CACHE LOCALITY EXPLOITING METHODS AND MODELS FOR SPARSE MATRIX-VECTOR MULTIPLICATION

A THESIS SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING AND
INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Kadir Akbudak
September, 2009

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Cevdet Aykanat (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Ayhan Altıntaş

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Özcan Öztürk

Approved for the Institute of Engineering and Science:

Prof. Mehmet Baray
Director of the Institute

ABSTRACT

CACHE LOCALITY EXPLOITING METHODS AND MODELS FOR SPARSE MATRIX-VECTOR MULTIPLICATION

Kadir Akbudak
Master in Computer Engineering and Information Science
Supervisor: Prof. Cevdet Aykanat
September, 2009

The sparse matrix-vector multiplication (SpMxV) is an important kernel operation widely used in linear solvers. The same sparse matrix is multiplied by a dense vector repeatedly in these solvers to solve a system of linear equations. High performance gains can be obtained if we can take the advantage of today's deep cache hierarchy in SpMxV operations. Matrices with irregular sparsity patterns make it difficult to utilize data locality effectively in SpMxV computations. Different techniques are proposed in the literature to utilize cache hierarchy effectively via exploiting data locality during SpMxV. In this work, we investigate two distinct frameworks for cache-aware/oblivious SpMxV: single matrix-vector multiply and multiple submatrix-vector multiplies. For the single matrix-vector multiply framework, we propose a cache-size aware top-down row/column-reordering approach based on 1D sparse matrix partitioning by utilizing the recently proposed appropriate hypergraph models of sparse matrices, and a cache oblivious bottom-up approach based on hierarchical clustering of rows/columns with similar sparsity patterns. We also propose a column compression scheme as a preprocessing step which makes these two approaches cache-line-size aware. The multiple submatrix-vector multiplies framework depends on the partitioning the matrix into multiple nonzero-disjoint submatrices. For an effective matrix-to-submatrix partitioning required in this framework, we propose a cache-size aware top-down approach based on 2D sparse matrix partitioning by utilizing the recently proposed fine-grain hypergraph model. For this framework, we also propose a traveling salesman formulation for an effective ordering of individual submatrix-vector multiply operations. We evaluate the validity of our models and methods on a wide range of sparse matrices. Experimental results show that proposed methods and models outperforms state-of-the-art schemes.

Keywords: Cache locality, sparse matrices, matrix-vector multiplication, matrix reordering, computational hypergraph model, hypergraph partitioning, traveling salesman problem.

ÖZET

SEYREK MATRİS-VEKTÖR ÇARPIMINDA ÖNBELLEK YERELLİĞİ SAĞLAYAN YÖNTEM VE MODELLER

Kadir Akbudak
Bilgisayar ve Enformatik Mühendisliği, Master
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Eylül, 2009

Seyrek matris-vektör çarpımı doğrusal denklem sistemi çözen yazılımlarda çok önemli bir çekirdek işlemdir. Aynı seyrek matris, seyrek olmayan bir vektörle çok defa çarpılır. Şu anki teknolojinin sunduğu çok seviyeli önbellekler etkin kullanılırsa, bu çarpma işlemi sırasında önemli performans kazançları olabilmektedir. Lakin düzensiz veri erişimine neden olan matrisler önbellekteki veri yerelliğinin kullanımını olumsuz etkilemektedir. Bu problemi çözmek için önbellek yerelliğini kullanan pek çok yöntem şu zamana kadar sunulmuştur. Bu çalışmada, biz de iki farklı çerçeve sunuyoruz: tek matris-vektor çarpımı ve çoklu matris-vektor çarpımı. Tek matris-vektör çarpımı çerçevesinde, önbelleğin boyutunu dikkate alarak matrisin satır ve sütunlarını yeniden sıralayan ve bu sıralama işlemini hiperçizge bölümleme ile yapan bir yöntem sunuyoruz. Bir de önbelleğin boyutunu dikkate almadan yerelliği sağlayacak bir yöntem öneriyoruz. Ve bu yöntemlere ek olarak sütunları sıkıştırıp alansal yerelliği sağlayan önişleme yöntemi sunuyoruz. Çoklu matris-vektör çarpımı çerçevesinde, matrisi alt matrislere ayırarak veri yerelliğini sağlamaya çalışmayı hedefliyoruz. Yine bu ayırma işleminde de hiperçizge kullanılıyor. Alt matrislerin çarpma sırası da önem taşıdığından veri yerelliğini arttıran bir sıralamayı bulma problemini de seyyar satıcı problemi olarak çözülebileceğini açıklıyoruz. Deneysel sonuçlar bu önerilen çerçeve ve yöntemlerin şu anda kullanılan yöntemlerden daha hızlı çalıştığını göstermektedir.

Anahtar sözcükler: Önbellek yerelliği, seyrek matrisler, matris-vektör çarpımı, matrisi yeniden sıralama, bilişimsel hiperçizge modeli, hiperçizge bölümleme, seyyar satıcı problemi .

Acknowledgement

I would like to express my deepest gratitude to my supervisor Prof. Cevdet Aykanat for his guidance, suggestions, and invaluable encouragement throughout the development of this thesis. His patience, motivation, lively discussions and cheerful laughter provided an invaluable and comfortable atmosphere for our work.

I am grateful to my family and my friends for their infinite moral support and help. I owe special thanks to my friend Enver Kayaaslan.

Finally, I thank TÜBİTAK for supporting grant throughout my master program.

To my family

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Many applications became available in numerical computation on behalf of the developments in computer architecture. Nevertheless, these developments introduced some problems such as performance gap between processor and memory speeds. Also there still exits a trade-off between faster but small memories like cpu caches and slower but larger memories like RAM. As a result, the need of new methods and algorithms for efficient usage of higher levels of memory increased in every area of computational problems.

Efficiency in using higher level memories mainly depend on temporal and spatial localities. According to literature, these localities are provided throughout these two ways: data ordering and iteration ordering.

Here data ordering means in what order the elements are stored; and in the same way, iteration ordering means in which order the stored elements are processed. When the data in consecutive memory locations is accessed with stride one, both spatial and temporal localities can be exploited even in compilers. Such kind of applications are said to be *regular*. On the contrary, it is considerable difficult to utilize the data locality effectively in irregular computations which induce irregular memory access patterns.

The sparse matrix-vector multiplication is one of them and the most important kernel operation in linear solvers for the solution of large, sparse, linear systems of equations. These solvers repeat the matrix-vector multiplication $y \leftarrow Ax$ many times with the same sparse matrix to solve a system of equations. Irregular access pattern

during this multiply operation, causes poor usage of cpu caches in today's memory hierarchy technology. However, sparse matrix-vector multiply operation has a potential to exhibit very high performance gains when temporal and spatial localities discussed in Section 2.2 are respected and exploited.

In this work, we investigate two distinct frameworks for cache-aware/oblivious SpMxV: single matrix-vector multiply and multiple submatrix-vector multiplies. For the single matrix-vector multiply framework, we propose a cache-size aware top-down row/column-reordering approach based on transformation a sparse matrix to a singly-bordered block-diagonal form by utilizing the recently proposed appropriate hypergraph models. We provide an upper bound on the total number of cache misses based on this transformation, and show that the objective in the hypergraph-partitioning-based transformation model exactly corresponds to minimizing this upper bound. We also propose a cache oblivious bottom-up approach based on hierarchical clustering of rows/columns with similar sparsity patterns. Furthermore, a column compression scheme as a preprocessing step which makes these two approaches cache-line-size aware is presented.

The multiple submatrix-vector multiplies framework depends on the partitioning the matrix into multiple nonzero-disjoint submatrices and the ordering of submatrix-vector multiplies. For an effective matrix-to-submatrix partitioning required in this framework, we propose a cache-size aware top-down approach based on 2D sparse matrix partitioning by utilizing the recently proposed fine-grain hypergraph model. We provide an upper bound on the total number of cache misses based on this matrix-to-submatrix partitioning, and show that the objective in the hypergraph-partitioning-based matrix-to-submatrix partitioning exactly corresponds to minimizing this upper bound.

For this framework, we also propose a traveling salesman formulation for an effective ordering of individual submatrix-vector multiply operations. We provide a lower bound on the total number of cache misses based on the ordering of submatrix-vector multiplies, and show that the objective in TSP formulation exactly corresponds to minimizing this lower bound.

We evaluate the validity of our models and methods on a wide range of sparse matrices. Experimental results show that proposed methods and models outperforms

state-of-the-art schemes.

The rest of this thesis is organized as follows: Background material is introduced in Chapter 2. In Chapter 3, we review some of the previous works about iteration/data reordering and matrix transformations for exploting locality. Two frameworks as our contributions in sparse matrix-vector multiplication are described in Chapters 4 and 5. We present the experimental results of these two frameworks and comparisons with some of the previous works in Chapter 6. Finally, the thesis is concluded in Chapter 7.

Chapter 2

Background

In this chapter, we will review several schemes for storing sparse matrices in Section 2.1. Data locality issues during matrix-vector multiplication will be considered in Section 2.2. Then we will review definition of hypergraph and partitioning problems in Section 2.3. We will mention about two hypergraph models for 1D and 2D decomposition of sparse matrices in Section 2.4. Finally, definition of the well known *Travelling Salesman Problem* (TSP) will be told in Section 2.6.

2.1 Data Storage Schemes used in Sparse Matrix-Vector Multiplication

In this chapter we will review an important storage scheme *Compressed Storage by Rows* (CSR) and its variances, *Zig-Zag Compressed Storage by Rows* (ZZCSR), *Incremental Compressed Storage by Rows* (ICSR) and *Zig-Zag Incremental Compressed Storage by Rows* (ZZICSR) for sparse matrix-vector multiplication in Sections 2.1.1, 2.1.2,2.1.3 and 2.1.4, respectively.

There are two main storage schemes for sparse matrix-vector multiply operation. They are *Compressed Storage by Rows* (CSR) and *Compressed Storage by Columns* (CSC) [12, 34]. Each sparse-matrix storage scheme determines a distinct computation scheme for the matrix-vector multiplication. In this thesis, we restrict our

focus on cache-aware/oblivious computation of sparse matrix-vector multiply operation using the CSR storage scheme without loss of generality.

In the following subsections we review four CSR-based sparse-matrix storage schemes.

1. CSR
2. Zig-zag CSR
3. ICSR
4. Zig-zag ICSR

For other types of schemes, books such as Duff, Erisman, and Reid [14] can be investigated.

2.1.1 Compressed Storage by Rows

CSR scheme is widely used in sparse matrix operations. In this scheme and in all the remaining schemes mentioned in this section, only the nonzeros are naively stored without using any structural information. Nonzeros are stored in a row-major format, meanly nonzeros of a row are stored consecutively. This scheme contains three arrays: *nonzero*, *column-index* and *row-start*. The values and the column indices of the nonzeros are stored in row-major order in the $nonzero$ and $column\text{-}index$ arrays in one-to-one manner, respectively. That is, $column\text{-}index[i]$ stores the column-index of the nonzero and the value of this nonzero is stored in $nonzero[i]$. The *row-start* array stores the index of the first nonzero element of each row. This index is used to access both of *nonzero* and *column-index* arrays. Also the original row order of the sparse matrix A is preserved while constructing *row-start* array and similarly the original column order is preserved while constructing *nonzero* and *column-index* arrays; but these preservations of the original orders are not obligatory, we will assume these original orderings in this work. Algorithm 1 shows how the sparse matrix-vector multiplication can be performed using CSR storage scheme.

Algorithm 1 Sparse Matrix-Vector Multiplication using CSR scheme

Require: $nonzero$, $column\text{-}index$ and $row\text{-}start$ arrays of a m by n sparse matrix A
a dense input vector x

1: **for** $i \leftarrow 1$ *to* m **do**
2: $tmp \leftarrow 0$
3: **for** $j \leftarrow row\text{-}start[i]$ *to* $row\text{-}start[i+1]-1$ **do**
4: $tmp \leftarrow tmp + nonzero[i] * x[column\text{-}index[j]]$
5: **end for**
6: $y[i] \leftarrow tmp$
7: **end for**
8: **return** y

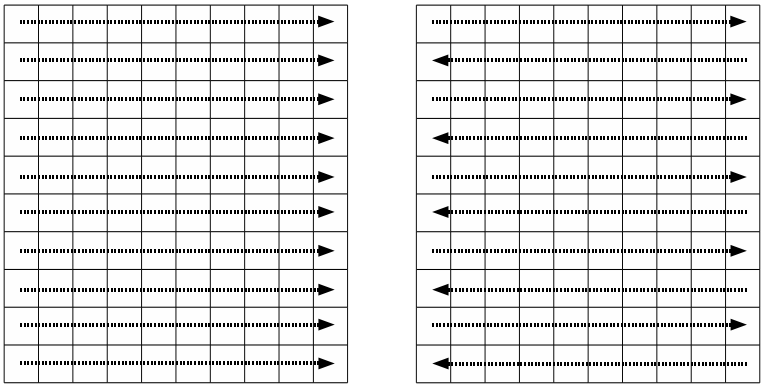

Figure 2.1: Processing order of nonzeros stored using the CSR (on the left) and ZZCSR (on the right) schemes. Arrows denote the storage order of nonzeros of a row.

2.1.2 Zig-Zag Compressed Storage by Rows

Zig-zag CSR (ZZCSR) scheme [40] is similar to CSR. In CSR scheme, multiplication is being performed through increasing index on each row. In ZZCSR, multiplication is being performed through increasing index on even-numbered rows and through decreasing index on odd-numbered rows. In this way, possibility of reuse of recently retrieved $x\text{-}vector$ entries in cache is increased. Figure 2.1 illustrates comparison of these two schemes. This ZZCSR scheme can be simply implemented by reversing the order of elements in *nonzero* and *column-index* arrays of odd-numbered rows in the CSR scheme.

2.1.3 Incremental Compressed Storage by Rows

Incremental Compressed Storage by Rows (ICSR) proposed in [25] and it is reported to decrease instruction overhead by using pointer arithmetic. In addition, the idea behind this storage scheme perfectly fits for matrices having empty rows. In the CSR scheme, all rows, whether they are empty or not, must be present in the *row-start* array. But in ICSR, row indices of the empty rows are not stored at all, because row indices and column indices are calculated by accumulating elements of *row-jump* and *column-diff* arrays on current values. In contrast, these indices are retrieved from *row-start* and *column-index* arrays in CSR scheme. In other words, index of the next row r_i to be processed is calculated by adding *row-jump[i]* to the current row index value. In the same way, the index of the next column c_j to be processed is calculated by adding *column-diff[j]* to the current column index value. Then, row increments are triggered just after column index value becomes greater than number of columns. The number of columns is subtracted from overflowed column index value and used as new column index. New row index is calculated using related element of *row-jump* array and current row index value. These steps can be easily understood from Algorithm 2.

Algorithm 2 Sparse Matrix-Vector Multiplication using ICSR scheme

Require: $nonzero$, $column\text{-}diff$ and $row\text{-}jump$ arrays of a m by n sparse matrix A
a dense input vector x
number of nonzeros nnz in matrix A

1: $i \leftarrow row\text{-}jump[0]$
2: $j \leftarrow column\text{-}diff\,[0]$
3: $k \leftarrow 0$
4: $r \leftarrow 1$
5: $tmp \leftarrow 0$
6: **while** $k < nnz$ **do**
7: $tmp \leftarrow tmp + nonzero[k] * x[j]$
8: $k \leftarrow k + 1$
9: $j \leftarrow j + column\text{-}diff\,[k]$
10: **if** $j \geq n$ **then**
11: $y[i] \leftarrow tmp$
12: $tmp \leftarrow 0$
13: $j \leftarrow j - n$
14: $i \leftarrow i + row\text{-}jump[r]$
15: $r \leftarrow r + 1$
16: **end if**
17: **end while**
18: **return** y

2.1.4 Zig-Zag Incremental Compressed Storage by Rows

Zig-Zag Incremental Compressed Storage by Rows (ZZICSR) [40] combines CSR and zig-zag property. Temporal locality in $x\text{-}vector$ is exploited using the zig-zag property. In addition to this, empty rows are not stored. As a result, this scheme becomes convenient for sparse matrices having a considerable amount of empty rows as well as the temporal locality is achieved for $x\text{-}vector$. Like ZZCSR scheme, this scheme can be implemented by putting negative values in *row-jump* and *column-index* arrays of odd-numbered rows in the ICSR scheme. So that flow of process is reversed in odd-numbered rows.

2.2 Data Locality in Sparse Matrix-Vector Multiply

Here, we will briefly mention about the data locality characteristics of matrix-vector multiply operation $y \leftarrow Ax$ using the CSR scheme as also discussed in [39]. In terms of the A-matrix stored in CSR format, temporal locality is not feasible since the elements of each of the $nonzero$, $column\text{-}index$ ($column\text{-}diff$ in ICSR) and $row\text{-}start$ ($row\text{-}jump$ in ICSR) arrays are accessed only once. Spatial locality is feasible and it is achieved automatically by nature of the CSR scheme since the elements of each of the three arrays are stored and accessed consecutively.

In terms of output vector y, temporal locality is not feasible since each $y\text{-}vector$ result is written only once to the memory. As a different view [39], temporal locality can be considered as feasible but automatically achieved at the register level. Spatial locality is feasible and it is achieved automatically since the $y\text{-}vector$ entry results are stored consecutively.

In terms of input vector x, both temporal and spatial locality are feasible. Temporal locality is feasible since each $x\text{-}vector$ entry may be accessed multiple times. However, exploiting the temporal and spatial locality for the $x\text{-}vector$ is the major concern in the CSR scheme since $x\text{-}vector$ entries are accessed through a $column\text{-}index$ array ($column\text{-}diff$ in ICSR) in a non-contiguous manner.

These locality issues can be solved by reordering rows/columns of matrix A and

the exploitation level of these data localities depends both on the existing sparsity pattern of matrix A and the effectiveness of reordering heuristics.

2.3 Hypergraph Partitioning

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$. Every net $n_j \in \mathcal{N}$ connects a subset of vertices, i.e., $n_j \subseteq \mathcal{V}$. The vertices connected by a net n_j are called its *pins* (i.e., $Pins(n_j)$). Weights can be associated with the vertices. We use $w(v_i)$ to denote the weight of the vertex v_i.

Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, $\Pi = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$ is called a K-way partition of the vertex set $\mathcal{V}$ if each part is nonempty, i.e., $\mathcal{V}_k \neq \emptyset$ for $1 \leq k \leq K$; parts are pairwise disjoint, i.e., $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for $1 \leq k < \ell \leq K$; and the union of parts gives $\mathcal{V}$, i.e., $\bigcup_k \mathcal{V}_k = \mathcal{V}$. A K-way vertex partition of $\mathcal{H}$ is said to satisfy the partitioning constraint if

$$W_k \leq W_{avg}(1 + \varepsilon), \quad \text{for } k = 1, 2, \ldots, K \tag{2.1}$$

In here, the weight W_k of a part $\mathcal{V}_k$ is defined as the sum of the weights of the vertices in that part (i.e., $W_k = \sum_{v_i \in \mathcal{V}_k} w(v_i)$), W_{avg} is the average part weight (i.e., $W_{avg} = (\sum_{v_i \in \mathcal{V}} w(v_i))/K$), and ε represents the predetermined, maximum allowable imbalance ratio.

In a partition Π of $\mathcal{H}$, a net that connects at least one pin (vertex) in a part is said to *connect* that part. *Connectivity set* Λ_j of a net n_j is defined as the set of parts connected by n_j. *Connectivity* $\lambda_j = |\Lambda_j|$ of a net n_j denotes the number of parts connected by n_j. A net n_j is said to be *cut* (*external*) if it connects more than one part (i.e., $\lambda_j > 1$), and *uncut* (*internal*) otherwise (i.e., $\lambda_j = 1$). The set of external nets of a partition Π is denoted as $\mathcal{N}_E$. The partitioning objective is to minimize the cutsize defined over the cut nets. There are various cutsize definitions. The relevant definition is:

$$cutsize(\Pi) = \sum_{n_j \in \mathcal{N}_E} (\lambda_j - 1) \tag{2.2}$$

In here, each cut net n_j contributes $\lambda_j - 1$ to the cutsize. The hypergraph partitioning problem is known to be NP-hard [27].

Recently, multilevel HP approaches [4, 19, 21] have been proposed, leading to successful HP tools hMetis [23] and Patoh [7]. These multilevel heuristics consist of 3 phases: coarsening, initial partitioning, and uncoarsening. In the first phase, a multilevel clustering is applied starting from the original hypergraph by adopting various matching heuristics until the number of vertices in the coarsened hypergraph decreases below a predetermined threshold value. Clustering corresponds to coalescing highly interacting vertices to supernodes. In the second phase, a partition is obtained on the coarsest hypergraph using various heuristcs including FM, which is an iterative refinement heuristic proposed for graph/hypergraph partitioning by Fiduccia and Mattheyses [15] as a faster implementation of the KL algorithm proposed by Kernighan and Lin [24]. In the third phase, the partition found in the second phase is successively projected back towards the original hypergraph by refining the projected partitions on the intermediate level uncoarser hypergraphs using various heuristics including FM.

The *recursive bisection* (RB) paradigm is widely used in K-way hypergraph partitioning and known to be amenable to produce good solution qualities. In the RB paradigm, first, a two-way partition of the hypergraph is obtained. Then, each part of the bipartition is further bipartitioned in a recursive manner until the desired number K of parts is obtained or part weights drop below a given maximum allowed part weight W_{max}. In RB-based hypergraph partitioning, the cut-net splitting scheme [6] is adopted to capture the $\lambda - 1$ cutsize metric given in Equation 2.2. In hypergraph partitioning, balancing the part weights of the bipartition is enforced as the bipartitioning constraint.

The RB paradigm is inherently suitable for partitioning hypergraphs when K is not known in advance. Hence, the RB paradigm can be successfully utilized in clustering rows/columns for cache-size aware row/column reordering.

2.4 Hypergraph Models for Sparse Matrix Partitioning

Recently, several successful hypergraph models and methods are proposed for efficient parallelization of sparse matrix-vector multiplication [5, 6, 9]. The relevant ones are row-net, column-net, and row-column-net models. The row-net and column-net models are proposed and used for 1D rowwise and 1D columnwise partitioning of sparse matrices, respectively, whereas row-column-net model is used for 2D fine-grain partitioning of sparse matrices.

In the *row-net hypergraph model* [5, 6, 9] $\mathcal{H}_{RN}(A) = (\mathcal{V}_{\mathcal{C}}, \mathcal{N}_{\mathcal{R}})$ of matrix A, there exist one vertex $v_j \in \mathcal{V}_{\mathcal{C}}$ and one net $n_i \in \mathcal{N}_{\mathcal{R}}$ for each column c_j and row r_i, respectively. The vertex v_j represents the DAXPY-like operation which multiplies x_j with column c_j and adds the result of this scalar-vector product to the output vector y. The weight $w(v_j)$ of a vertex $v_j \in \mathcal{V}_{\mathcal{R}}$ is set to the number of nonzeros in column c_j. The net n_i connects the vertices corresponding to the columns that have a nonzero entry in row r_i. That is, $v_j \in Pins(n_i)$ if and only if $a_{ij} \neq 0$. Here, n_i represents the $y\text{-}vector$ entry y_i and $Pins(n_i)$ represents the set of scalar multiply results needed to be accumulated in y_i during matrix-vector multiply.

In the *column-net hypergraph model* [5, 6, 9] $\mathcal{H}_{CN}(A) = (\mathcal{V}_{\mathcal{R}}, \mathcal{N}_{\mathcal{C}})$ of matrix A, there exist one vertex $v_i \in \mathcal{V}_{\mathcal{R}}$ and one net $n_j \in \mathcal{N}_{\mathcal{C}}$ for each row r_i and column c_j, respectively. The vertex v_i represents the inner product of row r_i with the input vector x. The weight $w(v_i)$ of a vertex $v_i \in \mathcal{V}_{\mathcal{R}}$ is set to the number of nonzeros in row r_i. Net n_j connects the vertices corresponding to the rows that have a nonzero entry in column c_j. That is, $v_i \in Pins(n_j)$ if and only if $a_{ij} \neq 0$. Here, n_j represents the $x\text{-}vector$ entry x_j and $Pins(n_j)$ represents the set of inner product operations that need x_j during matrix-vector multiply.

In the *row-column-net model* [8] (also called as fine-grain model) $\mathcal{H}_{RCN}(A) = (\mathcal{V}_{\mathcal{Z}}, \mathcal{N}_{\mathcal{RC}})$ of matrix A, there exists one vertex $v_{ij} \in \mathcal{V}_{\mathcal{Z}}$ corresponding to each nonzero a_{ij} in matrix A. In net set $\mathcal{N}_{\mathcal{RC}}$, there exists a row-net n_i^r for each row r_i, and there exists a column-net n_i^c and for each column c_j. The vertex v_{ij} represents the scalar multiply-and-add operation $y_{ij} \leftarrow a_{ij}x_j$. Therefore each vertex is assigned unit weight. The row-net n_i^r connects the vertices corresponding to the nonzeros in the

row r_i, and the column-net n_j^c connects the vertices corresponding to the nonzeros in the column c_j. That is, $v_{ij} \in Pins(n_i^r)$ and $v_{ij} \in Pins(n_j^r)$ if and only if $a_{ij} \neq 0$. Note that each vertex v_{ij} is a pin of exactly two nets. Here, n_j^c represents x_j and $Pins(n_j^c)$ represents the set of scalar multiply-and-add operations that need x_j, whereas n_i^r represents y_i and $Pins(n_i^r)$ represents the set of scalar multiply-and-add results needed to accumulate y_i.

The use of the hypergraphs $\mathcal{H}_{RN}(A)$, $\mathcal{H}_{CN}(A)$ and $\mathcal{H}_{RCN}(A)$ in sparse matrix partitioning for parallelization of matrix-vector multiply operation is described into detail in [6, 9]. In particular, it has been shown that the partitioning objective (2.2) corresponds exactly to the total communication volume, whereas the partitioning constraint (2.1) corresponds to maintaining a computational load balance for a given number K of processors.

In [3], it is shown that a K-way partition of 1D $\mathcal{H}_{RN}(A)$ and $\mathcal{H}_{CN}(A)$ models can be decoded as inducing row-and-column reordering for transforming matrix A into a K-way *singly-bordered block-diagonal* (SB) form. Here we will briefly describe how a K-way partition of column-net model can be decoded a row and column ordering for this purpose and a dual discussion holds for row-net model.

A K-way vertex partition $\Pi = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$ of $\mathcal{H}_{CN}(A)$ is considered as inducing a $(K+1)$-way partition $\{\mathcal{N}_1, \ldots, \mathcal{N}_K; , \mathcal{N}_E\}$ on the net set of $\mathcal{H}_{CN}(A)$. Here $\mathcal{N}_k$ denotes the set of internal nets of vertex part $\mathcal{V}_k$, for each $k = 1, 2, \ldots, K$, whereas $\mathcal{N}_E$ denotes the set of external nets. The vertex partition is decoded as a partial row reordering of matrix A such that the rows associated with vertices in $\mathcal{V}_{k+1}$ are ordered after the rows associated with vertices $\mathcal{V}_k$, $k = 1, 2, \ldots, K-1$. The net partition is decoded as a partial column reordering of matrix A such that the columns associated with nets in $\mathcal{N}_{k+1}$ are ordered after the columns associated with nets in $\mathcal{N}_k$, $k = 1, 2, \ldots, K-1$, where the columns associated with the external nets are ordered last to constitute the column border.

The above-mentioned approach of obtaining a K-way SB form of a K-way partition of the column-net model can be extended to obtain a hierarchic SB form of a K-way partition produced by using the RB paradigm. In this transformation, the bipartition obtained at each RB step is decoded as inducing a 2-way SB form, where these 2-way SB forms are nested according to RB hierarchy. SB forms for different

K values of the *psse1* matrix are shown in Appendix B.

2.5 Breadth-First-Search-Based Algorithm for Row/Column Reordering

Breadth-First Search (BFS) algorithm systematically explores edges of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, level by level to discover every vertex reachable from the source vertex s. All neighbors of a vertex v are visited before any sibling of v. This process is repeated for every unvisited vertex. The running-time complexity is $O(|\mathcal{V} + \mathcal{E}|)$.

A BFS-like algorithm can be used to order rows/columns of a sparse matrix. The resultant row order and column order are first-come first-serve basis. While processing a row, all required columns are reordered consecutively. If a column is already reordered by a previous row, it cannot be reordered again by other rows visited after. This approach exploits spatial locality more than temporal locality.

The first row to be processed can be selected as randomly or the row which has maximum degree can be source. In the algorithm, rows with maximum degrees are selected throughout the processing of all rows.

The Algorithm 3 shows an algorithm that orders rows and columns of a given matrix A . The $rowOrdered$ array is used to determine whether a row is already processed or not, similarly $colOrdered$ array is used for columns. The $newRowOrder$ and $newColumnOrder$ arrays contain corresponding new row and column indices of current row and column indices, respectively. For a given sparse matrix A, $rows(A)$ denotes index set of rows of matrix A. $columns(r)$ denotes set of column indices of nonzeros in the row r. Similarly, $rows(c)$ denotes set of row indices of nonzeros in the column c.

Algorithm 3 Modified BFS Algorithm for Row/Column Ordering

Require: Sparse matrix A
1: $rowOrdered[*] \leftarrow false$
2: $columnOrdered[*] \leftarrow false$
3: $columnIndex \leftarrow 1$
4: $rowIndex \leftarrow 1$
5: sort $rows(A)$ by nonincreasing number of nonzeros in a row
6: **for all** $r \in rows(A)$ in sorted order **do**
7: **if** $rowOrdered[r] = false$ **then**
8: $ENQUEUE(\mathcal{Q}, r)$
9: **while** $Q \neq \emptyset$ **do**
10: $r \leftarrow head[\mathcal{Q}]$
11: $rowOrdered[r] \leftarrow true$
12: $newRowOrder[r] \leftarrow rowIndex$
13: $rowIndex \leftarrow rowIndex + 1$
14: **for all** $c \in columns(r)$ **do**
15: **if** $columnOrdered[c] = false$ **then**
16: $columnOrdered[c] \leftarrow true$
17: $newColumnOrder[c] \leftarrow columnIndex$
18: $columnIndex \leftarrow columnIndex + 1$
19: **for all** $r2 \in rows(c)$ **do**
20: **if** $rowOrdered[r2] = false$ **then**
21: $ENQUEUE(\mathcal{Q}, r2)$
22: **end if**
23: **end for**
24: **end if**
25: **end for**
26: $DEQUEUE(\mathcal{Q})$
27: **end while**
28: **end if**
29: **end for**

2.6 Travelling Salesman Problem

Travelling salesman problem (TSP) is one of the most popular problems studied in combinatorial optimization. There are many other problems that can be cast to TSP. In this section, we will confine the problem definition on symmetric TSP with non-metric distances. Informal definition can be as follows:

Definition 1 *Given a list of cities and pairwise distances, find the shortest tour that passes all cities exactly once.*

TSP can also be modelled as a graph. Graph's vertices correspond to cities, and edges correspond to connections between city pairs. The edge weights are the distances

between cities. The resultant graph may not be complete graph, there may not be edges between some vertices, or the graph may be defined as complete but some edge weights may be zero representing the non-existing edges. When this graph is represented by a adjacency matrix $\mathcal{W}$, entries of this matrix are edge weights so this $\mathcal{W}$ matrix can be called as a *weight matrix*. The weight w_{ij} represents the distance between vertices v_i and v_j. Then aim is finding a permutation of vertices $\Pi =< \Pi(1), \Pi(2) \dots \Pi(n) >$ that minimizes following objective function:

$$L = w_{v_{\Pi(n)} v_{\Pi(1)}} + \sum_{i=1}^{n-1} w_{v_{\Pi(i)} v_{\Pi(i+1)}} \tag{2.3}$$

where L is the total length of the tour. Minimization and maximization of L are same problems. If each edge weight w_{ij} is subtracted from largest edge weight $(w^{max} + 1)$, minimizing 2.3 becomes maximization of length of the tour.

In the case of finding a path instead of a shortest tour, the tour can be converted to path by removing an edge. This edge must have the maximum weight so that the length of the path is minimized.

TSP is proved to be a member of the set of NP-complete problems. So the most efficient way solving this problem, is developing heuristics. The Lin-Kernighan heuristic [28] is the most effective method considered in the literature for generating optimal or near-optimal solutions for the symmetric traveling salesman problem. So, in this work, a TSP solver library [18] implementing the heuristic proposed by Kernighan and Lin [28] is used.

Chapter 3

Related Work

In the literature, there are numerous studies regarding computation and data ordering. They can be classified into two categories according to the kind of access pattern of applications. For applications having regular access pattern, compiler optimizations become more available for computation and data ordering [29, 26]. For applications whose access pattern changes through time, static improvements in compilers start being insufficient and this kind of applications are referred as irregular [33]. As a result dynamic orderings are required [13, 11, 1, 10, 31].

Reordering rows/columns of sparse matrices to exploit locality during sparse matrix-vector multiplication is a special case of this general computation(or iteration) and data ordering problem. Here, if we consider a matrix A stored in CSR scheme, computation order corresponds to row order of matrix A and data order corresponds to column order. Adding that *dynamic* reordering algorithms work as inspector-executor method used by Saltz [30]. This corresponds to matrix reordering algorithms that are run before multiplication, they do not run at run-time an it is not necessary because we have the whole matrix that determines computation and data orders. The example code given in Figure 3.1 is the general case of the problem and the code in Figure 3.2 is the special case of computation and data ordering.

Initial studies start with the work of Ding and Kennedy [13]. They propose a dynamic approach for both data and computation reordering. They present *consecutive packing* (CPACK) where data is ordered when a computaion require it and after then

for($i = 1\ to\ 7$)
$\ldots C[a[i]] \ldots$
$\ldots C[b[i]] \ldots$
endfor

$$\begin{array}{c} \\ a \\ b \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \left[\begin{array}{ccccccc} 3 & 2 & 1 & 3 & 6 & 4 & 2 \\ 1 & 5 & 4 & 2 & 3 & 6 & 6 \end{array}\right] \end{array}$$

$$\begin{array}{c} \\ C \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \left[\begin{array}{cccccc} A & B & C & D & E & F \end{array}\right] \end{array}$$

Figure 3.1: Example for irregular code that are the focus in computation and data ordering problem. C array is accessed through two index arrays a and b. These two arrays cause indirection so the code shows irregular access patter.

this reordered data no more moved.

Space-filling curves (e.g., Morton, Hilbert) can be used for computation ordering [20]. They are also used for data ordering [10] and are shown to be successful in improving locality. Haase et al. [16] use Hilbert space-filling curve to order nonzeros of the matrix A along the curve. They report speedups of up to 50 percent according to the original CSR scheme.

Hwansoo and Tseng [17] propose an algorithm, Z-SORT, for reordering computations and another algorithm, GPART, for reordering data at run-time in. Z-SORT algorithm finds a new loop iteration order using Z-curve which is a kind of space-filling curve. GPART orders elements in data array to exploit spatial locality. After data ordered by GPART, Z-SORT finds suitable computation ordering respecting the order found by GPART.

Strout and Hovland [35] give metrics that guide while selecting the best ordering method according to irregularity of applications. They introduce a temporal hypergraph model for ordering iterations to exploit temporal locality. They also generalize spatial locality graph model to spatial locality hypergraph model to encompass the applications having multiple arrays that are accessed irregularly. Additionally, they propose a modified algorithm like Breadth-First Search for ordering data and iterations simultaneously whereas Breadth-First Search is used for only data ordering in [1].

In a very recent work by Yzelman and Bisseling [40], row/column reordering

for $i = 1$ *to* m do
    $sum = 0$
    for $k = row\text{-}start[i]$ *to* $row\text{-}start[i+1] - 1$ do
        $sum = sum + nonzero[i] * x[column\text{-}index[k]]$
    endfor
    $y[i] = sum$
endfor

$$A = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{bmatrix} a_{11} & & & a_{14} & \\ & a_{22} & a_{23} & a_{24} & \\ & a_{32} & & a_{34} & a_{35} \end{bmatrix} \end{array}$$

$$column-index \quad \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{matrix} \\ \begin{bmatrix} 1 & 4 & 2 & 3 & 4 & 2 & 4 & 5 \end{bmatrix} \end{array}$$

$$x \quad \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix} \end{array}$$

Figure 3.2: Sparse matrix-vecto multiply algorithm based on using the CSR scheme. x array is the $x\text{-}vector$ in the sparse matrix-vector multiplication $y \leftarrow Ax$.

scheme is proposed based on partitioning the rownet hypergraph model of an input sparse matrix A. They achieve spatial locality on $x\text{-}vector$ entries by clustering the columns with similar sparsity pattern. They also exploit temporal locality for $x\text{-}vector$ entries by using zig-zag property of ZZCSR and ZZICSR schemes mentioned in Section 2.1.2 and 2.1.4 respectively.

White and Sadayappan [39] use graph model for ordering rows and columns using the Metis graph partitioner [22]. Unfortunately they report that they could not gain improvement for reordered matrices compared to original matrices during sparse matrix-vector multiplication.

Pınar and Heath [32] try to permute nonzeros of the matrix A into contiguous blocks to decrease the number of indirections and they formulate this problem as an instance of the traveling salesman problem.

Beside reordering transformations, locality can be exploited by a cache-aware method called *cache blocking* like in OSKI framework [37] and in [38]. Our contributions in this thesis have both cache-aware and cache-oblivious aspects and they will be presented in Chapters 4 and 5.

Chapter 4

Single Matrix-Vector Multiply Framework

This is the conventional approach to matrix-vector multiply operation. The $y\text{-}vector$ results are computed simply by multiplying matrix A with $x\text{-}vector$, i.e.,

$$y \leftarrow Ax \tag{4.1}$$

The objective in this scheme is to reorder the columns and rows of matrix A for maximizing the exploitation of temporal and spatial locality in accessing $x\text{-}vector$ entries. Recall that temporal locality in accessing $y\text{-}vector$ entries is not feasible, whereas spatial locality is achieved automatically because $y\text{-}vector$ results are stored and processed consecutively. Reordering the rows with similar sparsity pattern nearby increases the possibility of exploiting temporal locality in accessing $x\text{-}vector$ entries. Reordering the columns with similar sparsity pattern nearby increases the possibility of exploiting spatial locality in accessing $x\text{-}vector$ entries. This row/column reordering problem can be considered as a row/column clustering problem and this clustering process can be achieved in two distinct ways: top-down and bottom-up. In this section, we first propose and discuss a cache-size aware top-down approach based on 1D partitioning of sparse matrix A and then a cache oblivious bottom-up approach based on hierarchical clustering of rows with similar patterns. Then we propose a column compression scheme as a preprocessing step which makes these two approaches cache-line-size aware.

4.1 1D Decomposition of Sparse Matrices

We consider a row/column reordering which permutes a given matrix A into a K-way SB form

$$A_{SB} = PAQ = \begin{bmatrix} A_{11} & & & & A_{1B_1} \\ & A_{22} & & & A_{2B_2} \\ & & \ddots & & \vdots \\ & & & A_{KK} & A_{KB_K} \end{bmatrix} = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_K \end{bmatrix}. \tag{4.2}$$

where the CSR data structure associated with each submatrix A_k as follows

$$A_k = [0 \ldots 0 \; A_{kk} \; 0 \ldots 0 \; A_{kB_k}]. \tag{4.3}$$

Here A_{kk} denotes the kth diagonal block of A_{SB}, whereas A_B denotes the column border as follows

$$A_B = \begin{bmatrix} A_{1B_1} \\ A_{2B_2} \\ \vdots \\ A_{KB_K} \end{bmatrix}. \tag{4.4}$$

Each column in the border is called a *row-coupling column* or simply *coupling column*. Let $\lambda(c_j)$ denote the number of submatrices that contain at least one nonzero of column c_j of matrix A_{SB}, i.e.,

$$\lambda(c_j) = |\{A_k : c_j \in A_k\}| \tag{4.5}$$

In this notation, a column c_j is coupling column if $\lambda(c_j) > 1$.

The following theorem gives the guidelines for a "good" A-to-A_{SB} transformation.

Theorem 1 *Given a K-way SB form of matrix A such that every submatrix A_k can fit into the cache. Then the number $\Phi(A_{SB})$ of cache misses due to the access of x-vector entries can be upperbounded as*

$$\Phi(A_{SB}) \leq \sum_{c_j} \lambda(c_j) \tag{4.6}$$

in case of full-associativity of cache is assumed.

Proof Since each submatrix A_k fits into the cache, each $x\text{-}vector$ entry corresponding to a nonzero column of matrix A_k will be loaded to the cache only once during the $y_k = A_k x$ multiply, under the full-associativity assumption. Therefore for a column c_j maximum number of cache misses that can occur is bounded above by $\lambda(c_j)$ due to the access of corresponding $x\text{-}vector$ entry x_j. Thus, the number $\Phi(A_{SB})$ of cache misses due to the access of $x\text{-}vector$ entries cannot exceed $\sum_{c_j} \lambda(c_j)$. Note that this upperbound also holds for the larger cache-line sizes.

Theorem 1 leads us to a cache-size aware top-down row/column reordering through an A-to-A_{SB} transformation which minimizes the sum $\sum_{c_j} \lambda(c_j)$ of the λ values of columns. Here, minimizing objective relates to minimizing the cache misses due to temporal locality.

More precisely, under the assumption that there is no empty column, since there has to be at least one cache-miss for each column c_j. The column c_j brings $\lambda(c_j) - 1$ extra cache-misses due to temporal locality in the worst case.

Corollary 1 *Given a K-way SB form of matrix A such that every submatrix A_k can fit into the cache. Then the number $\Phi_{additional}(A_{SB})$ of additional cache misses due to the access of $x\text{-}vector$ entries can be upperbounded as*

$$\Phi_{additional}(A_{SB}) \leq \sum_{c_j} (\lambda(c_j) - 1) \tag{4.7}$$

As also discussed in [2], this A-to-A_{SB} transformation problem can be formulated as an HP problem using the column-net model of matrix A with a part size constraint of cache size and partitioning objective of minimizing cutsize according to the connectivity-1 metric definition given in 2.2.

4.2 Hierarchical Clustering

For row/column reordering of sparse matrices, an hierarchical bottom-up approach is also proposed. This idea is inspired from GPART Algorithm proposed by Han and Tseng [17]. A nice property of this approach is being cache-oblivious. Different from

Han and Tseng, in this approach, hypergraph is used instead of graph. The given sparse matrix is represented as a hypergraph by utilizing the column-net hypergraph model. Thus, the rows are represented by vertices and the columns are represented by nets. The reordering algorithm works in a bottom-up fashion and performs clustering phases as far as it could be. On each clustering pass, the vertices are clustered according to the "heavy net connectivity" metric which is commonly used on coarsening phase of hypergraph partitioning tools. Each cluster is then behaved as a single vertex in the next pass and this forms coarsened hypergraph constituting a hierarchical structure. The coarsening process continues until there exists one vertex left or all vertices are disconnected in the coarsened hypergraph.

The proposed hierarchical clustering algorithm is presented in Algorithm 5. The rows of the sparse matrix is reordered respecting the hierarchy of clustering. That is, the rows are reordered in such a way that the rows corresponding to vertices of a cluster are grouped together. This refers to the idea of clustering the rows with similar sparsity patterns and consequently improves the exploitation of temporal locality. On each clustering pass, first the vertices are sorted in decreasing order of net degrees. Then all vertices are processed respecting to this order. That is, the vertex with more nets is processed before. The intention of "processing the vertex" is either assigning the vertex to a cluster or form a new cluster with another vertex. If the vertex, to be processed, is already clustered, then it is not further assigned to any cluster and the algorithm passes to the next vertex. But if the vertex is not yet clustered, the most attractive cluster is selected. The attractiveness of a cluster is evaluated by heavy net connectivity metric. In this metric, the cluster with largest number of shared nets is most attractive. Note that the other unclustered vertices are considered as one-vertex clusters when we evaluate the attractiveness. Therefore, the processing vertex can either select a cluster or an unclustered vertex as most attractive. If it selects a cluster, the vertex simply joins that cluster. However, if the processing vertex selects an unclustered vertex, than these two vertices form a new cluster. The above-mentioned procedure only reorders the rows of the matrix. The columns are reordered as in CPACK approach in which columns are moved into adjacent locations in the order they are first encountered by a row [13]. Consequently, the overall process presents a simple yet effective algorithm where temporal locality is exploited by reordering rows with similar sparsity patterns nearby by utilizing the hierarchy of clustering and spatial locality is exploited via a post processing.

4.3 Compression Preprocessing for Spatial Locality

The column-net model exploits temporal locality in the first place. Reordering columns utilizing the information obtained from vertex partition exploits spatial locality. Process of reordering of columns is not necessarily to be done if spatial locality is exploited via any method. Such a method is compression of columns. This is a preprocessing step in which columns are grouped to form cache-line-sized clusters so that only temporal locality will be considered in further steps. The requirement for taking care of spatial locality disappears in further steps. This approach can be used as a preprocessing step of any row reordering method.

The columns with similar sparsity pattern are clustered to form cache-line-sized clusters. If a cluster cannot reach size of cache line, then they are left single. Clustering process is performed via successive matchings of columns. All columns are singleton clusters at the beginning. Clusters are processed in random order. Each cluster selects the most attractive unprocessed cluster. The cluster that shares maximum number of rows with the selector cluster is most attractive. After every cluster selects another cluster, one level ends and another level starts so number of levels is $log_2 cachelinesize$. Each final cluster corresponds to a new column. This matrix can be further processed for temporal locality. After this process, it is decompressed and passed to matrix-vector multiply operation.

Consequently, this preprocessing approach makes any reordering method cache-line-size aware.

Algorithm 4 Hypergraph Based Bottom-up Reordering HPART

Require: Hypergraph $\mathcal{H} = (\mathcal{U}, \mathcal{N})$, Tree level t
1: **if** $|\mathcal{U}| = 1$ or nodes of $\mathcal{H}$ are disconnected **then**
2: **return** t
3: **end if**
4: $\mathcal{H}_{coarsen} \leftarrow HCLUSTER(\mathcal{H})$
5: $t_{upper}.lower \leftarrow t$
6: **return** HPART($\mathcal{H}_{coarsen}$, t)

Algorithm 5 Hypergraph Based Clustering HCLUSTER

Require: Hypergraph $\mathcal{H} = (\mathcal{U}, \mathcal{N})$
1: $\mathcal{C} \leftarrow \emptyset$
2: **for each** node $u \in \mathcal{U}$ **do**
3: selected[u] $\leftarrow FALSE$
4: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\{u\}\}$
5: **end for**
6: **for each** node $u \in \mathcal{U}$ in decreasing order of number of nets **do**
7: **if** selected[u] $= FALSE$ **then**
8: $\mathcal{C} \leftarrow \mathcal{C} - \{\{u\}\}$
9: $max \leftarrow 0$
10: **for each** cluster $c \in \mathcal{C}$ **do**
11: $\mathcal{S} \leftarrow \emptyset$
12: **for each** node $v \in c$ **do**
13: $\mathcal{S} \leftarrow \mathcal{S} \cup Nets[v]$
14: **end for**
15: $\mathcal{S} \leftarrow \mathcal{S} \cap Nets[u]$
16: **if** $max < |\mathcal{S}|$ **then**
17: $max \leftarrow |\mathcal{S}|$
18: $maxc \leftarrow c$
19: **end if**
20: **end for**
21: **if** $max > 0$ **then**
22: $\mathcal{C} \leftarrow \mathcal{C} - \{maxc\}$
23: **if** $|maxc| = 1$ **then**
24: selected[v] $\leftarrow TRUE$, where $maxc = \{v\}$
25: **end if**
26: $c \leftarrow maxc \cup \{u\}$
27: $\mathcal{C} \leftarrow \mathcal{C} \cup c$
28: **end if**
29: **end if**
30: **end for**
31: **return** $\mathcal{H}_{coarsen}$ due to cluster $\mathcal{C}$

Chapter 5

Multiple Submatrix-Vector Multiplies Framework

In this framework, we assume that the nonzeros of matrix A are partitioned arbitrarily among K submatrices such that each submatrix A^k contains a mutually disjoint subset of nonzeros. Then matrix A can be written as

$$A = A^1 + A^2 + \cdots + A^k \tag{5.1}$$

Note that this partitioning is not necessarily row disjoint or column disjoint. That is, the nonzeros of a given column of matrix A might be shared by multiple submatrices. Similarly, the nonzeros of a given row of matrix A might be shared by multiple submatrices. In this framework, $y \leftarrow Ax$ can be computed as

$$\begin{aligned} &\textbf{for } k \leftarrow 1 \textbf{ to } K \\ &\quad y \leftarrow y + A^k x \end{aligned} \tag{5.2}$$

The partitioning of matrix A into submatrices A^k should be done in such a way that the temporal and spatial locality of individual submatrix-vector multiplications are exploited in order to minimize cache misses during an individual submatrix-vector multiplication. This goal is similar as Single Matrix-Vector Multiply framework discussed in Chapter 4. On the contrary, this framework requires partitioning of the matrix A into submatrices whereas previous framework uses the method of reordering rows and columns. We discuss pros and cons of this framework according to the conventional

framework $y \leftarrow Ax$ in Section 5.1. In Section 5.2, we also show that partitioning the matrix A into submatrices can be performed by 2D-partitioning of fine-grain hypergraph model. The order of individual submatrix-vector multiply operations is also important to exploit temporal locality. We state this ordering problem as an instance of traveling salesman problem in Section 5.3.

5.1 Pros and Cons compared to Conventional Framework

Since a global row and column ordering is assumed in Equation 5.3, submatrices are likely to contain empty rows. Hence, each individual sparse submatrix-vector multiply operation $y \rightarrow y + A^k x$ is performed using the ICSR scheme. As seen Algorithm 6, individual submatrix-vector multiply results are accumulated in the output vector y on the fly in order to avoid additional write operations.

Algorithm 6 Multiple Sparse Submatrix-Vector Multiplications using ICSR scheme

Require: $nonzero^k$, $column\text{-}diff^k$ and $row\text{-}jump^k$ arrays of a m^k by n^k sparse submatrix A^k where $k = 1, 2 \dots K$, K is total number of submatrices, number of nonzeros nnz^k in matrix A^k,
a dense input vector x

```
1: for k ← 1 to K do
2:     i ← row-jump^k[0]
3:     j ← column-diff^k[0]
4:     t ← 0
5:     r ← 1
6:     tmp ← 0
7:     while t < nnz^k do
8:         tmp ← tmp + nonzero^k[t] * x[j]
9:         t ← t + 1
10:        j ← j + column-diff^k[t]
11:        if j ≥ n then
12:            y[i] ← tmp
13:            tmp ← 0
14:            j ← j − n^k
15:            i ← i + row-jump^k[r]
16:            r ← r + 1
17:        end if
18:    end while
19: end for
20: return y
```

Note that the conventional single matrix-vector multiply framework can be considered as a special case in which submatrices are also restricted to be row disjoint. Thus, this framework brings an additional flexibility for exploiting the temporal and spatial locality. Clustering A-matrix rows/subrows with similar sparsity pattern into the same submatrices increases the possibility of exploiting temporal locality in accessing $x\text{-}vector$ entries. Clustering A-matrix columns/subcolumns with similar sparsity pattern into the same submatrices increases the possibility of exploiting spatial locality in accessing $x\text{-}vector$ entries as well as temporal locality in accessing $y\text{-}vector$ entries.

However, this additional flexibility comes at a cost of disturbing the following locality compared to conventional approach. There is some disturbance in the spatial locality in accessing the nonzeros of the A-matrix due to the division of three arrays associated with nonzeros into K parts. However, this disturbance in spatial locality is negligible since elements of each of the three arrays are stored and accessed consecutively during each submatrix vector multiply operation. That is, at most $3(K-1)$ extra cache misses occur compared to the conventional $y \leftarrow Ax$ scheme due to the disturbance of spatial locality in accessing the nonzeros of A-matrix . Furthermore, multiple read/writes of the submatrix-vector multiply results might bring some disadvantages compared to conventional single matrix-vector multiply. These multiple read/writes disturb the spatial locality of accessing $y\text{-}vector$ entries as well as introducing a temporal locality exploitation problem in $y\text{-}vector$ entries.

Our problem here, can be defined as the matrix-to-submatrix partitioning problem. As a solution, the following theorem gives the guidelines for a "good" matrix-to-submatrix partitioning:

Theorem 2 *Consider a partition $\Pi(A)$ of matrix A into K nonzero-disjoint submatrices $A^1, A^2, \ldots, A^K$. Let $\lambda(r_i)$ denote the number of submatrices that contain at least one nonzero of row r_i of matrix A, i.e., $\lambda(r_i) = |\{A^k : r_i \in A^k\}|$. Similarly let $\lambda(c_j)$ denote the number of submatrices that contain at least one nonzero of column c_j of matrix A, i.e., $\lambda(c_j) = |\{A^k : c_j \in A^k\}|$.Let q denote the maximum number of caches that a submatrix can fit into. Then the number $\Phi(\Pi(A))$ of cache misses due to the access of $x\text{-}vector$ and $y\text{-}vector$ entries can be upperbounded as*

$$\Phi(\Pi(A)) \leq \sum_{r_i} \lambda(r_i) + q \sum_{c_j} \lambda(c_j) \tag{5.3}$$

if cache is assumed to be fully-associative.

Proof Consider the case that the line size is equal to the x/y-vector entry size. For each submatrix A^k, each $y\text{-}vector$ result of A^k is written only once to the memory. For the sake of simplicity, we refer $\Phi(\Pi(A))$ as Φ. Let Φ_x and Φ_y respectively denote the number of cache misses due to the access of $x\text{-}vector$ and $y\text{-}vector$ entries for $\Pi(A)$. Then,

$$\Phi = \Phi_x + \Phi_y \tag{5.4}$$

The number of cache misses due to the access of y_i is at most $\lambda(r_i)$ which happens when no cache-reuse occurs in accessing to y_i, that is,

$$\Phi_y \leq \sum_{r_i} \lambda(r_i). \tag{5.5}$$

Let q_k denote the minimum number of caches that submatrix A^k can fit into. Since full-associativity is assumed, for each submatrix A^k, each $x\text{-}vector$ entry of A^k is accessed at most q_k times. Therefore, the number of cache misses due to the access of x_j is at most q_k for each submatrix A^k that x_j is needed to be accessed. Then,

$$\begin{aligned}
\Phi_x &\leq \sum_{c_j} \sum_{k:c_j \in A^k} q_k && (5.6)\\
&\leq \sum_{c_j} \sum_{k:c_j \in A^k} q && (5.7)\\
&= q \sum_{c_j} \sum_{k:c_j \in A^k} 1 && (5.8)\\
&= q \sum_{c_j} \lambda(c_j) && (5.9)
\end{aligned}$$

Equation 5.4, Equation 5.5 and Equation 5.9 together yield to Equation 5.3. Extending the line size can only increase the cache-reuse and accordingly decrease the cache-miss. Therefore, Equation 5.3 still holds for larger line sizes.

Corollary 2 *When all submatrices fit into the cache then the number $\Phi(\Pi(A))$ of cache misses due to the access of $x\text{-}vector$ and $y\text{-}vector$ entries can be upperbounded as*

$$\Phi(\Pi(A)) \leq \sum_{r_i} \lambda(r_i) + \sum_{c_j} \lambda(c_j) \tag{5.10}$$

These theorems give exact upper bounds for when temporal reuse is exploited at the utmost degree via fully-associativity.

5.2 2D Decomposition of Sparse Matrices

The aim is to partition the given sparse matrix A into K nonzero-disjoint submatrices. Corollary 2 leads us to a cache-size aware top-down matrix-to-submatrix partitioning which minimizes the sum $\sum_{r_i} \lambda(r_i) + \sum_{c_j} \lambda(c_j)$ of λ values of rows and columns such that the storage of each submatrix-vector multiply fits into the cache. Here, minimizing objective relates to minimizing the cache misses due to temporal locality.

More precisely, under the assumption that there is no empty column, since there has to be at least one cache-miss for each row r_i and each column c_j. Thus the row r_i and the column c_j, respectively, bring $\lambda(r_i) - 1$ and $\lambda(c_j) - 1$ extra cache-misses due to temporal locality in the worst case.

Corollary 3 *Given a K-way matrix-to-submatrix partition $\Pi(A)$ of matrix A such that every submatrix A^k can fit into the cache. Then the number $\Phi_{additional}(\Pi(A))$ of additional cache misses due to the access of x-$vector$ and y-$vector$ entries can be upperbounded as*

$$\Phi_{additional}(\Pi(A)) \leq \sum_{r_i} (\lambda(r_i) - 1) + \sum_{c_j} (\lambda(c_j) - 1) \tag{5.11}$$

The matrix-to-submatrix partition problem can be formulated as an HP problem using the row-column-net model of matrix A with a part size constraint of cache size and partitioning objective of minimizing cutsize according to the connectivity-1 metric definition given in Equation 2.2.

5.3 Ordering Submatrix-Vector Multiplies

The partitioning of matrix A into submatrices A^k should be done in such a way that the temporal and spatial locality of individual submatrix-vector multiplications are exploited in order to minimize cache misses during an individual submatrix-vector multiplication. When all the multiplications are considered, data reuse between two consecutive submatrix-vector multiplications must be maximized to exploit temporal locality. We give an exact lower bound for the cache misses due to the access of x-$vector$ and y-$vector$ entries for a given order of submatrices.

Theorem 3 *Consider a partition $\hat{\Pi}(A)$ of matrix A into K nonzero-disjoint submatrices $A^1, A^2, \ldots, A^K$ with a given ordering of the submatrices. Let $\gamma(r_i)$ and $\gamma(c_j)$, denote the number of submatrix-subchains in which all submatrices contain at least one nonzero of row r_i and column c_j, respectively. Let w denote the line size in terms of a unit x/y-vector entry. If no submatrix A^k can fit into one cache, then the number $\Phi(\hat{\Pi}(A))$ of cache misses due to the access of x-vector and y-vector entries can be lowerbounded as*

$$\Phi(\hat{\Pi}(A)) \geq \frac{\sum_{r_i} \gamma(r_i) + \sum_{c_j} \gamma(c_j)}{w} \tag{5.12}$$

Proof We will give the proof only for the columns, since a similar proof applies for the rows; then total number of cache misses can be written as sum of cache misses due to access of y-vector entries and x-vector entries and can be formulated as

$$\Phi(\hat{\Pi}(A)) = \Phi_r(\hat{\Pi}(A)) + \Phi_c(\hat{\Pi}(A)) \tag{5.13}$$

Consider a column c_j of matrix A. Then there exists $\gamma(c_j)$ submatrix-subchains for column c_j. Since no submatrix A^k can fit into one cache, it is guaranteed that there will be no cache-reuse of column c_j between two different submatrix-subchains including c_j. Therefore, at least $\gamma(c_j)$ cache misses will occur for each column c_j which yields that the number $\Phi_c(\hat{\Pi}(A))$ of cache misses due to the access of x-vector entries is greater than or equal to $\sum_{r_i} \sum_{c_j} \gamma(c_j)$ in the case of unit cache-line-size, i.e., $w = 1$. Since the number of cache-misses can maximally decrease w-fold, the number $\Phi_c(\hat{\Pi}(A))$ of cache misses due to the access of x-vector entries is greater than or equal to $\frac{\sum_{c_j} \gamma(c_j)}{w}$.

Theorem 4 *Consider the TSP Instance ($\mathcal{G} = (\mathcal{V}, \mathcal{E}), w$), where vertex set $\mathcal{V}$ denotes the K submatrices. There exists an edge e_{ij} in $\mathcal{E}$ if and only if there exists at least one row or column shared between submatrices A^i and A^j. The weight of edge w_{ij} denotes the sum of the number of shared rows and the number of shared columns between submatrices A^i and A^j. Then, finding an order of $\mathcal{V}$ which maximizes the path weight corresponds to finding an order of submatrices which minimizes $\sum_{r_i} \gamma(r_i) + \sum_{c_j} \gamma(c_j)$.*

Proof

$$\begin{aligned}
\sum_{r_i}\gamma(r_i)+\sum_{c_j}\gamma(c_j) &= \sum_{r_i}(|A_{i_1}\cap\{r_i\}|+\sum_{k=2}^{K}|(A_{i_k}-A_{i_{k-1}})\cap\{r_i\}|) \\
&+ \sum_{c_j}(|A_{i_1}\cap\{c_j\}|+\sum_{k=2}^{K}|(A_{i_k}-A_{i_{k-1}})\cap\{c_j\}|) \\
&= |A_{i_1}|+\sum_{k=2}^{K}|(A_{i_k}-A_{i_{k-1}})| \\
&= |A_{i_1}|+\sum_{k=2}^{K}|(A_{i_k}-(A_{i_k}\cap A_{i_{k-1}}))| \\
&= |A_{i_1}|+\sum_{k=2}^{K}(|A_{i_k}|-|A_{i_k}\cap A_{i_{k-1}}|) \\
&= \sum_{k=1}^{K}|A_{i_k}|-\sum_{k=2}^{K}|A_{i_k}\cap A_{i_{k-1}}| \\
&= \sum_{k=1}^{K}|A_{i_k}|-\sum_{k=2}^{K}w_{i_k,i_{k-1}}
\end{aligned}$$

In the above formulation, A_{i_k} is used to denote the kth submatrix in the order of submatrices and A_{i_k} is also used to denote the set of rows and columns that belong to the submatrix A_{i_k}. The maximum value of $\sum_{k=2}^{K}w_{i_k,i_{k-1}}$ will yield the minimum value of $\sum_{r_i}\gamma(r_i)+\sum_{c_j}\gamma(c_j)$. Then, finding an order of $\mathcal{V}$ which maximizes the path weight $\sum_{k=2}^{K}w_{i_k,i_{k-1}}$ corresponds to finding an order of submatrices which minimizes $\sum_{r_i}\gamma(r_i)+\sum_{c_j}\gamma(c_j)$.

According to Theorem 4, the lower bound $\sum_{r_i}\gamma(r_i)+\sum_{c_j}\gamma(c_j)$ corresponds to the objective function of the TSP instance constructed in the theorem.

Chapter 6

Experimental Results

Throughout the previous chapters, we investigate ways of exploiting data locality by reordering/partitioning a sparse matrix A. In this chapter, we show the improvements gained by the proposed models and frameworks. A cache simulator is used to show these improvements. The existing state-of-the-art models such as row-net model [40] and BFS-based algorithm [1] are also tested.

6.1 Experimental Setup

The two contributed frameworks and underlying models are based on decreasing cache misses incurred by $x\text{-}vector$ and $y\text{-}vector$ entries so using a cache simulator will make the improvement obtained by our contributions more clear. One must pay more attention to sum of cache misses caused by $x\text{-}vector$ and $y\text{-}vector$ entries to see the proof of our proposed concepts.

All simulation results are normalized according to the number of cache misses for original, unprocessed matrices. The cache miss ratio is 1.00 if the number of cache misses does not change. If number of cache misses is decreased when a method applied, the ratio is smaller than 1.00. Similarly, if number of cache misses is increased when a method applied, the ratio is greater than 1.00. The used normalization equation

is as follows:

$$ratio = \frac{miss_{reordered}}{miss_{original}} \tag{6.1}$$

The data type used in storage of matrices is double precision floating-point number which has size of 8 bytes on the test platform. Only the index arrays use integers which have size of 4 bytes.

6.1.1 Platform

The experiments of using original matrices and reordered matrices in the single matrix-vector multiply framework are performed on a cache simulator also used in [40].The experiments related with the multiple matrix-vector multiplies framework are also performed on the cache simulator.

Simulation results of experiments are given for when cache line size is size of 8 doubles, cache size is 32KB and set-associativity is 8. This configuration is taken from [40]. Some of results are given for when cache line size of 1 double and number of cache lines is one eighth of original cache line number so that cache size is still 32KB. The aim is to show only the effect of temporal locality clearly because spatial locality for x and y vectors cannot exist when only 1 double is retrieved when a cache miss occurs.

6.1.2 Data Sets

The proposed frameworks are tested and validated on numerous matrices collected from The University of Florida Sparse Matrix Collection [36]. General properties of these matrices can be seen Table 6.1.

The columns can be explained as follows:

1. $\#Rows$: number of rows
2. $\#Cols$: number of columns
3. $\#Nonzeros$: number of nonzeros

Table 6.1: Properties of test matrices.

name	$\#Rows$	$\#Cols$	$\#Nonzeros$	$Symmetricity$	d_{row}	d_{col}
Square symmetric matrices						
bloweya	30004	30004	150009	yes	5	5
bloweybl	30003	30003	109999	yes	4	4
dixmaanl	60000	60000	299998	yes	5	5
dtoc	24993	24993	69972	yes	3	3
F2	71505	71505	5294285	yes	74	74
msc10848	10848	10848	1229776	yes	113	113
msc23052	23052	23052	1142686	yes	50	50
Na5	5832	5832	305630	yes	52	52
ncvxqp9	16554	16554	54040	yes	3	3
ship_001	34920	34920	3896496	yes	112	112
smt	25710	25710	3749582	yes	146	146
Trefethen_20000	20000	20000	554466	yes	28	28
TSOPF_FS_b300	29214	29214	4400122	yes	151	151
tuma1	22967	22967	87760	yes	4	4
tuma2	12992	12992	49365	yes	4	4
Square unsymmetric matrices						
mixtank_new	29957	29957	1990919	99%	66	66
powersim	15838	15838	64424	53%	4	4
memplus	17758	17758	99147	50%	6	6
sme3Db	29067	29067	2081063	44%	72	72
sme3Dc	42930	42930	3148656	44%	73	73
circuit_4	80209	80209	307604	36%	4	4
circuit_3	12127	12127	48137	30%	4	4
poli_large	15575	15575	33033	0.05%	2	2
fd18	16428	16428	63406	0%	4	4
ns3Da	20414	20414	1679599	0%	82	82
poisson3Da	13514	13514	352762	0%	26	26
Zd_Jac3	22835	22835	1915726	0%	84	84
Zhao1	33861	33861	166453	0%	5	5
Zhao2	33861	33861	166453	0%	5	5
Rectangular matrices						
baxter	27441	30733	111576	no	4	4
ch7-8-b2	11760	1176	35280	no	3	30
co9	10789	22924	109651	no	10	5
cq9	9278	21534	96653	no	10	4
ex3sta1	17443	17516	68779	no	4	4
fome11	12142	24460	71264	no	6	3
fome12	24284	48920	142528	no	6	3
ge	10099	16369	44825	no	4	3
Kemelmacher	28452	9693	100875	no	4	10
lp_dfl001	6071	12230	35632	no	6	3
lp_pds_02	2953	7716	16571	no	6	2
lp_stocfor3	16675	23541	72721	no	4	3
psse0	26722	11028	102432	no	4	9
psse1	14318	11028	57376	no	4	5
psse2	28634	11028	115262	no	4	10
shar_te2-b1	17160	286	34320	no	2	120

4. $Symmetricity$: For square matrices, it is percantage of the number of off-diangonal nonzeros that have symmetric entries to total number of off-diangonal nonzeros

5. d_{row} : number of nonzeros per row

6. d_{col} : number of nonzeros per column

6.2 Experiments with Single Matrix-Vector Multiply Framework

The two column-net and row-net hypergraph models for 1D partitioning of sparse matrices are evaluated. Using row-net hypergraph model is proposed in [40] and using column-net model is our proposal. First of all, performances of sparse matrix-vector multiply operation using matrices partitioned according to these two models are compared. In row-net model, columns with similar sparsity pattern are gathered together by reordering columns so spatial locality of $x\text{-}vector$ entries is exploited. Also partitions on columns induce partitions on rows. This induced row partition is used to reorder rows to increase temporal locality of $x\text{-}vector$ entries. Additionally, the cut rows are placed between two partitions as proposed in [40] instead of putting these rows at the end. If these cut rows are put at the end, $x\text{-}vector$ entries retrieved to cache to be used by previous parts cannot be reused in cut rows.

On the other hand, column-net model is directly related with temporal locality. Rows with similar sparsity pattern are gathered together by reordering rows so temporal locality of$x\text{-}vector$ entries is exploited. Also the induced column partition is used to reorder columns to increase spatial locality of $x\text{-}vector$ entries. The columns in the cut are placed at the end. Putting these columns between two parts may decrease cache missed incurred by $x\text{-}vector$ entries but the gain so small to be significant.

These two mentioned methods exploit both spatial and temporal locality but first method gives more importance to spatial locality. In contrast, the second method gives more importance to temporal locality. The results show that temporal locality is more important and proves correctness of Theorem 1.

Table 6.2: Normalized geometric and arithmetic means of simulation results for matrices partitioned into 32K-sized parts using row-net and column-net models. Original matrices are partitioned and their transposes, too. Cache line size is 8 times size of double, 64Bytes.

	Existing Method								*Proposed Method*							
	$A-$*Row-net* [40]				A^T-*Row-net* [40]				$A-$*Column-net*				A^T-*Column-net*			
	x	y	$x+y$	tot	x	y	$x+y$	tot	x	y	$x+y$	tot	x	y	$x+y$	tot
Geometric means																
Square Symmetric	0.69	1.00	0.75	0.93	0.69	1.00	0.75	0.93	0.52	1.00	0.61	0.91	0.52	1.00	0.61	0.91
Square Unsymmetric	0.49	1.00	0.51	0.76	0.48	1.00	0.50	0.76	0.27	1.00	0.30	0.70	0.28	1.00	0.31	0.70
Rectangular	0.39	1.00	0.47	0.72	0.55	1.00	0.66	0.87	0.28	1.00	0.37	0.67	0.39	1.00	0.53	0.82
Overall	0.50	1.00	0.56	0.80	0.57	1.00	0.63	0.85	0.34	1.00	0.41	0.75	0.39	1.00	0.47	0.81
Arithmetic means																
Square Symmetric	0.76	1.00	0.81	0.93	0.76	1.00	0.81	0.93	0.57	1.00	0.65	0.91	0.57	1.00	0.65	0.91
Square Unsymmetric	0.75	1.00	0.75	0.82	0.71	1.00	0.72	0.82	0.46	1.00	0.52	0.76	0.48	1.00	0.53	0.76
Rectangular	0.51	1.00	0.55	0.75	0.78	1.00	0.85	0.91	0.38	1.00	0.47	0.71	0.64	1.00	0.73	0.86
Overall	0.67	1.00	0.70	0.83	0.76	1.00	0.80	0.89	0.47	1.00	0.54	0.79	0.57	1.00	0.64	0.85

When we consider the structure of a matrix, its structure may favour either row-net model or column-net model. If a sparse matrix A gives greater cutsize in row-net model and its transpose A^T give less cutsize in column-net model, then this matrix A is said to favour row-net model. Taking best result of partitioning hypergraph induced by A and A^T will get rid of this bias. This bias does not exist in square symmetric matrices. It exists in square unsymmetric and rectangular matrices and its effect is more clear especially in rectangular matrices as seen in Table 6.2. In Table 6.3, the best of A and A^T results is selected for each matrix. The colum-net model also outperforms the row-net model in this unbiased condition. Results for each matrix can be seen in Table A.1.

A cache-oblivious method is using the BFS-like Algorithm-3. Here we denote this algorithm as BFS. The columns of a sparse matrix A are reordered while reordering rows so spatial locality may be exploited for $x\text{-}vector$ entries. The difference between cache miss ratios of row-net model and BFS is considerably small as seen in Table 6.4. BFS is simple and it gives effective row/column order when spatial locality is feasible. If cache line size is equal to size of one double, BFS loses its effectiveness in its resulting order. However, Hiearchical algorithm does not lose its effectivenetss as BFS when cache line size is equal to size of one double, because it considers temporal locality beside spatial locality. Selecting initial row in BFS does not affect the quality of ordering a lot, selecting a row with minimum or maximum number of nonzeros performs slightly better than using the row order of original matrix.

Table 6.3: Normalized geometric and arithmetic means of simulation results for matrices partitioned into 32K-sized parts using row-net and column-net models. Best result of either original matrix or its transpose is selected. Cache line size is 8 times size of double, 64Bytes.

	Existing Method				*Proposed Method*			
	Row-net [40]				*Column-net*			
	x	y	$x+y$	tot	x	y	$x+y$	tot
Geometric means								
Square Symmetric	0.69	1.00	0.75	0.93	0.52	1.00	0.61	0.91
Square Unsymmetric	0.47	1.00	0.49	0.76	0.26	1.00	0.30	0.69
Rectangular	0.33	1.00	0.42	0.71	0.23	1.00	0.33	0.67
Overall	0.47	1.00	0.53	0.79	0.32	1.00	0.39	0.75
Arithmetic means								
Square Symmetric	0.76	1.00	0.80	0.93	0.57	1.00	0.64	0.91
Square Unsymmetric	0.70	1.00	0.71	0.82	0.45	1.00	0.51	0.75
Rectangular	0.42	1.00	0.49	0.73	0.32	1.00	0.41	0.70
Overall	0.62	1.00	0.66	0.83	0.44	1.00	0.52	0.79

Table 6.4: Normalized geometric and arithmetic means of simulation results for matrices partitioned into 32K-sized parts using row-net and column-net models; and matrices reordered using BFS and Hierarchical algorithms

	Existing Methods								*Proposed Methods*							
	Row-net [40]				*BFS* [1]				*Column-net*				*Hierarchical*			
	x	y	$x+y$	tot	x	y	$x+y$	tot	x	y	$x+y$	tot	x	y	$x+y$	tot
Cache line size is 8 times size of double, 64Bytes.																
ARITHMETIC	0.67	1.00	0.70	0.83	0.65	1.00	0.67	0.83	0.47	1.00	0.54	0.79	0.69	1.00	0.69	0.83
GEOMETRIC	0.51	1.00	0.56	0.80	0.45	1.00	0.50	0.79	0.34	1.00	0.41	0.75	0.45	1.00	0.50	0.79
Cache line size equals size of one double, 8Bytes.																
ARITHMETIC	0.73	1.00	0.79	0.95	0.82	1.00	0.84	0.96	0.60	1.00	0.67	0.92	0.69	1.00	0.75	0.94
GEOMETRIC	0.67	1.00	0.75	0.94	0.71	1.00	0.75	0.96	0.50	1.00	0.59	0.92	0.58	1.00	0.66	0.94

The compression method can be used alone or as a preprocessing step. When it is used as preprocessing step, time consumed in HP is decreased enough as seen in Table 6.5. However this method does not give good results for all matrices, even situation worsens.

Table 6.5: Normalized simulation results for some matrices. Results for only compression method applied are in $Comp$ column. Results for matrices are partitioned into 32K-sized parts using column-net model without column reordering are in Row column; with column ordering in Col. Results for column-net model without column reordering but with compression are in $ColC$ column. Time elapsed for reordering and compression are measured in milliseconds. Timing results for reordering using column-net model in $t_{\{Col\}}$ column. Compression, partitioning and total times for reordering using column-net model without column reordering but with compression are given separately in $t_{\{ColC\}}$ column.

	$Comp$	Row	Col	$ColC$	$t_{\{Col\}}$	$t_{\{ColC\}}$		
name						t_{part}	t_{comp}	t_{tot}
msc10848	1.14	0.55	0.48	0.67	8710	1253	402	1655
ship_001	1.02	0.71	0.67	0.88	35243	5433	1657	7089
smt	0.87	0.61	0.58	0.64	29880	5065	1401	6466
F2	0.79	0.38	0.31	0.56	47010	8358	1829	10186
sme3Db	0.46	0.1	0.03	0.03	17875	2965	1177	4142
sme3Dc	0.45	0.1	0.03	0.03	31430	5555	1845	7400
ns3Da	0.42	0.1	0.04	0.03	14738	2683	1009	3691
dixmaanl	1.02	0.35	0.34	0.38	1763	725	102	827
fd18	1.97	0.75	0.44	0.8	710	333	31	364
poli_large	1.1	0.82	0.64	0.88	355	170	14	184
Zhao2	3.02	0.99	0.46	0.92	2050	1025	83	1108
Zhao1	3.03	0.99	0.46	1.04	2030	1020	83	1103

As a result, applying the proposed reordering scheme using column-net model decreases number of cache misses considerably according to the unordered case. However, the improvement is *highly* dependent on the structure of the matrix. If preprocessing time is also important BFS or Hierarchical algorithms can be used because hypergraph partitioning takes longer time.

Table 6.6: Normalized geometric and arithmetic means of simulation results for matrices partitioned into 32K-sized parts using fine-grain model. $NOTSP$ column contains results when TSP ordering not used.Cache line size is 8 times size of double, 64Bytes.

	$NOTSP$				TSP			
	x	y	$x+y$	tot	x	y	$x+y$	tot
ARITHMETIC	0.51	7.53	0.80	0.85	0.45	5.97	0.73	0.82
GEOMETRIC	0.39	3.20	0.62	0.81	0.30	2.72	0.51	0.78

6.3 Experiments with Multiple Submatrix-Vector Multiplies Framework

The fine-grain hypergraph model is used in 2D decomposition sparse matrices. This decomposition is used to partition the matrix A int submatrices A_k. Rows of each submatrix have similar sparsity pattern and similarly columns of each submatrix have similar sparsity pattern. Dimensions of these submatrices are as the original matrix's so numbers of empty rows and columns are considerable great. Great number of empty columns disturbs spatial locality of $x\text{-}vector$ entries. Locating internal column nets of each submatrix consecutively decreases bad effects of this disturbance. Large number of empty rows causes performance loss in the CSR storage scheme but not in ICSR. Number of cache misses measured using CSR is proportional with number of whole rows but in ICSR it is proportional with number of non-empty rows. When row ordering of each submatrix is considered, the internal row nets are ordered consecutively to increase the disturbed spatial locality of $y\text{-}vector$ entries.

The order of submatrix-vector multiplies is also important. The cache miss difference between random ordering and order found by TSP can be seen in Table 6.6.

As a result, it is shown that the multiple submatrix-vector multiplies framework can preferred to conventional scheme and ordering submatrices increases temporal reuse between consecutive multiply operations.

6.4 Comparison of Frameworks

Column-net model in single matrix-vector multiply framework outperform others in both cases, spatial locality is available or not. Unfortunately the second framework

Table 6.7: Normalized geometric and arithmetic means of simulation results for matrices partitioned into 32K-sized parts using column-net model and fine-grain model with TSP ordering.

	Column-net				*Fine-grain*			
	x	y	$x+y$	tot	x	y	$x+y$	tot
Cache line size is 8 times size of double, 64Bytes.								
ARITHMETIC	0.47	1.00	0.54	0.79	0.45	5.97	0.73	0.82
GEOMETRIC	0.34	1.00	0.41	0.75	0.30	2.72	0.51	0.78
Cache line size equals size of one double, 8Bytes.								
ARITHMETIC	0.60	1.00	0.67	0.92	0.51	2.09	0.67	0.93
GEOMETRIC	0.50	1.00	0.59	0.92	0.42	1.57	0.59	0.93

cannot achieve lower cache miss ratios as shown in Table 6.7. When cache line size is size of one double, performances of these two fraweworks are similar. This shows that the disturbed spatial localities of x and y vectors cannot be re-gained, there still exist unsolved issues in exploting spatial locality.

Chapter 7

Conclusion

In this chapter, we will conclude the results of our work, and consider some opinions about the future work of this thesis.

7.1 Conclusions

Two hypergraph partitioning models were proposed for reordering a sparse matrix to minimize cache misses caused by input/output vector during SpMxV. For each model, a framework proposed to exploit the benefits of the model. These models aim firstly to exploit temporal locality of input vector in single matrix-vector multiply framework and of both input and output vectors in multiple matrix-vector multiplies framework. After then, spatial locality is exploited for these vectors. Furthermore, column-net and fine-grain hypergraph models introduce a much more accurate representation for cache misses during SpMxV than the row-net hypergraph model when the temporal locality is considered. It is shown that exploting temporal locality has generally more importance during SpMxV. Reordering rows/columns for obtaining spatial locality comes after, and it must respect the order exploiting temporal locality.

Although 2D partitioning using fine-grain hypergraph model gives smaller cutsizes than 1D partitioning using column-net model, the multiple matrix-vector multiplies framework cannot outperform. One of the possible reasons is the problem of recruiting the spatial locality of output vector entries whose spatial locality was disturbed while

reordering rows.

7.2 Future Work

The proposed frameworks can be more cache-oblivious. Meanly, partitioning a matrix till it fits into cache is not always feasible when we consider today's HP partitioners. After some recursive bisections, a cache-oblivious algorithm such as BFS can be used to reorder rows/columns to exploit both temporal and spatial locality.

There are further research issues in fine-grain model because although 2D partitioning using fine-grain hypergraph model gives smaller cutsizes than 1D partitioning using column-net model, the multiple matrix-vector multiplies framework cannot outperform.

Appendices

Appendix A

Experimental Results in Detail

Experimental results for each matrix are given in this appendix. There are four types of columns:

1. x : misses caused by $x\text{-}vector$ entries
2. y : misses caused by $y\text{-}vector$ entries
3. $x+y$: sum of $x\text{-}vector$ and $y\text{-}vector$ misses
4. tot : total miss count

Table A.1: Simulation results for matrices partitioned into 32K-sized parts. Cache line size is 8 times size of double, 64Bytes.

	Existing Method								*Proposed Method*							
	$A-$ *Row-net* [40]				A^T- *Row-net* [40]				$A-$ *Column-net*				A^T- *Column-net*			
name	x	y	$x+y$	tot	x	y	$x+y$	tot	x	y	$x+y$	tot	x	y	$x+y$	tot
Square symmetric matrices																
bloweya	1.18	1.00	1.13	1.03	1.18	1.00	1.13	1.03	0.61	1.00	0.73	0.94	0.61	1.00	0.73	0.94
bloweybl	0.90	1.00	0.93	0.98	0.90	1.00	0.93	0.98	0.62	1.00	0.74	0.93	0.62	1.00	0.73	0.93
dixmaanl	0.35	1.00	0.51	0.87	0.35	1.00	0.51	0.87	0.34	1.00	0.51	0.87	0.34	1.00	0.51	0.87
dtoc	0.72	1.00	0.84	0.96	0.72	1.00	0.84	0.96	0.72	1.00	0.84	0.96	0.72	1.00	0.84	0.96
F2	0.54	1.00	0.55	0.93	0.53	1.00	0.55	0.93	0.28	1.00	0.31	0.90	0.28	1.00	0.31	0.90
msc10848	0.68	1.00	0.70	0.98	0.69	1.00	0.71	0.98	0.44	1.00	0.47	0.97	0.44	1.00	0.47	0.97
msc23052	0.50	1.00	0.55	0.96	0.50	1.00	0.55	0.96	0.35	1.00	0.41	0.95	0.35	1.00	0.41	0.95
Na5	1.24	1.00	1.22	1.02	1.26	1.00	1.24	1.03	1.10	1.00	1.10	1.01	1.08	1.00	1.08	1.01
ncvxqp9	0.36	1.00	0.45	0.75	0.36	1.00	0.46	0.75	0.23	1.00	0.35	0.70	0.23	1.00	0.35	0.70
ship_001	0.97	1.00	0.97	1.00	0.97	1.00	0.97	1.00	0.70	1.00	0.72	0.99	0.70	1.00	0.72	0.99
smt	0.70	1.00	0.71	0.98	0.69	1.00	0.70	0.98	0.54	1.00	0.56	0.97	0.53	1.00	0.55	0.97
Trefethen_20000	0.42	1.00	0.43	0.63	0.42	1.00	0.43	0.63	0.58	1.00	0.58	0.72	0.59	1.00	0.59	0.72
TSOPF_FS_b300	1.62	1.00	1.61	1.07	1.62	1.00	1.60	1.07	1.12	1.00	1.11	1.01	1.13	1.00	1.13	1.01
tuma1	0.66	1.00	0.76	0.93	0.66	1.00	0.76	0.93	0.49	1.00	0.64	0.89	0.49	1.00	0.64	0.89
tuma2	0.62	1.00	0.73	0.92	0.62	1.00	0.73	0.92	0.49	1.00	0.64	0.90	0.49	1.00	0.64	0.90
Square unsymmetric matrices																
mixtank_new	0.61	1.00	0.62	0.92	0.61	1.00	0.62	0.92	0.24	1.00	0.26	0.85	0.24	1.00	0.26	0.85
powersim	0.49	1.00	0.64	0.91	0.46	1.00	0.61	0.90	0.45	1.00	0.62	0.90	0.42	1.00	0.59	0.89
memplus	1.22	1.00	1.18	1.04	1.24	1.00	1.19	1.05	0.68	1.00	0.74	0.93	0.67	1.00	0.73	0.93
sme3Db	0.08	1.00	0.09	0.38	0.08	1.00	0.08	0.38	0.03	1.00	0.03	0.34	0.03	1.00	0.03	0.34
sme3Dc	0.08	1.00	0.08	0.36	0.08	1.00	0.08	0.36	0.03	1.00	0.03	0.33	0.03	1.00	0.03	0.33
circuit_4	1.67	1.00	1.50	1.16	1.53	1.00	1.39	1.12	1.07	1.00	1.05	1.02	1.08	1.00	1.06	1.02
circuit_3	1.20	1.00	1.13	1.03	1.25	1.00	1.17	1.04	0.79	1.00	0.86	0.97	1.03	1.00	1.02	1.01
poli_large	0.93	1.00	0.95	0.98	0.96	1.00	0.98	0.99	0.62	1.00	0.76	0.91	0.79	1.00	0.88	0.96
fd18	0.61	1.00	0.72	0.92	0.73	1.00	0.81	0.95	0.44	1.00	0.59	0.88	0.52	1.00	0.66	0.91
ns3Da	0.12	1.00	0.12	0.38	0.12	1.00	0.12	0.38	0.04	1.00	0.04	0.33	0.04	1.00	0.04	0.33
poisson3Da	0.13	1.00	0.14	0.44	0.13	1.00	0.14	0.44	0.07	1.00	0.08	0.40	0.07	1.00	0.08	0.40
Zd_Jac3	1.80	1.00	1.75	1.07	1.26	1.00	1.26	1.08	1.03	1.00	1.02	1.00	0.93	1.00	0.93	0.98
Zhao1	0.76	1.00	0.81	0.94	0.75	1.00	0.81	0.94	0.46	1.00	0.57	0.87	0.47	1.00	0.58	0.87
Zhao2	0.75	1.00	0.80	0.94	0.76	1.00	0.81	0.94	0.46	1.00	0.57	0.87	0.47	1.00	0.58	0.87
Rectangular matrices																
baxter	0.38	1.00	0.47	0.77	0.33	1.00	0.42	0.73	0.42	1.00	0.50	0.79	0.32	1.00	0.41	0.72
ch7-8-b2	0.43	1.00	0.75	0.95	2.36	0.99	2.29	1.32	0.80	1.00	0.91	0.98	2.59	0.99	2.52	1.38
co9	0.22	1.00	0.24	0.56	0.69	1.00	0.79	0.95	0.15	1.00	0.18	0.52	0.48	1.00	0.65	0.92
cq9	0.24	1.00	0.26	0.58	0.72	1.00	0.82	0.96	0.15	1.00	0.18	0.53	0.49	1.00	0.67	0.93
ex3sta1	1.50	1.00	1.32	1.08	1.41	1.00	1.29	1.08	0.82	1.00	0.88	0.97	1.09	1.00	1.06	1.02
fome11	0.37	1.00	0.39	0.55	0.24	1.00	0.30	0.55	0.17	1.00	0.19	0.40	0.10	1.00	0.17	0.46
fome12	0.38	1.00	0.39	0.55	0.24	1.00	0.30	0.55	0.17	1.00	0.19	0.40	0.10	1.00	0.17	0.47
ge	0.23	1.00	0.29	0.60	0.57	1.00	0.70	0.90	0.17	1.00	0.24	0.57	0.37	1.00	0.57	0.86
Kemelmacher	1.25	1.00	1.09	1.01	0.63	1.00	0.68	0.91	0.87	1.00	0.95	0.99	0.47	1.00	0.53	0.87
lp_dfl001	0.37	1.00	0.39	0.55	0.23	1.00	0.30	0.55	0.16	1.00	0.19	0.40	0.10	1.00	0.17	0.46
lp_pds_02	0.14	1.00	0.18	0.43	0.46	1.00	0.69	0.90	0.12	1.00	0.15	0.42	0.32	1.00	0.61	0.88
lp_stocfor3	0.79	1.00	0.86	0.97	0.38	1.00	0.58	0.88	0.79	1.00	0.86	0.97	0.36	1.00	0.57	0.87
psse0	0.32	1.00	0.55	0.89	0.44	1.00	0.51	0.85	0.24	1.00	0.50	0.87	0.36	1.00	0.44	0.83
psse1	0.20	1.00	0.33	0.73	0.37	1.00	0.47	0.82	0.16	1.00	0.30	0.72	0.28	1.00	0.40	0.79
psse2	0.16	1.00	0.34	0.78	0.34	1.00	0.40	0.79	0.12	1.00	0.31	0.77	0.27	1.00	0.33	0.77
shar_te2-b1	1.12	1.00	1.00	1.00	3.13	1.00	3.11	1.78	0.81	1.00	1.00	1.00	2.48	1.00	2.47	1.54
ARITHMETIC	0.67	1.00	0.70	0.83	0.76	1.00	0.80	0.89	0.47	1.00	0.54	0.79	0.57	1.00	0.64	0.85
GEOMETRIC	0.50	1.00	0.56	0.80	0.57	1.00	0.63	0.85	0.34	1.00	0.41	0.75	0.39	1.00	0.47	0.81

Appendix B

Pictures of Reordered Matrices

Pictures in this appendix show the resulting matrices after partitioning the original matrix *psse1*. Each diagonal block in the pictures of partitioned matrices represents a part in the partitioned hypergraph $\mathcal{H}^B$ corresponding to partitioned sparse matrix A^B. Here B denotes total number of recursive bisections. And $\mathcal{H}^B$ contains $K = 2^B$ number of parts. The off-diagonal borders consists of column nets that are cut between two distinct parts during recursive bisection in column-net model and the maximum number of such border blocks is B.

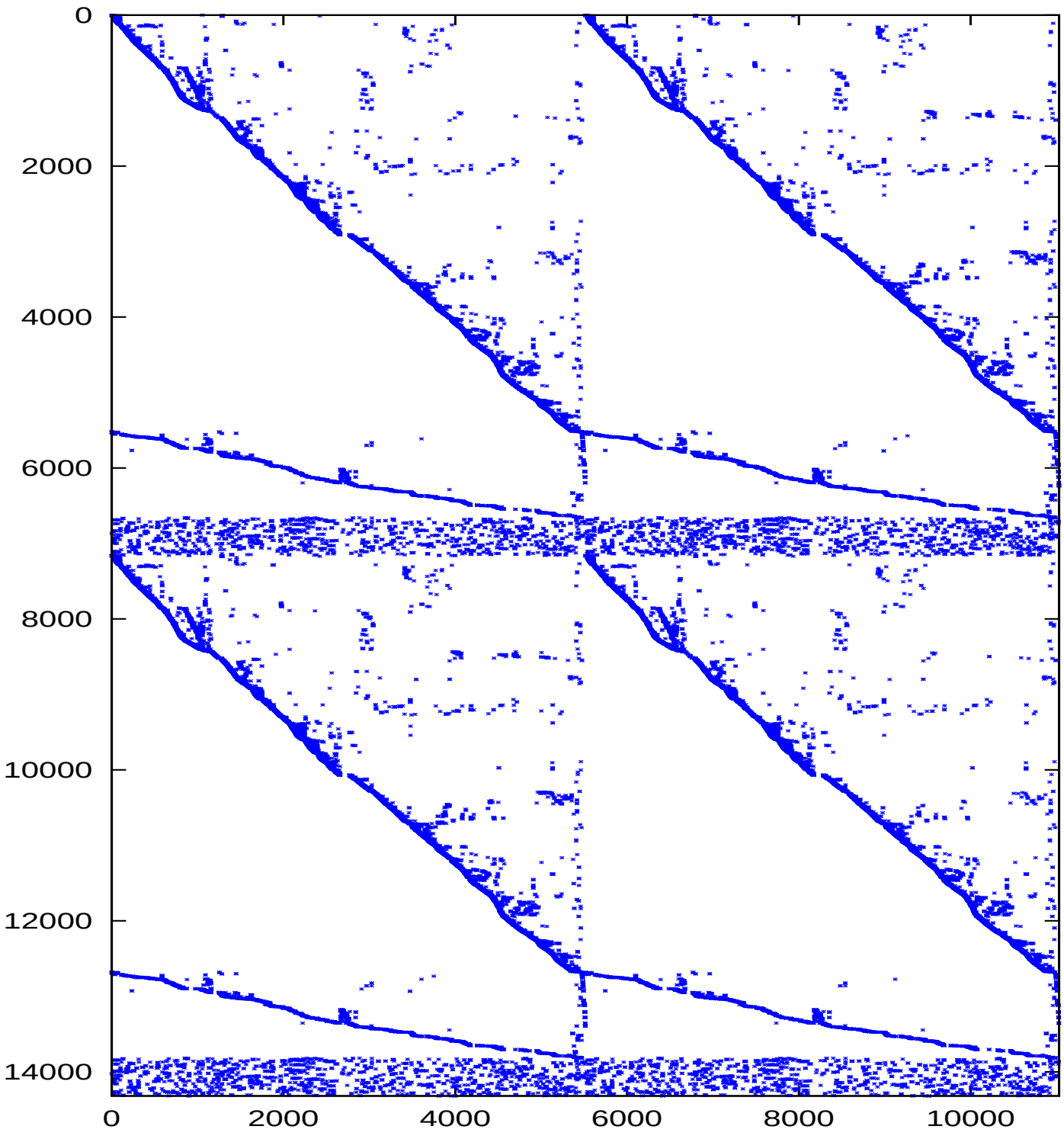

Figure B.1: Original Matrix psse1

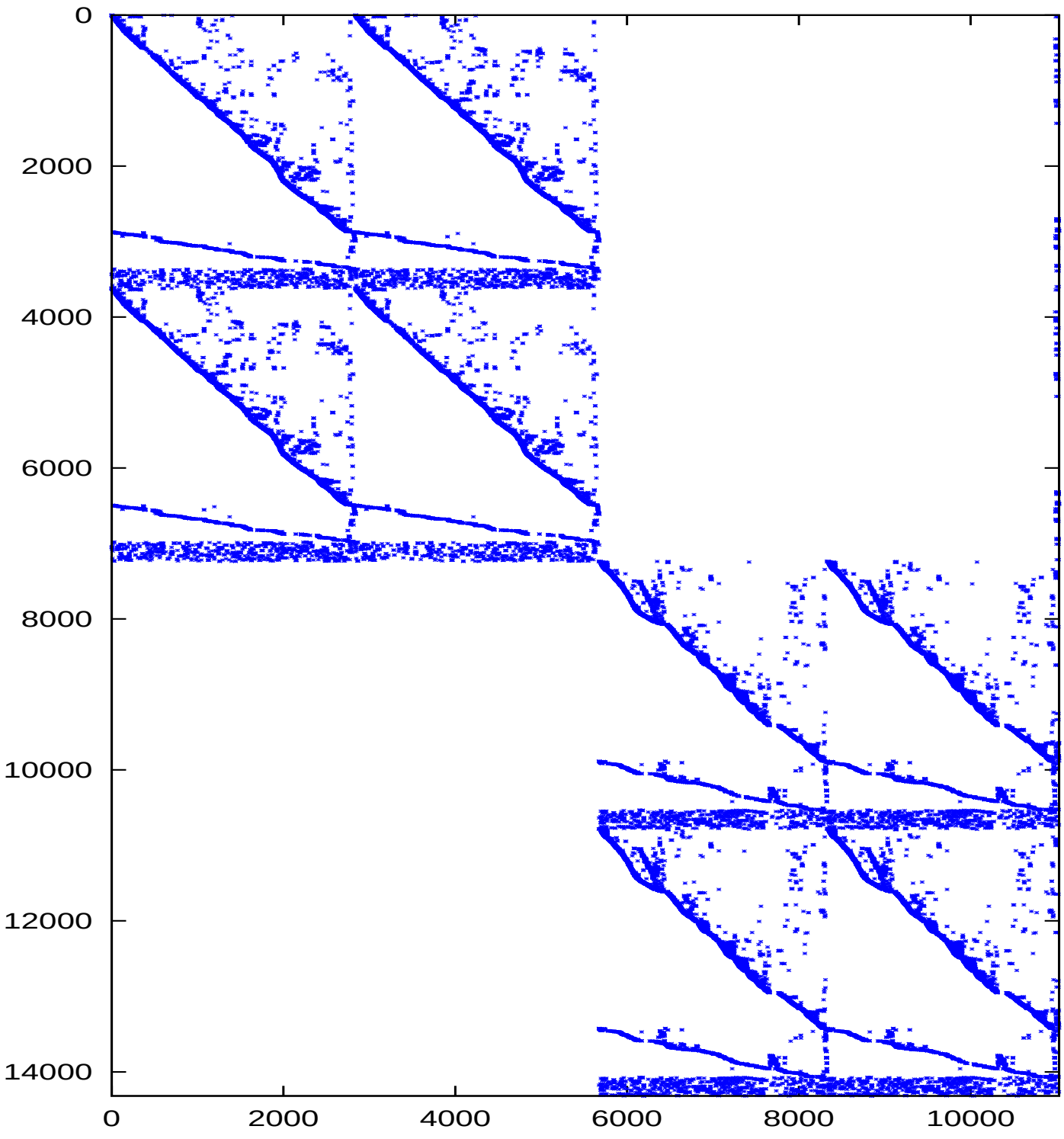

Figure B.2: Partitioned Matrix psse1 when $B = 1$ and $K = 2$

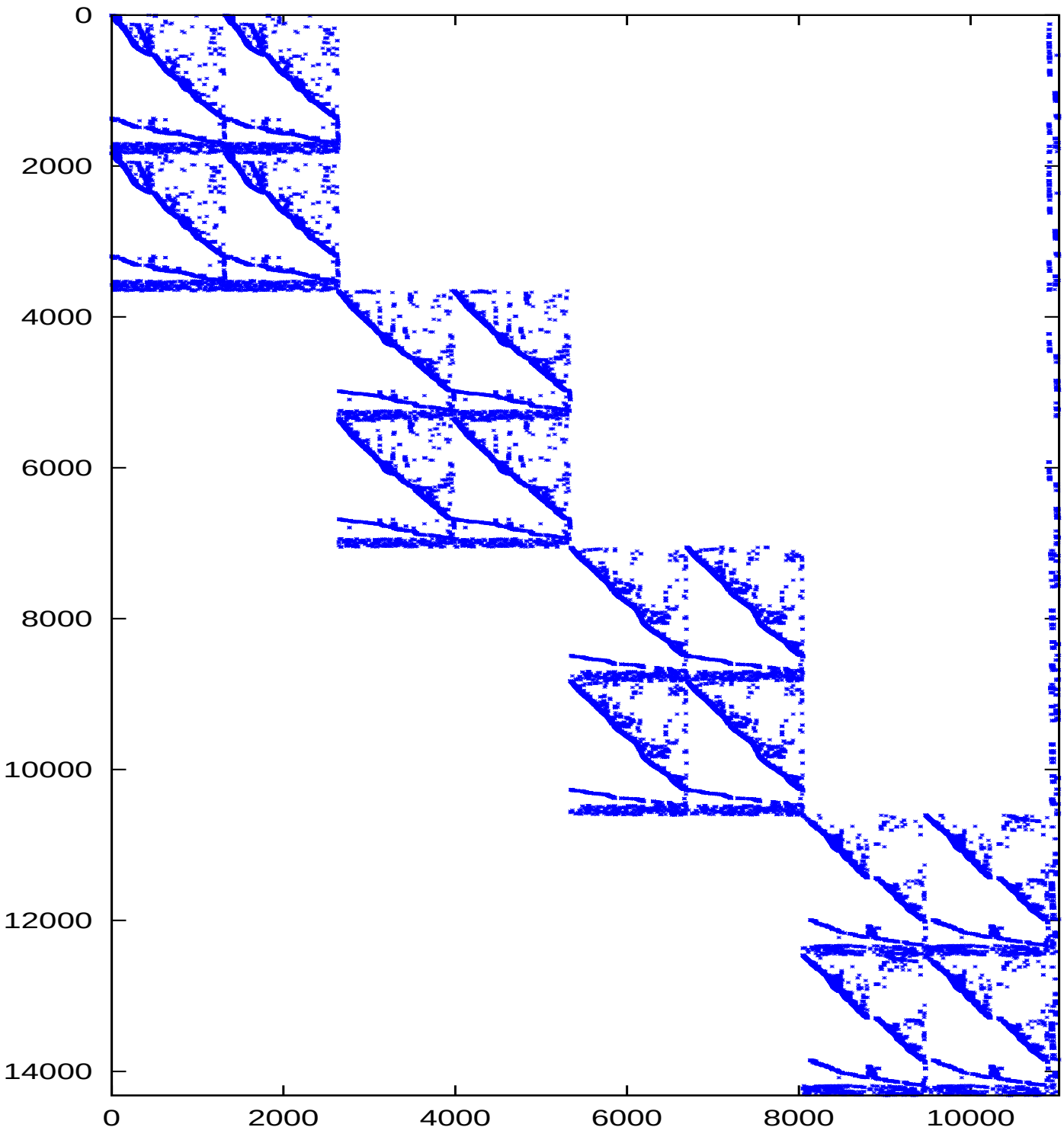

Figure B.3: Partitioned Matrix psse1 when $B = 2$ and $K = 4$

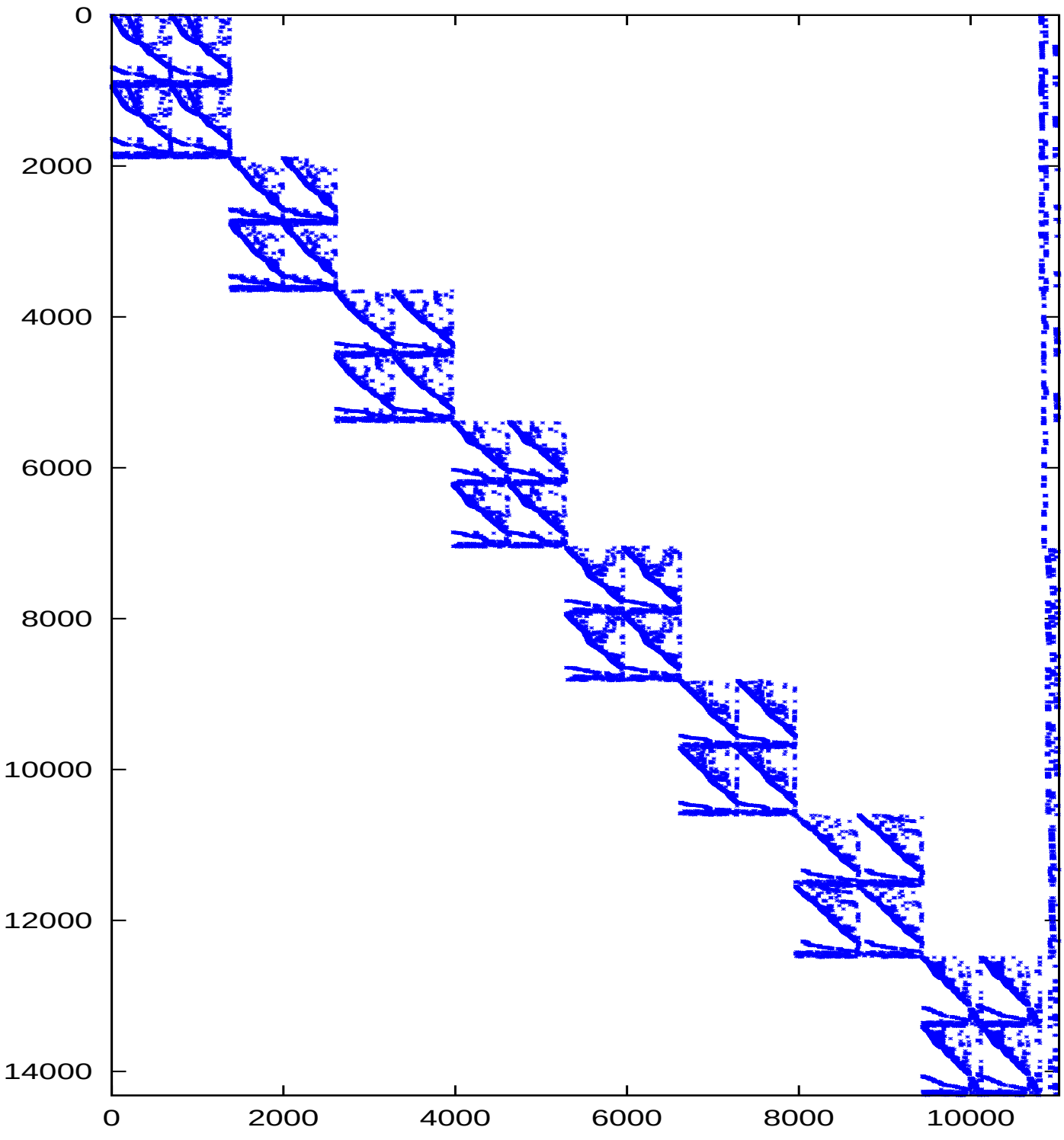

Figure B.4: Partitioned Matrix psse1 when $B = 3$ and $K = 8$

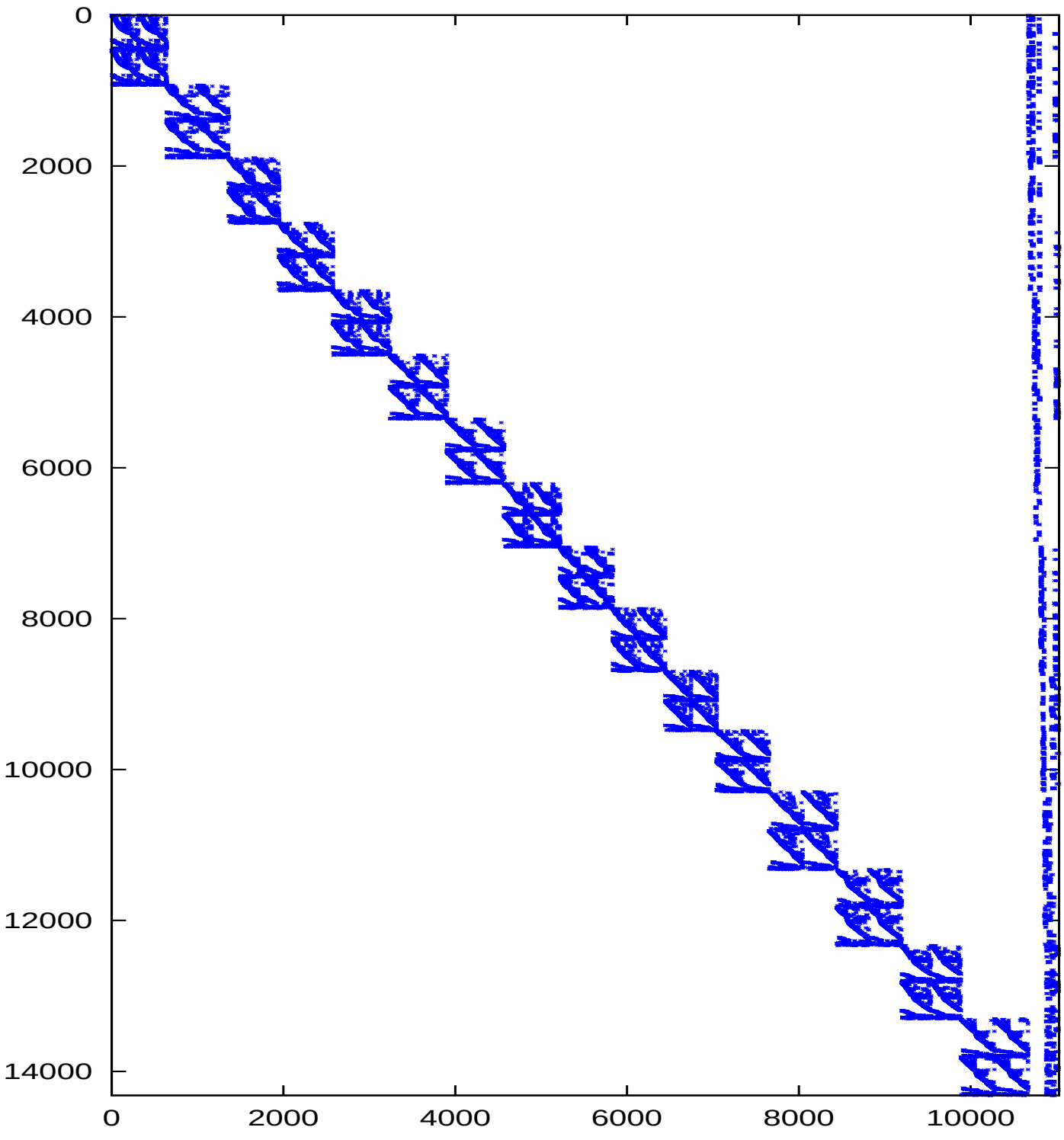

Figure B.5: Partitioned Matrix psse1 when $B = 4$ and $K = 16$

Bibliography

[1] I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. *Parallel Processing Symposium, International*, 0:0298, 1998.

[2] C. Aykanat, A. Pınar, and Ü. V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. Technical Report BU–CE–0203, Computer Engineering Department, Bilkent University, Turkey, 2002. a shorter version appears on SIAM Journal on Scientific Computing, Vol. 26, No. 6, 2004.

[3] C. Aykanat, A. Pınar, and Ü. V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. *SIAM Journal on Scientific Computing*, 26(6):1860–1879, 2004.

[4] T. N. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorization. In *Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 445–452, 1993.

[5] Ü. V. Çatalyürek and C. Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplications. In *Proceedings of 3rd International Symposium on Solving Irregularly Structured Problems in Parallel, Irregular'96*, volume 1117 of *Lecture Notes in Computer Science*, pages 75–86. Springer-Verlag, 1996.

[6] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[7] Ü. V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at `http://bmi.osu.edu/~umit/software.htm`, 1999.

[8] Ü. V. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2d decomposition of sparse matrices. *Parallel and Distributed Processing Symposium, International*, 3:30118b, 2001.

[9] Ü. V. Çatalyürek, C. Aykanat, and B. Ucar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *Submitted to SIAM Journal on Scientific Computing*.

[10] J. M. Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. In *International Journal of Parallel Programming*, pages 425–433, 2001.

[11] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured euler solver using software primitives. In *AIAA Journal*, 1992.

[12] J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[13] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. *SIGPLAN Not.*, 34(5):229–241, 1999.

[14] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Clarendon Press, New York, NY, USA, 1989.

[15] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181. IEEE, 1982.

[16] G. Haase, M. Liebmann, and G. Plank. A hilbert-order multiplication scheme for unstructured sparse matrices. *Int. J. Parallel Emerg. Distrib. Syst.*, 22(4):213–220, 2007.

[17] H. Han and C. Tseng. Exploiting locality for irregular scientific codes. *IEEE Trans. Parallel Distrib. Syst.*, 17(7):606–618, 2006.

[18] K. Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126:106–130, 2000.

[19] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical report, Sandia National Laboratories, 1993.

[20] G. Jin and M. J. Crummey. Using space-filling curves for computation reordering. In *Proceedings of the Los Alamos Computer Science Institute*, 2005.

[21] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[22] G. Karypis and V. Kumar. *MeTiS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*. University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

[23] G. Karypis, V. Kumar, R. Aggarwal, and S. Shekhar. *hMeTiS A Hypergraph Partitioning Package Version 1.0.1*. University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

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

[25] J. Koster. Parallel Templates for Numerical Linear Algebra, a High-Performance Computation Library. Master's thesis, Utrecht University, July 2002.

[26] M.S. Lam and M.E. Wolf. A data locality optimizing algorithm. *SIGPLAN Not.*, 39(4):442–459, 2004.

[27] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Willey–Teubner, Chichester, U.K., 1990.

[28] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973.

[29] K.S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, 1996.

[30] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 140–152, New York, NY, USA, 1988. ACM.

[31] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 192, Washington, DC, USA, 1999. IEEE Computer Society.

[32] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 30, New York, NY, USA, 1999. ACM.

[33] L. Rauchwerger. Run-time parallelization: its time has come. *Parallel Comput.*, 24(3-4):527–556, 1998.

[34] Y. Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. Society for Industrial and Applied Mathematics, April 2003.

[35] M. M. Strout and P. D. Hovland. Metrics and models for reordering transformations. In *Proc. of the Second ACM SIGPLAN Workshop on Memory System Performance (MSP04)*, pages 23–34, Washington DC., June 2004. ACM.

[36] A. D. Timothy. University of florida sparse matrix collection. *NA Digest*, 92, 1994.

[37] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521+, 2005.

[38] R. W. Vuduc and H. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. pages 807–816. 2005.

[39] J. White. On improving the performance of sparse matrix-vector multiplication. In *In Proceedings of the International Conference on High-Performance Computing*, pages 578–587. IEEE Computer Society, 1997.

[40] A. N. Yzelman and Rob H. Bisseling. Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing*, 31(4):3128–3154, 2009.