq, bcrs, O_{similarity})$, $SpMxV(compaq, bcrs, O_{brgc})$ and $SpMxV(compaq, bcrs, O_{improvement})$ is shown in Figure 5.2. The value of τ' is 3.0. From the figure we can see that $\rho_{SpMxV(compaq,bcrs,O_{natural})}(\tau)$ dominates $\rho_{SpMxV(compaq,bcrs,O_{brgc})}(\tau)$ when $1 \leq \tau \leq 1.08$. And $\rho_{SpMxV(compaq,bcrs,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(compaq,bcrs,O_{natural})}(\tau)$ when $\tau > 1.08$. Furthermore $\rho_{SpMxV(compaq,bcrs,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(compaq,bcrs,O_{intersection})}(\tau)$, $\rho_{SpMxV(compaq,bcrs,O_{improvement})}(\tau)$, and $\rho_{SpMxV(compaq,bcrs,O_{similarity})}(\tau)$ for all values of τ. And $\rho_{SpMxV(compaq,bcrs,O_{brgc})}(1.16) = 0.96$, which means $SpMxV(compaq, bcrs, O_{brgc})$ can multiply 96% of test matrices with a dense vector within a factor 1.16 of

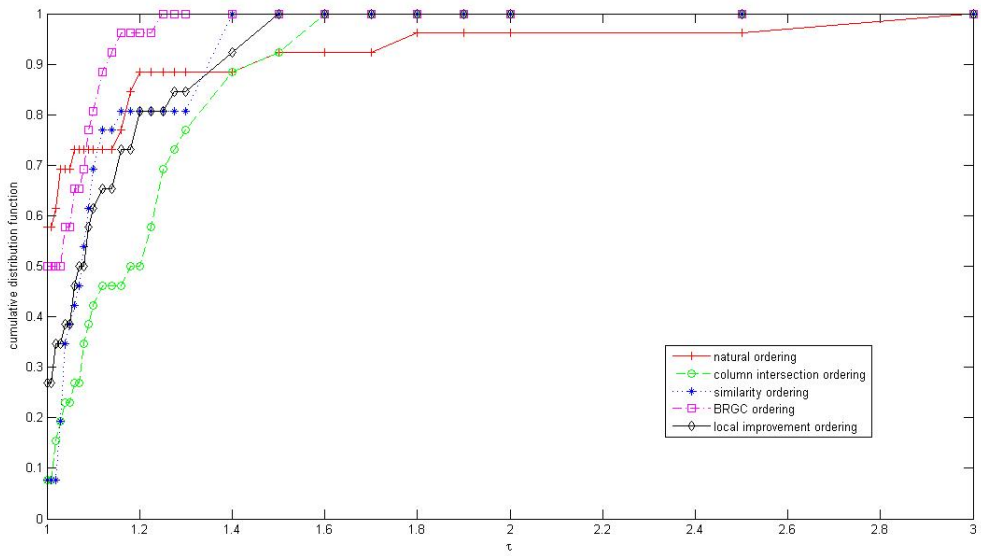

Figure 5.2: Performance of *SpMxV*s on compaq platform with BCRS scheme.

best performing *SpMxV* on compaq platform with BCRS scheme. So, we can say that $SpMxV(compaq, bcrs, O_{brgc})$ performs best among these five *SpMxV*s.

5.6.3 *SpMxV*s on Compaq Platform with FSB2 Scheme

A graph showing the performance of $SpMxV(compaq, fsb2, O_{natural})$, $SpMxV(compaq, fsb2, O_{intersection})$, $SpMxV(compaq, fsb2, O_{similarity})$, $SpMxV(compaq, fsb2, O_{brgc})$ and $SpMxV(compaq, fsb2, O_{improvement})$ is shown in Figure 5.3. The value of τ' is 2.5. From the figure we can see that $\rho_{SpMxV(compaq,fsb2,O_{natural})}(\tau)$ dominates $\rho_{SpMxV(compaq,fsb2,O_{brgc})}(\tau)$ when $1 \leq \tau \leq 1.12$. $\rho_{SpMxV(compaq,fsb2,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(compaq,fsb2,O_{natural})}(\tau)$ when $\tau > 1.12$. $\rho_{SpMxV(compaq,fsb2,O_{similarity})}(\tau)$ dominates $\rho_{SpMxV(compaq,fsb2,O_{brgc})}(\tau)$ when $1.05 \leq \tau \leq 1.06$. $\rho_{SpMxV(compaq,fsb2,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(compaq,fsb2,O_{natural})}(\tau)$ for all other values of τ. $\rho_{SpMxV(compaq,fsb2,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(compaq,fsb2,O_{intersection})}(\tau)$, and $\rho_{SpMxV(compaq,fsb2,O_{improvement})}(\tau)$ for all values of τ. And $\rho_{SpMxV(compaq,fsb2,O_{brgc})}(1.225) = 0.96$, which means $SpMxV(compaq, fsb2, O_{brgc})$ can multiply 96% of test matrices with

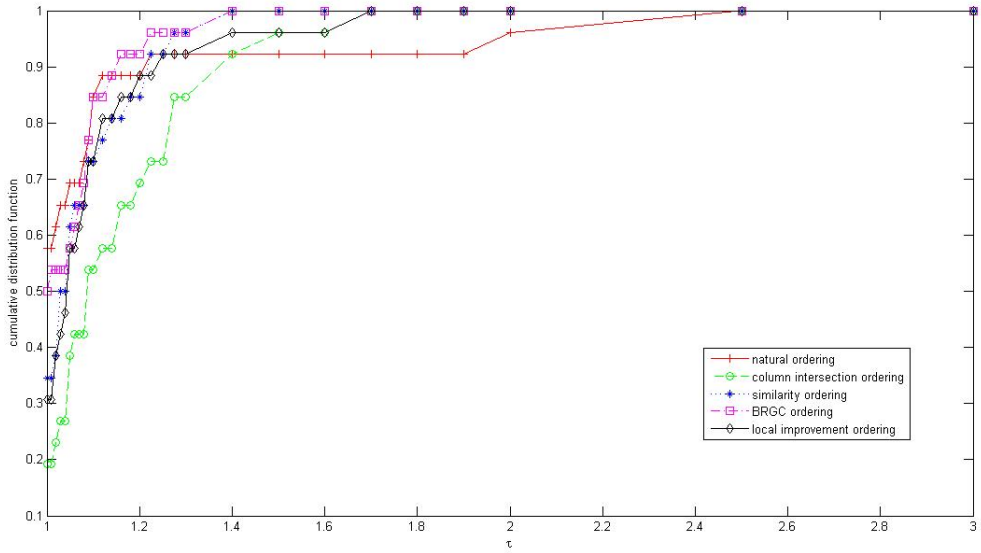

Figure 5.3: Performance of *SpMxV*s on compaq platform with FSB2 scheme.

a dense vector within a factor 1.225 of best performing *SpMxV* on compaq platform with FSB2 scheme. So, we can say that $SpMxV(compaq, fsb2, O_{brgc})$ performs best among these five *SpMxV*s.

5.6.4 *SpMxV*s on Compaq Platform with FSB3 Scheme

A graph showing the performance of $SpMxV(compaq, fsb3, O_{natural})$, $SpMxV(compaq, fsb3, O_{intersection})$, $SpMxV(compaq, fsb3, O_{similarity})$, $SpMxV(compaq, fsb3, O_{brgc})$ and $SpMxV(compaq, fsb3, O_{improvement})$ is shown in Figure 5.4. The value of τ' is 3.0. From the figure we can see that $\rho_{SpMxV(compaq,fsb3,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(compaq,fsb3,O_{natural})}(\tau)$, $\rho_{SpMxV(compaq,fsb3,O_{intersection})}(\tau)$, $\rho_{SpMxV(compaq,fsb3,O_{improvement})}(\tau)$, and $\rho_{SpMxV(compaq,fsb3,O_{similarity})}(\tau)$ for all values of τ. And $\rho_{SpMxV(compaq,fsb3,O_{brgc})}(1.12) = 0.92$, which means $SpMxV(compaq, fsb3, O_{brgc})$ can multiply 92% of test matrices with a dense vector within a factor 1.12 of best performing *SpMxV* on compaq platform with FSB3 scheme. So, we can say that $SpMxV(compaq, fsb3, O_{brgc})$ performs best among these five *SpMxV*s.

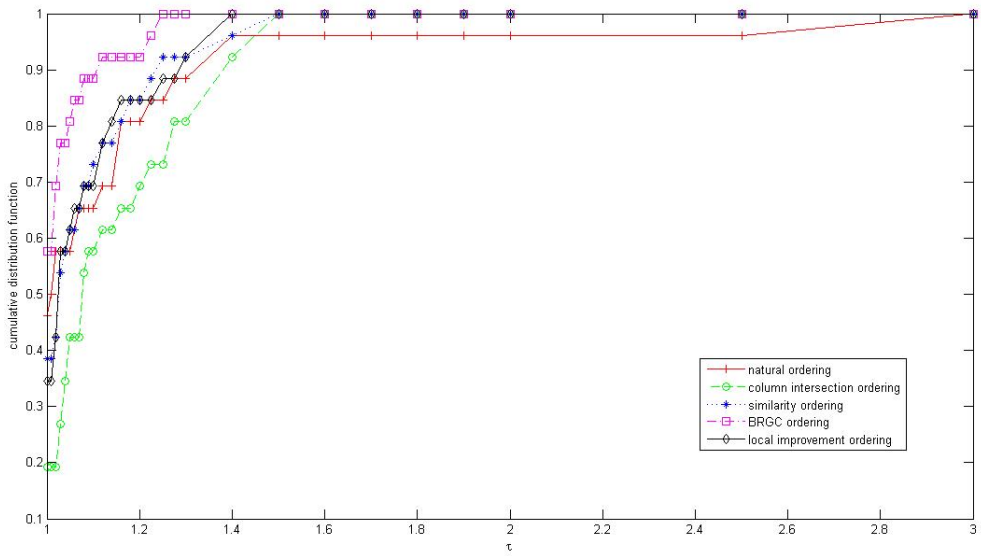

Figure 5.4: Performance of *SpMxV*s on compaq platform with FSB3 scheme.

5.6.5 Optimal *SpMxV* on compaq Platform

BRGC ordering performs the best in each of the storage scheme on compaq platform. Now, we will compare the performance of $SpMxV(compaq, crs, O_{brgc})$, $SpMxV(compaq, bcrs, O_{brgc})$, $SpMxV(compaq, fsb2, O_{brgc})$, and $SpMxV(compaq, fsb3, O_{brgc})$. We present the comparison data among these four *SpMxV*s in Table 5.2.

Table 5.2: Optimal *SpMxV* on compaq platform.

τ	$\rho_{SpMxV(compaq,crs,O_{brgc})}(\tau)$	$\rho_{SpMxV(compaq,bcrs,O_{brgc})}(\tau)$	$\rho_{SpMxV(compaq,fsb2,O_{brgc})}(\tau)$	$\rho_{SpMxV(compaq,fsb3,O_{brgc})}(\tau)$
1.0	0.423076923	0	0.230769231	0.615384615
1.05	0.423076923	0	0.5	0.730769231
1.10	0.576923077	0.076923077	0.807692308	0.884615385
1.20	0.692307692	0.346153846	0.961538462	0.923076923
1.3	0.923076923	0.692307692	0.961538462	1
1.4	0.923076923	0.884615385	1	1
1.5	1	0.961538462	1	1
2	1	1	1	1

$\rho_{SpMxV(compaq,fsb3,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(compaq,crs,O_{brgc})}(\tau)$ and $\rho_{SpMxV(compaq,bcrs,O_{brgc})}(\tau)$. $\rho_{SpMxV(compaq,fsb2,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(compaq,fsb3,O_{brgc})}(\tau)$ only for a small interval of τ. So, we can infer that $SpMxV(compaq, fsb3, O_{brgc})$ is the best $SpMxV$ on compaq platform.

The total CPU time of $SpMxV(compaq, fsb3, O_{brgc})$ improves 11% than that of both $SpMxV(compaq, crs, O_{brgc})$ and $SpMxV(compaq, fsb3, O_{natural})$ separately. Again we observe 22% improvement of total CPU time in $SpMxV(compaq, fsb3, O_{brgc})$ over $SpMxV($ $compaq, crs, O_{natural})$.

5.7 Experiments on IBM Platform

5.7.1 *SpMxV*s on IBM Platform with CRS Scheme

A graph showing the performance of $SpMxV(ibm, crs, O_{natural})$, $SpMxV(ibm, crs,$ $O_{intersection})$, $SpMxV(ibm, crs, O_{similarity})$, $SpMxV(ibm, crs, O_{brgc})$ and $SpMxV(ibm, crs,$ $O_{improvement})$ is shown in Figure 5.5. The value of τ' is 3.0. From the figure we can see that $\rho_{SpMxV(ibm,crs,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(ibm,crs,O_{natural})}(\tau)$, $\rho_{SpMxV(ibm,crs,O_{intersection})}(\tau)$, $\rho_{SpMxV(ibm,crs,O_{improvement})}(\tau)$, and $\rho_{SpMxV(ibm,crs,O_{similarity})}(\tau)$ for all values of τ.

And $\rho_{SpMxV(ibm,crs,O_{brgc})}(1.2) = 0.92$, which means $SpMxV(ibm, crs, O_{brgc})$ can multiply 92% of test matrices with a dense vector within a factor 1.2 of best performing $SpMxV$ on ibm platform with CRS scheme. So, we can say that $SpMxV(ibm, crs, O_{brgc})$ performs best among these five $SpMxV$s.

5.7.2 *SpMxV*s on IBM Platform with BCRS Scheme

A graph showing the performance of $SpMxV(ibm, bcrs, O_{natural})$, $SpMxV(ibm, bcrs,$ $O_{intersection})$, $SpMxV(ibm, bcrs, O_{similarity})$, $SpMxV(ibm, bcrs, O_{brgc})$ and $SpMxV(ibm, bcrs,$ $O_{improvement})$ is shown in Figure 5.6. The value of τ' is 3.0. From the figure we can see that

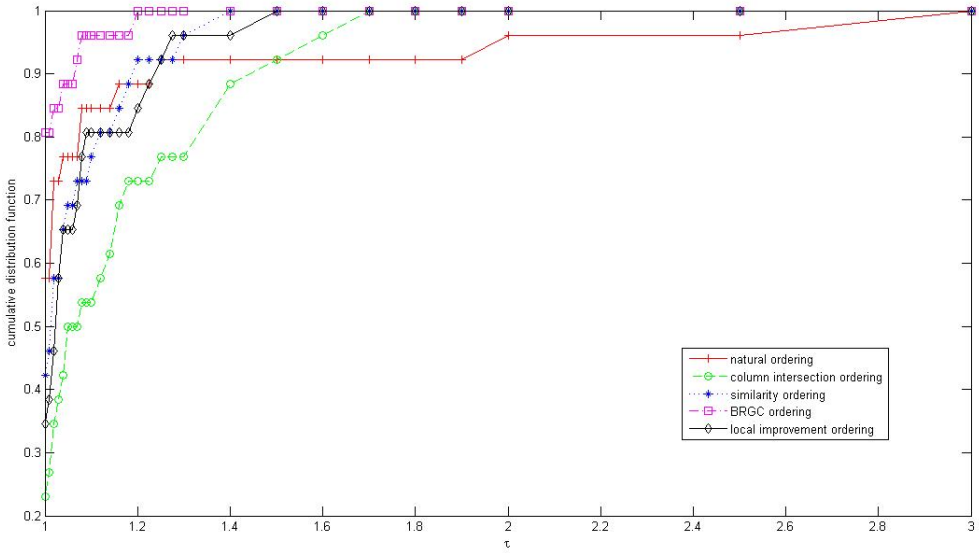

Figure 5.5: Performance of *SpMxV*s on ibm platform with CRS scheme.

$\rho_{SpMxV(ibm,bcrs,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(ibm,bcrs,O_{natural})}(\tau)$, $\rho_{SpMxV(ibm,bcrs,O_{intersection})}(\tau)$, $\rho_{SpMxV(ibm,bcrs,O_{improvement})}(\tau)$, and $\rho_{SpMxV(ibm,bcrs,O_{similarity})}(\tau)$ for all values of τ.

And $\rho_{SpMxV(ibm,bcrs,O_{brgc})}(1.06) = 0.92$, which means $SpMxV(ibm,bcrs,O_{brgc})$ can multiply 92% of test matrices with a dense vector within a factor 1.06 of best performing *SpMxV* on ibm platform with BCRS scheme. So, we can say that $SpMxV(ibm,bcrs,O_{brgc})$ performs best among these five *SpMxV*s.

5.7.3 *SpMxV*s on IBM Platform with FSB2 Scheme

A graph showing the performance of $SpMxV(ibm,fsb2,O_{natural})$, $SpMxV(ibm,fsb2,O_{intersection})$, $SpMxV(ibm,fsb2,O_{similarity})$, $SpMxV(ibm,fsb2,O_{brgc})$ and $SpMxV(ibm,fsb2,O_{improvement})$ is shown in Figure 5.7. The value of τ' is 3.0. Both $\rho_{SpMxV(ibm,fsb2,O_{improvement})}(\tau)$ and $\rho_{SpMxV(ibm,fsb2,O_{similarity})}(\tau)$ dominate $\rho_{SpMxV(ibm,fsb2,O_{brgc})}(\tau)$ for a very small interval of τ $(1.4,1.5)$. $\rho_{SpMxV(ibm,fsb2,O_{brgc})}(\tau)$ dominates both $\rho_{SpMxV(ibm,fsb2,O_{improvement})}(\tau)$ and $\rho_{SpMxV(ibm,fsb2,O_{similarity})}(\tau)$ for all other values of τ. And $\rho_{SpMxV(ibm,fsb2,}$

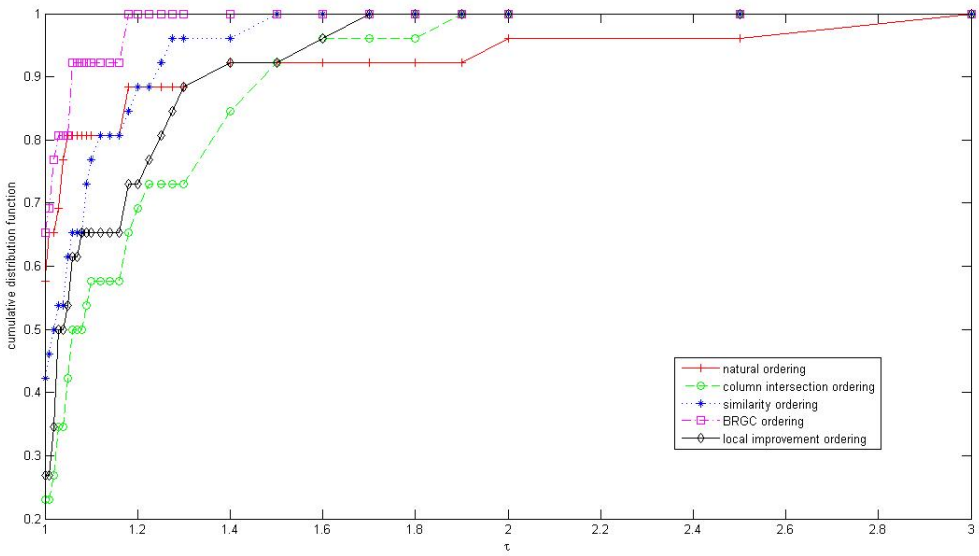

Figure 5.6: Performance of *SpMxV*s on ibm platform with BCRS scheme.

$O_{brgc})(\tau)$ dominates $\rho_{SpMxV(ibm,fsb2,O_{natural})}(\tau)$ and $\rho_{SpMxV(ibm,fsb2,O_{intersection})}(\tau)$ for all values of τ.

Furthermore $\rho_{SpMxV(ibm,fsb2,O_{brgc})}(1.14) = 0.92$, which means $SpMxV(ibm, fsb2, O_{brgc})$ can multiply 92% of test matrices with a dense vector within a factor 1.14 of best performing *SpMxV* on ibm platform with FSB2 scheme. So, we can say that $SpMxV(ibm, fsb2, O_{brgc})$ performs best among these five *SpMxV*s.

5.7.4 *SpMxV*s on IBM Platform with FSB3 Scheme

A graph showing the performance of $SpMxV(ibm, fsb3, O_{natural})$, $SpMxV(ibm, fsb3, O_{intersection})$, $SpMxV(ibm, fsb3, O_{similarity})$, $SpMxV(ibm, fsb3, O_{brgc})$ and $SpMxV(ibm, fsb3, O_{improvement})$ is shown in Figure 5.8. The value of τ' is 3.0. Both $\rho_{SpMxV(ibm,fsb3,O_{improvement})}(\tau)$ and $\rho_{SpMxV(ibm,fsb3,O_{similarity})}(\tau)$ dominate $\rho_{SpMxV(ibm,fsb3,O_{brgc})}(\tau)$ for a very small interval of τ $(1.4, 1.5)$. $\rho_{SpMxV(ibm,fsb3,O_{brgc})}(\tau)$ dominates both $\rho_{SpMxV(ibm,fsb3,O_{improvement})}(\tau)$ and $\rho_{SpMxV(ibm,fsb3,O_{similarity})}(\tau)$ for all other values of τ. And $\rho_{SpMxV(ibm,fsb3,}$

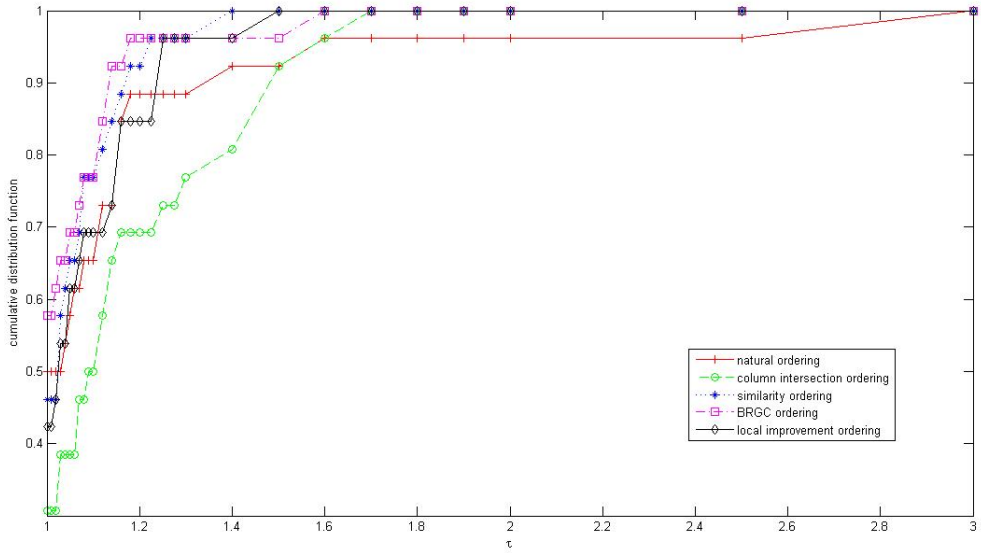

Figure 5.7: Performance of *SpMxV*s on ibm platform with FSB2 scheme.

$_{O_{brgc}})(\tau)$ dominates $\rho_{SpMxV(ibm,fsb3,O_{natural})}(\tau)$ and $\rho_{SpMxV(ibm,fsb3,O_{intersection})}(\tau)$ for all values of τ.

Furthermore $\rho_{SpMxV(ibm,fsb3,O_{brgc})}(1.2) = 0.96$, which means $SpMxV(ibm,fsb3,O_{brgc})$ can multiply 96% of test matrices with a dense vector within a factor 1.2 of best performing *SpMxV* on ibm platform with FSB3 scheme. So, we can say that $SpMxV(ibm,fsb3,O_{brgc})$ performs best among these five *SpMxV*s.

5.7.5 Optimal *SpMxV* on IBM Platform

BRGC ordering performs the best in each of the storage scheme on ibm platform. Now, we will compare the performance of $SpMxV(ibm,crs,O_{brgc})$, $SpMxV(ibm,bcrs,O_{brgc})$, $SpMxV(ibm,fsb2,O_{brgc})$, and $SpMxV(ibm,fsb3,O_{brgc})$. We present the comparison data among these four *SpMxV*s in Table 5.3.

$\rho_{SpMxV(ibm,fsb3,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(ibm,crs,O_{brgc})}(\tau)$, $\rho_{SpMxV(ibm,bcrs,O_{brgc})}(\tau)$, and $\rho_{SpMxV(ibm,fsb2,O_{brgc})}(\tau)$ for all values of τ. So we can infer that $SpMxV(ibm,fsb3,O_{brgc})$

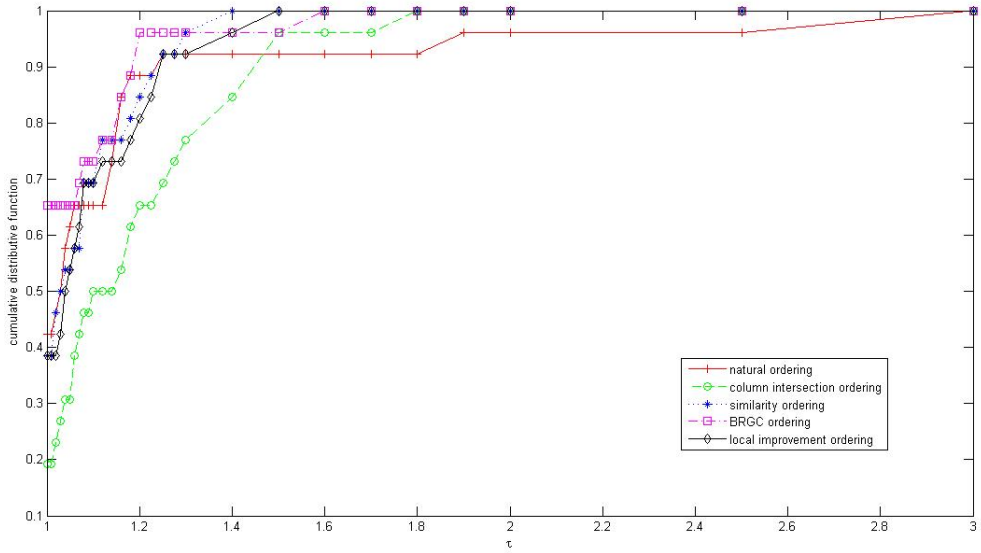

Figure 5.8: Performance of *SpMxV*s on ibm platform with FSB3 scheme.

Table 5.3: Optimal *SpMxV* on ibm platform.

τ	$\rho_{SpMxV(ibm,crs,O_{brgc})}(\tau)$	$\rho_{SpMxV(ibm,bcrs,O_{brgc})}(\tau)$	$\rho_{SpMxV(ibm,fsb2,O_{brgc})}(\tau)$	$\rho_{SpMxV(ibm,fsb3,O_{brgc})}(\tau)$
1.0	0.423076923	0	0.346153846	0.576923077
1.08	0.461538462	0.038461538	0.538461538	0.692307692
1.13	0.5	0.038461538	0.807692308	0.884615385
1.16	0.538461538	0.038461538	0.884615385	0.923076923
1.28	0.615384615	0.076923077	0.923076923	0.923076923
1.34	0.653846154	0.153846154	0.961538462	1
1.35	0.653846154	0.153846154	0.961538462	1
1.36	0.653846154	0.153846154	1	1

is the best $SpMxV$ on ibm platform.

The total CPU time of $SpMxV(ibm, fsb3, O_{brgc})$ improves 26% and 12% than that of $SpMxV(ibm, crs, O_{brgc})$ and $SpMxV(ibm, fsb3, O_{natural})$ respectively. Again we observe 27% improvement of total CPU time in $SpMxV(ibm, fsb3, O_{brgc})$ over $SpMxV(ibm, crs, O_{natural})$.

5.8 Experiments on Sun Platform

5.8.1 *SpMxV*s on Sun Platform with CRS Scheme

A graph showing the performance of $SpMxV(sun, crs, O_{natural})$, $SpMxV(sun, crs, O_{intersection})$, $SpMxV(sun, crs, O_{similarity})$, $SpMxV(sun, crs, O_{brgc})$ and $SpMxV(sun, crs, O_{improvement})$ is shown in Figure 5.9. The value of τ' is 1.7. From the figure we can see that $\rho_{SpMxV(sun,crs,O_{brgc})}(\tau)$ dominates $\rho_{SpMxV(sun,crs,O_{similarity})}(\tau)$ when $1 \leq \tau \leq 1.04$. $\rho_{SpMxV(sun,crs,O_{similarity})}(\tau)$ dominates $\rho_{SpMxV(sun,crs,O_{brgc})}(\tau)$ for all other values of τ. $\rho_{SpMxV(sun,crs,O_{similarity})}(\tau)$ dominates $\rho_{SpMxV(sun,crs,O_{intersection})}(\tau)$, $\rho_{SpMxV(sun,crs,O_{improvement})}(\tau)$, and $\rho_{SpMxV(sun,crs,O_{natural})}(\tau)$ for all values of τ. Furthermore $\rho_{SpMxV(sun,crs,O_{similarity})}(1.06) = 0.96$, which means $SpMxV(sun, crs, O_{similarity})$ can multiply 96% of test matrices with a dense vector within a factor 1.06 of best performing $SpMxV$ on sun platform with CRS scheme. So, we can say that $SpMxV(sun, crs, O_{similarity})$ performs best among these five $SpMxV$s.

5.8.2 *SpMxV*s on Sun Platform with BCRS Scheme

A graph showing the performance of $SpMxV(sun, bcrs, O_{natural})$, $SpMxV(sun, bcrs, O_{intersection})$, $SpMxV(sun, bcrs, O_{similarity})$, $SpMxV(sun, bcrs, O_{brgc})$ and $SpMxV(sun, bcrs, O_{improvement})$ is shown in Figure 5.10. The value of τ' is 1.7. From the figure we can see that $\rho_{SpMxV(sun,bcrs,O_{similarity})}(\tau)$ dominates $\rho_{SpMxV(sun,bcrs,O_{improvement})}(\tau)$ when $1 \leq \tau \leq$

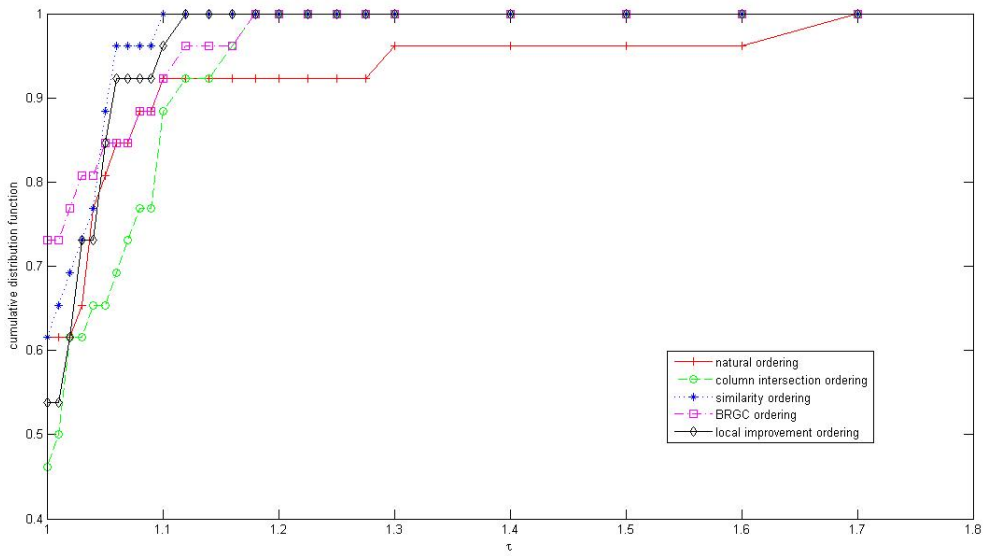

Figure 5.9: Performance of *SpMxV*s on sun platform with CRS scheme.

1.04. $\rho_{SpMxV(sun,bcrs,O_{improvement})}(\tau)$ dominates $\rho_{SpMxV(sun,bcrs,O_{similarity})}(\tau)$ for all other values of τ. $\rho_{SpMxV(sun,bcrs,O_{improvement})}(\tau)$ dominates $\rho_{SpMxV(sun,bcrs,O_{intersection})}(\tau)$, $\rho_{SpMxV(sun,bcrs,O_{brgc})}(\tau)$, and $\rho_{SpMxV(sun,bcrs,O_{natural})}(\tau)$ for all values of τ.

Furthermore $\rho_{SpMxV(sun,bcrs,O_{improvement})}(1.05) = 1.0$, which means $SpMxV(sun, bcrs, O_{improvement})$ can multiply all test matrices with a dense vector within a factor 1.05 of best performing *SpMxV* on sun platform with BCRS scheme. So, we can say that $SpMxV(sun, bcrs, O_{improvement})$ performs best among these five *SpMxV*s.

5.8.3 *SpMxV*s on Sun Platform with FSB2 Scheme

A graph showing the performance of $SpMxV(sun, fsb2, O_{natural})$, $SpMxV(sun, fsb2, O_{intersection})$, $SpMxV(sun, fsb2, O_{similarity})$, $SpMxV(sun, fsb2, O_{brgc})$ and $SpMxV(sun, fsb2, O_{improvement})$ is shown in Figure 5.11. The value of τ' is 1.7. $\rho_{SpMxV(sun,fsb2,O_{similarity})}(\tau)$ dominates $\rho_{SpMxV(sun,fsb2,O_{intersection})}(\tau)$, $\rho_{SpMxV(sun,fsb2,O_{improvement})}(\tau)$, $\rho_{SpMxV(sun,fsb2,O_{brgc})}(\tau)$, and $\rho_{SpMxV(sun,fsb2,O_{natural})}(\tau)$ for all values of τ.

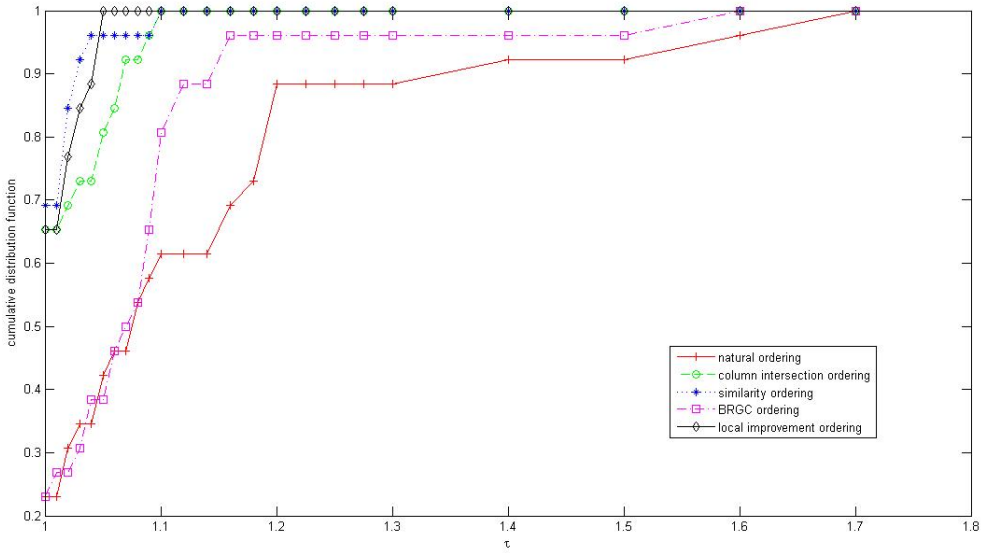

Figure 5.10: Performance of *SpMxV*s on sun platform with BCRS scheme.

Furthermore $\rho_{SpMxV(sun,fsb2,O_{similarity})}(1.1) = 1.0$, which means $SpMxV(sun, fsb2, O_{similarity})$ can multiply all test matrices with a dense vector within a factor 1.1 of best performing *SpMxV* on sun platform with FSB2 scheme. So, we can say that $SpMxV(sun, fsb2, O_{similarity})$ performs best among these five *SpMxV*s.

5.8.4 *SpMxV*s on Sun Platform with FSB3 Scheme

A graph showing the performance of $SpMxV(sun, fsb3, O_{natural})$, $SpMxV(sun, fsb3, O_{intersection})$, $SpMxV(sun, fsb3, O_{similarity})$, $SpMxV(sun, fsb3, O_{brgc})$ and $SpMxV(sun, fsb3, O_{improvement})$ is shown in Figure 5.12. The value of τ' is 1.7. $\rho_{SpMxV(sun,fsb3,O_{similarity})}(\tau)$ dominates $\rho_{SpMxV(sun,fsb3,O_{intersection})}(\tau)$, $\rho_{SpMxV(sun,fsb3,O_{improvement})}(\tau)$, $\rho_{SpMxV(sun,fsb3,O_{brgc})}(\tau)$, and $\rho_{SpMxV(sun,fsb3,O_{natural})}(\tau)$ for all values of τ.

Furthermore $\rho_{SpMxV(sun,fsb3,O_{similarity})}(1.1) = 1.0$, which means $SpMxV(sun, fsb3, O_{similarity})$ can multiply all test matrices with a dense vector within a factor 1.1 of best performing *SpMxV* on sun platform with FSB3 scheme. So, we can say that $SpMxV(sun, fsb3,$

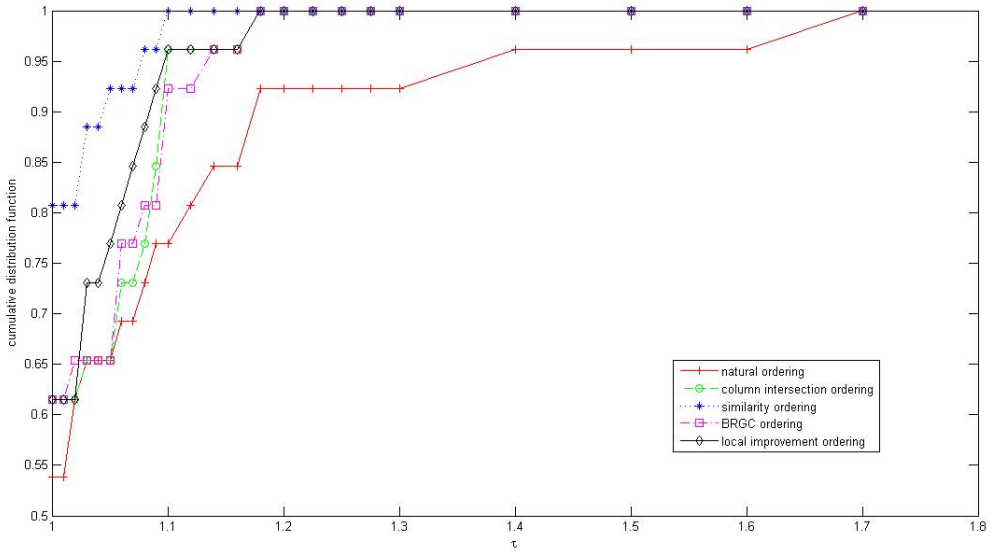

Figure 5.11: Performance of *SpMxV*s on sun platform with FSB2 scheme.

Table 5.4: Optimal *SpMxV* on sun platform.

τ	$\rho_{SpMxV(sun,crs,O_{similarity})}(\tau)$	$\rho_{SpMxV(sun,bcrs,O_{improvement})}(\tau)$	$\rho_{SpMxV(sun,fsb2,O_{similarity})}(\tau)$	$\rho_{SpMxV(sun,fsb3,O_{similarity})}(\tau)$
1	0.384615385	0.346153846	0.576923077	0.384615385
1.05	0.384615385	0.423076923	0.653846154	0.576923077
1.1	0.538461538	0.615384615	0.807692308	0.884615385
1.15	0.692307692	0.846153846	0.961538462	0.923076923
1.2	0.730769231	0.923076923	1	1
1.5	1	1	1	1

$O_{similarity}$) performs best among these five *SpMxV*s.

5.8.5 Optimal *SpMxV* on Sun Platform

Now we will compare of four *SpMxV*s ($SpMxV(sun,crs,O_{similarity})$, $SpMxV(sun,bcrs,O_{improvement})$, $SpMxV(sun,fsb2,O_{similarity})$, and $SpMxV(sun,fsb3,O_{similarity})$) to find out the optimal *SpMxV* on sun platform. We present the comparison data among these four *SpMxV*s in Table 5.4.

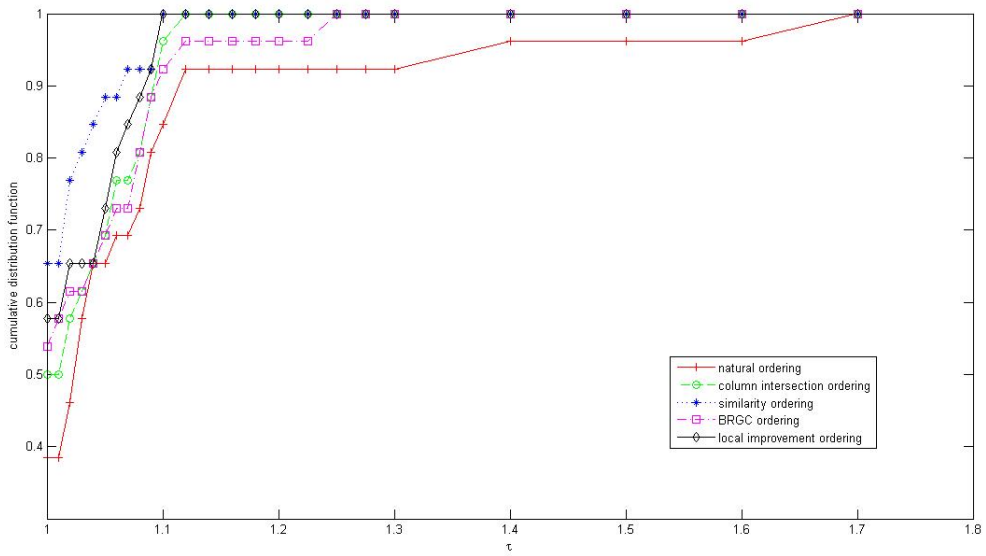

Figure 5.12: Performance of *SpMxV*s on sun platform with FSB3 scheme.

$\rho_{SpMxV(sun,fsb3,O_{similarity})}(\tau)$ dominates $\rho_{SpMxV(sun,fsb2,O_{similarity})}(\tau)$ for a small interval of $\tau\,(1.10, 1.12)$. $\rho_{SpMxV(sun,fsb2,O_{similarity})}(\tau)$ dominates $\rho_{SpMxV(sun,fsb3,O_{similarity})}(\tau)$ for all other values of τ. $\rho_{SpMxV(sun,fsb2,O_{similarity})}(\tau)$ also dominates $\rho_{SpMxV(sun,crs,O_{similarity})}(\tau)$, and $\rho_{SpMxV(sun,bcrs,O_{improvement})}(\tau)$ for all values of τ. So we can infer that $SpMxV(sun, fsb2, O_{similarity})$ is the best $SpMxV$ for sun platform.

If we consider the sum of total CPU time as the performance metric then BRGC ordering performs the best in CRS scheme. $SpMxV(sun, crs, O_{brgc})$ improves 4% CPU time over $SpMxV(sun, crs, O_{natural})$. In the other storage schemes, the performance of BRGC and similarity ordering are almost same. For example, both $SpMxV(sun, fsb2, O_{similarity})$ and $SpMxV(sun, fsb3, O_{similarity})$ improves less than 1% CPU time improvement over $SpMxV(sun, fsb2, O_{brgc})$ and $SpMxV(sun, fsb3, O_{brgc})$ respectively.

5.9 Summary

$SpMxV(compaq, fsb3, O_{brgc})$, $SpMxV(ibm, fsb3, O_{brgc})$, and$SpMxV(sun, fsb2, O_{similarity})$ are the best $SpMxV$ on compaq, ibm, and sun platform respectively. So we can say that FSB is the best storage scheme for sparse matrices. The choice of l depends on the computing platform.

BRGC ordering algorithm performs very well on compaq platform. It also perform very well on ibm platform for both *CRS* and *BCRS* storage schemes. But in FSB2 and FSB3 schemes on ibm platform, its performance is close to that of similar ordering and local improvement ordering. Finally, on sun platform it performs poorly. Both compaq and ibm have same size of L2 cache. But compaq has larger L1 cache. In case of mapping function, compaq has 16-way set associative L2 cache . Whereas, ibm has 8-way set associative L2 cache. As mentioned in Chapter 4, the widths of free rectangles of sparse matrix A (where A is preprocessed by BRGC ordering algorithm) is expected to be proportional with the N of N-way set associative cache of the computing platform. So the probability of self conflict cache miss is inversely proportional with the number of N of N-way set associative.

The L2 cache of sun platform can be 4-way set associative or direct mapped. We are unable to figure out the exact mapping function used in this cache. If it uses direct mapped function then sparse matrix-vector multiplication code may not reuse the elements of x effectively which increases memory traffic. Or in case of using 4-way set associative mapping function, 4 cache lines per set may not be enough to reduce self conflict cache misses during sparse matrix-vector multiplication when the sparse matrix is preprocessed using BRGC ordering. Furthermore, both compaq and ibm have L2 cache of size two times larger than the L2 cache in sun. These can be the reasons for not performing as expected of BRGC ordering.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Sparse matrix-vector multiplication ($y = Ax$) is one of the most performance-critical operation of many scientific applications. It often performs poorly on superscalar architecture. This poor performance is due to

- A random distribution of nonzeros of A produces poor data locality in accessing either x or y during computing Ax.
- The memory system is slower than the processor speed.
- The row and column indices of the nonzeros of A are stored explicitly and these coordinates consumes memory bandwidth during sparse matrix-vector multiplication.

Many researchers have worked on the efficient implementation of sparse matrix-vector multiplication. Permuting the rows or columns of A is one way to improve the data locality of x or y during sparse matrix-vector multiplication. One common approach is to store A in a number of fixed size blocks or tile blocks or nonzero blocks. If the sparse matrix is stored in a number of fixed size blocks or tile blocks or nonzero blocks then the code for computing sparse matrix-vector multiplication can use loop unroll and jam [34] [20] [2]. Recently, cache blocking and register blocking have used to improve data locality

in both x and y during sparse matrix-vector multiplication [19] [35]. If a combination of these techniques are used then the performance of sparse matrix-vector multiplications is expected to improve.

If the nonzeros of a sparse matrix are very sparse or the number of nonzero blocks is very high then permuting the rows or columns of that sparse matrix is necessary. Because by permuting columns or rows of a sparse matrices we can expect to improve the performance of all techniques mentioned above.

The main contribution of this thesis is the proposal for a new column ordering algorithm based on binary reflected gray code namely BRGC ordering. The features of BRGC ordering are given below.

- Given this column ordering, both the temporal and the spatial locality of x improves during sparse matrix-vector multiplication, if we access the sparse matrix row wise. In contrast, other column ordering algorithms consider only the temporal locality of x.

- This column ordering algorithm reveals a certain sparsity structure (describe in Section 4.4.5) of a sparse matrix. So we can expect to store a sparse matrix after BRGC ordering by a number of blocks or tile blocks or nonzero blocks efficiently. Other column ordering algorithms try to minimize the number of nonzero blocks but do not necessarily form any sparsity pattern of the sparse matrices.

- We found this ordering competitive with other column ordering algorithms considering the number of nonzero blocks.

- BRGC ordering produces the same column ordering if the columns are randomly permuted. Furthermore this ordering does not change the sparsity structure of a banded matrix much.

Based on our experimental results on a number of sparse matrices we can conclude that fixed-size block storage scheme performs better than CRS and BCRS schemes on all computing platforms. We also found that BRGC ordering performs better than other column ordering algorithms during sparse matrix-vector multiplication on both compaq and ibm platforms.

6.2 Future Direction

For future research on this work we would like to consider:

- The application of BRGC ordering to improve data locality during sparse matrix-vector multiplication ($y = Ax$) is a new approach. The applicability of this approach to other sparse matrix problems requires further investigation. We would like to find out the relationship among the number of cache misses, cache size, cache line size and mapping function, when the input sparse matrix is preprocessed by BRGC ordering algorithm.

- We would like to use register blocking and cache blocking method in sparse matrix-vector multiplication where the columns of sparse matrix is ordered by BRGC ordering algorithm [19] [35].

- We would like to measure the performance of BRGC ordering on storage schemes of sparse matrices where sparse matrices are stored by a number of fixed size blocks or tile blocks.

Bibliography

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[2] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. Performance optimization and modeling of blocked sparse kernels. *Int. J. High Perform. Comput. Appl.*, 21(4):467–484, 2007.

[3] Thomas F. Coleman and Jorge J. Moré. Estimation of sparse jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20(1):187–209, 1983.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. McGraw-Hill, 2001.

[5] Tim Davis. University of Florida Sparse Matrix Collection, url: http:// www.cise.ufl.edu/ research/ sparse/. Access Date: April 10, 2008.

[6] UltraSPARC Processor Documentation. http:// www.sun.com/ processors/ documentation.html. Access Date: April 10, 2008.

[7] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[8] Anand Ekambaram and Eurpides Montagne. An Alternative Compressed Storage Format for Sparse Matrices. In *Computer and Information Science*, volume 2869 of Lecture Notes in Computer Science, pages 196–203, 2003.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[10] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What Color Is Your Jacobian? Graph Coloring for Computing Derivatives. *SIAM Review*, 47(4):629–705, 2005.

[11] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl*, 13:333–356, 1992.

[12] W. D. Gropp, D. K. Kasushik, D. E. Keyes, and B. F. Smith. Towards realistic bounds for implicit CFD codes. *Proceedings of Parallel Computational Fluid Dynamics*, pages 241–248, 1999.

[13] Geir Gundersen and Trond Steihaug. Data structures in Java for matrix computations. *Concurrency-Practice and Experience.*, 16(8):799–815, 2004.

[14] V. Carl Hamacher, Zvonko G. Vranesic, and Safwat G. Zaky. *Computer Organization, 4th Edition*. McGraw-Hill, 1996.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Third Edition*. Morgan Kaufmann Publishers, 2003.

[16] D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Modeling data locality for the sparse matrix-vector product using distance measures. *Journal of Parallel Computing*, 27:897–912, 2001.

[17] Shahadat Hossain. On Efficient Storage of Sparse Matrices. In *the Proceedings of ICCSE*, pages 1–7. Hasan Dag, Yuefan Deng (Eds.), 2006.

[18] Shahadat Hossain. A New Data Structure for Multiplying a Sparse Matrix with a Dense Vector. In *Proceedings of the International Conference on Computational and Mathematical Methods in Science and Engineering*, pages 197–204. CMMSE, 2007.

[19] Eun-Jin Im. Optimizing the performance of sparse matrix-vector multiplication. *PhD Thesis, University of California Berkeley*, 2000.

[20] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18:135–158, 2004.

[21] Donald L. Kreher and Douglas R. Stinson. *Combinatorial Algorithms :Generation, Enumeration, and Search*. CRC Press, 1999.

[22] Amy N. Langville and Carl D. Meyer. A reordering for the PageRank problem. *SIAM J. Sci. Comput.*, 27(6):2112–2120, 2006.

[23] The Mathworks. http:// www.mathworks.com/. Access Date: April 10, 2008.

[24] J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix-vector product computations using unroll-and-jam. *International Journal of High Performance Computing Applications*, 18:225–236, 2004.

[25] Rajesh Nishtala, Richard Vuduc, James W. Demmel, and Katherine A. Yelick. When cache blocking sparse matrix vector multiply works and why. In *In Proceedings of the PARA04 Workshop on the State-of-the-art in Scientific Computing*, 2004.

[26] Linda Null and Julia Lobur. *The Essentials of Computer Organization and Architecture, 4th Edition*. Jones and Bartlett, 2003.

[27] Advanced Micro Devices AMD-Global Provider of Innovative Microprocessor. http:// www.amd.com/ us-en/ Processors/ TechnicalResources/. Access Date: April 10, 2008.

[28] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Performance optimization of irregular codes based on the combination of reordering and blocking techniques. *Parallel Comput.*, 31(8+9):858–876, 2005.

[29] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, New York, NY, USA, 1999. ACM.

[30] Intel Processors. http:// www.intel.com/ products/ processor/. Access Date: April 10, 2008.

[31] Yousef Saad. *Iterative methods for sparse linear systems, 2nd Edition*. SIAM, 2003.

[32] W. Richard Stevens. *Advanced Programming in the Unix Environment*. Addison-Wesley, 1999.

[33] O. Temam and W. Jalby. Characterizing the behaviour of sparse algorithms on caches. *Proceedings of Supercomputing 92*, pages 578–587, 1992.

[34] S. Toledo. Improving Memory-System Performance of Sparse Matrix-Vector Multiplication. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[35] Richard Vuduc. Automatic performance tuning of sparse matrix kernels. *PhD Thesis, University of California Berkeley*, 2003.

[36] Richard Vuduc and Hyun-Jin Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High Performance Computing and Communications*, volume 3726 of Lecture Notes in Computer Science, pages 807–816, 2005.