LOCALITY AWARE REORDERING FOR SPARSE TRIANGULAR SOLVE

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Tuğba Torun

September, 2014

ABSTRACT

LOCALITY AWARE REORDERING FOR SPARSE TRIANGULAR SOLVE

Tuğba Torun
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
September, 2014

Sparse Triangular Solve (SpTS) is a commonly used kernel in a wide variety of scientific and engineering applications. Efficient implementation of this kernel on current architectures that involve deep cache hierarchy is crucial for attaining high performance. In this work, we propose an effective framework for cache-aware SpTS.

Solution of sparse linear symmetric systems utilizing the direct methods require the triangular solve of the form $LUz = b$, where L is lower triangular factor and U is upper triangular factor. For cache utilization, we reorder the rows and columns of the L factor regarding the data dependencies of the triangular solve. We represent the data dependencies of the triangular solve as a directed hypergraph and construct an ordered partitioning model on this structure. For this purpose, we developed a variant of Fiduccia-Mattheyses (FM) algorithm which respects the dependency constraints. We also adopt the idea of splitting L factors into dense and sparse components and solving them seperately with different autotuned kernels for achieving more flexibility in this process. We investigate the performance variation of different storage schemes of L factors and the corresponding sparse and dense components. We utilize autotuning provided by Optimized Sparse Kernel Interface (OSKI) to reduce performance degradation that incurs due to the gap between processors and memory speeds. Experiments performed on real-world datasets verify the effectiveness of the proposed framework.

Keywords: Sparse matrices, triangular solve, cache locality, matrix reordering, hypergraph partitioning, directed hypergraph.

ÖZET

SEYREK ÜÇGENSEL SİSTEMLERİN ÖNBELLEK YERELLİĞİNE GÖRE YENİDEN SIRALANMASI

Tuğba Torun
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Eylül, 2014

Seyrek üçgensel sistem çözümü, bir çok bilimsel ve mühendislik uygulamalarında yaygın olarak kullanılan bir çekirdek işlemdir. Bu çekirdek işlemin çok seviyeli önbellekler üzerinde etkili bir şekilde yürütülmesi, yüksek permormans elde etmek açısından önemlidir. Biz bu çalışmada, seyrek üçgensel sistem çözümünde kullanılmak üzere etkili bir çerçeve sunuyoruz.

Seyrek lineer sistemlerin direkt metod ile çözümü, L alt üçgensel faktör ve U üst üçgensel faktör olmak üzere, $LUz = b$ biçimindeki bir üçgensel denklem çözümünü gerektirir. Önbelleği kullanmak için, üçgensel sistemdeki veri bağlılığını da göz önüne alarak L faktörünün satır ve sütunlarını yeniden sıraladık. Üçgensel sistemdeki veri bağlılıklarını yönlü bir hiperçizge olarak kodlayıp, bu yapı üzerinde sıralı bir bölümleme modeli inşa ettik. Bu amaçla, bağlılık sınırlamalarına riayet edecek biçimde Fiduccia-Mattheyses (FM) algoritmasını farklı bir şekilde yeniden geliştirdik. Ayrıca, bu işlemde daha fazla esneklik elde etmek için, L faktörlerini seyrek ve yoğun parçalara ayırma fikrini benimsedik ve her bir parçayı otomatik ayarlanmış farklı çekirdek yöntemlerle çözdük. Farklı depolama yöntemleri kullanarak L faktörünün ve buna karşılık gelen seyrek ve yoğun parçaların performans değişimini inceledik. İşlemci ve hafıza arasındaki hız farklarından kaynaklanan performans kayıplarını önlemek için, OSKI tarafından sağlanan otomatik ayarlama yöntemlerinden faydalandık. Gerçek veriler üzerinde yürütülen deneyler, önerilen modelin etkinliğini doğrular niteliktedir.

Anahtar sözcükler: Seyrek matrisler, üçgensel sistemler, önbellek yerelliği, matrisi yeniden sıralama, hiperçizge bölümleme, yönlü hiperçizge .

Acknowledgement

I would like to thank my supervisor Prof. Dr. Cevdet Aykanat for his valuable advices, suggestions and assistence throughout the development of this thesis.

I owe thanks to Prof. Dr. Uğur Güdükbay and Assoc. Prof. Dr. Oya Ekin Karaşan for spending their time to evaluate and comment on the thesis. I also thank Kadir Akbudak for his helps and contributions for the progress of this study.

I am grateful to my family for their infinite moral support and help. My special thank goes to my husband F. Şükrü Torun for his endless support, encouragement and patience all along my studies. He deserves the best I can give for facilitating and beautifying my life.

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

To my family...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Developments in computer architectures bring along the performance gap between the processor and memory speeds. This gap becomes more critical due to the hierarchical memory structure. Yet it can be reduced by exploiting high level memory units (caches) efficiently, which are faster but smaller memories compared to RAM. Hence extra attention is needed for the applications whose performance highly depends on memory utilization.

Data localities can be exploited if the data is accessed in consecutive memory localitions. An application is called *regular* if it enables such utilizations of data localities and called *irregular*, otherwise. Utilizing data locality in irregular computations is a challenging task due to the irregularity in memory access patterns.

Sparse Triangular Solve (SpTS) is an instance of such an irregular computation and it is one of the most important kernels in numerical computing. It arises in several scientific and engineering applications including direct solution of sparse linear systems, preconditioning of iterative methods and least squares problems [1, 2, 3]. It requires to solve the sparse triangular system of the form $Tx = b$ with respect to dense solution vector x, given a sparse triangular matrix T and dense vector b.

The performance of SpTS is directly effected by the sparsity pattern of the triangular matrix. The data localities in SpTS can be utilized when the nonzeros of triangular matrix stay close to each other. However in most of the real-world applications, the nonzero distributions and hence the memory access patterns are irregular, which yields a poor utilization of cache. Nevertheless, altering the nonzero distribution to obtain more regular patterns is possible by applying proper reordering methods.

In literature, the method of reordering rows/ columns of matrices for cache utilization has been frequently exploited in applications like sparse matrix-vector multiplication (SpMxV) [4, 5, 6]. However, these reordering techniques has not been adressed for cache utilization in SpTS to the best of our knowlegde. The main reason for this case might be the low flexibility in SpTS due to high dependencies between computations unlike other applications: Solution of SpTS include computations which may depend on others, according to the nonzero distribution in triangular solve. Hence these order of computations must be protected while reordering the rows/ columns associated with the corresponding computations.

In this thesis, we investigate a reordering method on the rows and columns of the triangular matrix for better cache utilization, by taking the data dependencies into account. To obtain such a reordering, we represent dependencies among the computations of SpTS as a directed hypergraph. We develop a novel ordered partitioning model upon this directed hypergraph representation. We argue that cutsize reduction in this directed hypergraph partitioning (dHP) model corresponds to minimizing cache misses in SpTS. We extend the Fiduccia-Mattheyses (FM) algorithm [7] with more feasibility restrictions and updates in order to capture the ordered-directed structure of the required partitioning model. The rows and the columns of the triangular factor is reaordered symmetrically according to the final order of vertices obtained by the dHP model.

Based upon this reordering model, we propose a framework for efficient cache utilization in SpTS. We exploit the idea of splitting the triangular matrix into dense and sparse parts proposed by [8] in order to obtain higher flexibilities for our dHP model. By this way the SpTS problem is converted into solving a sub-SpTS, a SpMxV and a dense triangular solve. We utilize our dHP model for solving this smaller SpTS which shows a more flexible charecteristic than the original SpTS in terms of having fewer data dependencies and the ease of reordering. We also apply the hypergraph-partitioning based reordering method of rectangular matrices proposed in [6] for cache utilization in SpMxV in order to enhance the overall SpTS performance. The autotuning provided by OSKI library [9] is exploited to utilize higher levels of memory more effectively.

The rest of the thesis is organized as follows: Necessary background material for this study is provided in Chapter 2. In Chapter 3, we review the previous works which are most relevant to our framework. The directed hypergraph partitioning model is introduced and demonstrated in Chapter 4. We present the experimental results in Chapter 5 and conclude the thesis in Chapter 6.

Chapter 2

Background

2.1 Direct Method for Solving Linear Systems

Most of the scientific computing problems requires the solution of a linear system. Linear systems are used in linear programming, discretization of nonlinear systems and differential equations. In a linear system of the form

$$Az = b, \tag{2.1}$$

the aim is to find the column vector z, with a given coefficient matrix A and a column vector b. There are two classes of methods for solving systems of linear equations: Direct methods and iterative methods.

In iterative methods, beginning with an initial approximation, a sequence of approximate solutions is computed until a desired accuracy is obtained [10]. On the other hand, in direct methods, the solution is obtained in a finite number of operations. In these methods, the coefficient matrix of the linear system is transformed or factorized into a simpler form which can be solved easier. Direct methods have been preferred over iterative methods for solving linear systems mainly because of their stability and robustness [11].

In the direct solution of linear systems, the coefficient matrix A is factorized into its lower triangular factor L and upper triangular factor U as

$$A = LU. \tag{2.2}$$

A *lower triangular matrix* is a square matrix whose all of the nonzero entries lie below and on the main diagonal. Similarly, an *upper triangular matrix* is a square matrix which has all of its nonzero entries above and on the main diagonal. A square matrix which is either lower triangular or upper triangular is referred to as a *triangular matrix*.

The diagonal entries of the triangular factors should be nonzero in order to avoid round-off errors in the upcoming operations of direct method. For this reason, permuting the rows and columns of the coefficient matrix is allowed in the LU factorization. This procedure is called *pivoting* and there are two ways to do it: *Partial pivoting* reorders the rows of the coefficient matrix during the LU factorization in order to move the entry with maximum absolute value of a column to the diagonal. In *complete pivoting*, both the rows and the columns can be permuted in order to make the diagonal entry to have the largest absolute value in the entire remaining unprocessed submatrix. This procedure enhances the numerical stability.

In particular, if the coefficient matrix A is symmetric and positive definitive (having all eigenvalues positive), the diagonal elements of L and U factors become nonzero without requiring any pivoting. Furthermore in this case there exist a unique factorization, namely *Cholesky factorization*, such that $U = L^T$ holds in Equation (2.2).

After obtaining the triangular factors, the Equation (2.2) is substituted into the original linear system Equation (2.1) and the problem becomes

$$LUz = b, \tag{2.3}$$

which is equivalent to solving the following set of two equations:

$$Lx = b, \tag{2.4}$$
$$Uz = x. \tag{2.5}$$

Here, Equation (2.4) is solved with a procedure called *forward substitution* and Equation (2.5) is solved with *backward substitution.* First the forward substitution (2.4) is solved to find x vector, and then the x vector is substituted in the backward substitution Equation (2.5) to get the z vector.

2.1.1 Forward Substitution

Let us assume that $L = (l_{i,j})_{0\leq i,j\leq n}$ is a lower triangular matrix. Note that by the definition of lower triangularity, $l_{i,j} = 0$ whenever $i < j$. Then the lower-triangular system $Lx = b$ with $x = (x_i)_{0\leq i\leq n}$ and $b = (b_i)_{0\leq i\leq n}$

$$\begin{bmatrix} l_{1,1} & & & \\ l_{2,1} & l_{2,2} & & \\ & & \ddots & \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

is equivalent to the following system of equations:

$$\begin{aligned} l_{1,1}x_1 &= b_1 \\ l_{2,1}x_1 + l_{2,2}x_2 &= b_2 \\ l_{3,1}x_1 + l_{3,2}x_2 + l_{3,3}x_3 &= b_3 \\ &\vdots \\ l_{n-1,1}x_1 + l_{n-1,2}x_2 + \cdots + l_{n-1,n-1}x_{n-1} &= b_{n-1} \\ l_{n,1}x_1 + l_{n,2}x_2 + \cdots + l_{n,n-1}x_{n-1} + l_{n,n}x_n &= b_n \end{aligned} \tag{2.6}$$

By these equations, the x vector can be solved by computing x_i entries from the following equations in order:

$$\begin{aligned}
x_1 &= \frac{b_1}{l_{1,1}} \\
x_2 &= \frac{b_2 - l_{2,1}x_1}{l_{2,2}} \\
x_3 &= \frac{b_3 - l_{3,2}x_2 - l_{3,1}x_1}{l_{3,3}} \\
&\vdots \\
x_n &= \frac{b_n - l_{n,n-1}x_{n-1} - \cdots - l_{n,2}x_2 - l_{n,1}x_1}{l_{n,n}}
\end{aligned} \tag{2.7}$$

Forward substitution method finds the x_i entries in the increasing order of the indices: x_1 is computed first and then substituted forward into the next equation for solving x_2, and this process goes on until x_n.

There are two popular ways of implementing forward substution: Row-oriented and column-oriented.

2.1.1.1 Row-Oriented Forward Substitution

Row-oriented forward substitution solves Equation (2.7) one row at a time. Firstly, x_1 is found from the first equation in (2.7). Then x_2 is calculated by substituting x_1 in the second equation of (2.7). Similarly, x_3 is obtained by substituting the values x_1 and x_2 in the third equation of (2.7) and this process is continued until the last equation to get x_n. Algorithm 1 represents the pseudocode of row-oriented forward substitution.

Algorithm 1 Row-oriented forward substitution

```
1: for $i \leftarrow 1$ to $n$ do
2:     $x_i \leftarrow b_i$
3:     for $j \leftarrow 1$ to $i - 1$ do
4:         $x_i \leftarrow x_i - l_{i,j} \times x_j$
5:     $x_i \leftarrow \frac{x_i}{l_{i,i}}$
```

2.1.1.2 Column-Oriented Forward Substitution

Column-oriented forward substitution computes Equation (2.7) one column at a time. For $i > j$, the $l_{i,j}x_j$ value is subtracted from x_i immediately after computing x_j, rather than doing all substractions at the computation time of x_i as in the row-oriented substitution. Algorithm 2 shows the pseudocode of column-oriented forward substitution.

Algorithm 2 Column-oriented forward substitution

```
1: for $j \leftarrow 1$ to $n$ do
2:     $x_j \leftarrow b_j$
3:     $x_j \leftarrow \frac{x_j}{l_{j,j}}$
4:     for $i \leftarrow j + 1$ to $n$ do
5:         $x_i \leftarrow x_i - l_{i,j} \times x_j$
```

2.1.2 Backward Substitution

Let us assume that $U = (u_{i,j})_{0 \leq i,j \leq n}$ is an upper triangular matrix. Note that by the definition of upper triangularity, $u_{i,j} = 0$ whenever $i > j$. Then the upper-triangular system $Uz = x$ with $z = (z_i)_{0 \leq i \leq n}$ and $x = (x_i)_{0 \leq i \leq n}$

$$\begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ & u_{2,2} & \cdots & u_{2,n} \\ & & \ddots & \\ & & & u_{n,n} \end{bmatrix} \cdot \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

is equivalent to the following system of equations:

$$\begin{aligned} u_{1,1}z_1 + u_{1,2}z_2 + u_{1,3}z_3 + \cdots + u_{1,n}z_n &= x_1 \\ u_{2,2}z_2 + u_{2,3}z_3 + \cdots + u_{2,n}z_n &= x_2 \\ &\vdots \\ u_{n-1,n-1}z_{n-1} + u_{n-1,n}z_n &= x_{n-1} \\ u_{n,n}z_n &= x_n \end{aligned} \tag{2.8}$$

By these equations, the x vector can be solved by computing x_i entries from the following equations in order:

$$\begin{aligned} z_n &= \frac{x_n}{u_{n,n}} \\ z_{n-1} &= \frac{x_{n-1} - u_{2,1}z_1}{u_{n-1,n-1}} \\ &\vdots \\ z_1 &= \frac{x_1 - u_{1,2}z_2 - u_{1,3}z_3 - \cdots - u_{1,n}z_n}{u_{1,1}} \end{aligned} \tag{2.9}$$

Backward substitution method finds the z_i entries in the decreasing order of the indices: It first computes z_n, then substitute it back into the next equation to find z_{n-1}, and continue backwards until z_1.

In this thesis, we will focus on forward substitution since the backward substitution is the reverse of forward substitution and the proposed models and methods can be applied analogously without loss of generality.

2.2 Graph Partitioning

A *graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a representation consisting of a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$. Each edge e_{ij} connects two distinct vertices v_i and v_j. Each vertex v_i has weight $w(v_i)$ and each edge e_{ij} has cost $c(e_{ij})$. In directed graphs, edges also have directions associated with them. Directed edges are ordered pairs of vertices.

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $\Pi = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$ is called a K-way partition of the vertex set $\mathcal{V}$ if each part $\mathcal{V}_k$ contains at least one vertex, the intersection of two distinct parts is empty, and the union of all parts is equal to $\mathcal{V}$. Specifically, a K-way partition is called a *bipartition* if $K = 2$.

The sum of the weights of the vertices in part $\mathcal{P}_k$ is called the *weight* of that part and denoted as W_k. A K-way partition of $\mathcal{G}$ is said to satisfy the *balance criteria* and be *balanced* if

$$W_k \leq (1+\varepsilon)W_{avg}, \quad \text{for } k = 1, 2, \ldots, K \tag{2.10}$$

where ε is a predetermined, maximum allowable imbalance ratio and W_{avg} is the average part weight, i.e. $W_{avg} = (\sum_{1\leq k\leq K} W_k)/K$.

An edge is called *cut (external)* if it connects two vertices from different parts, and called *uncut (internal)* otherwise. Let the set of cut edges is represented by $\mathcal{E}_{cut}$. Then the *cutsize* of a partition Π is defined as the sum of the costs of cut edges:

$$cutsize(\Pi) = \sum_{e_{ij} \in \mathcal{E}_{cut}} c(e_{ij}). \tag{2.11}$$

The graph partitioning problem is to partition the graph into K disjoint parts with minimum cutsize, while satisfying the balanca criteria 2.10. This problem is known to be NP-hard even for unweighted graph bipartitioning [12].

2.3 Hypergraph Partitioning

Hypergraphs are generalization of graphs in which each hyperedge can connect more than two vertices as opposed to the edges in graphs connecting only two vertices. A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ consists of a set of vertices $\mathcal{V}$ and a set of hyperedges (nets) $\mathcal{N}$. Each net $n_j \in \mathcal{N}$ connects an arbitrary subset of vertices, i.e., $n_j \subseteq \mathcal{V}$. The set of vertices connected by a net n_j is called its *pins* and represented by $Pins(n_j)$. The set of nets connected to a vertex v_i is represented by $Nets(v_i)$. The weight of vertex $v_i \in \mathcal{V}$ and the cost of the net $n_j \in \mathcal{N}$ are denoted as $w(v_i)$ and $c(n_j)$, respectively.

The definitions of K-way partition and balance criteria for the graph partitioning hold for hypergraphs as well. For a partition Π of $\mathcal{H}$, if a net connects at least one pin in a part, then that net is said to *connect* that part. The set of parts connected by a net n_j is called the *connectivity set* Λ_j of n_j. The number of parts connected by a net n_j is called the *connectivity* of n_j and denoted by $\lambda_j = |\Lambda_j|$. If a net n_j connects more than one part (i.e., $\lambda_j > 1$), then it is said to be a *cut* (*external*) net, and otherwise (i.e., $\lambda_j = 1$) it is called an *uncut* (*internal*) net. The *cutsize* of a partition Π is defined as

$$cutsize(\Pi) = \sum_{n_j \in \mathcal{N}_E} (\lambda_j - 1), \tag{2.12}$$

in which each cut net n_j contributes $\lambda_j - 1$ to the cutsize. The partitioning objective is to minimize the cutsize while maintaining the balance criteria (2.10). The hypergraph partitioning problem is also known to be NP-hard [13].

For K-way hypergraph partitioning, a paradigm called *recursive bisection* is widely used [14, 15]. In this paradigm, the hypergraph is bipartitioned into two parts initially. Then each part of this bipartition is further bipartitioned recursively until a predetermined part count K is obtained or the weight of a part drops below a predetermined maximum part weight value. This paradigm is especially preferred for hypergraph partitioning if the number of parts is not known in advance.

Most of the hypergraph bipartitioning algorithms relies on Fiduccia-Mattheyses (FM) [7] and Kernighan-Lin (KL) [16] heuristics which were proposed for reducing the cutsize of a bipartition. KL-based heuristics swap two vertices from different parts of the bipartition while FM-based heuristics move a single vertex from one part to the other.

In FM-based heuristics, the change in the cutsize of the bipartition in the case of moving a vertex to other part is called the *gain* of that vertex. The gains of vertices can be stored in buckets or heaps [17, 18]. The FM heuristic consists of multiple passes each of which begins with unlocked vertices. At each step of a pass, the vertex with the highest gain value is selected, moved to the other part and locked. After each move, the gain values of the unlocked neighbors of the moved vertex are updated and the improvement in the cutsize is stored. A pass terminates if all vertices become locked or no feasible move reamins. At the end of each pass, the partition which gives the minimum cutsize is restored. Several passes are performed until the reduction in the cutsize drops below a predetermined threshold.

The hypergraph partitioning model is widely used to represent and partition sparse matrices [19]. Mainly there are two hypergraph models proposed for sparse matrix partitioning, namely row-net and column-net models [20, 21]. In row-net hypergraph model, the nets represents the rows of a matrix and the vertices represent the columns of the matrix whereas in the column-net model, nets represent the columns and vertices represent the rows.

2.4 Directed Graph Representation for Dependencies in Forward Substitution

In a forward substitution with dense L matrix, the computation order of x_i entries should be exactly the same as the order in (2.7). The reason is that all the x_j entries with $j < i$ should be computed beforehand for substituting the $l_{i,j}x_j$ value in the computation equation of x_i entry. For a sparse L matrix, on the other hand, the computation of x_i does not depend on x_j $(j < i)$ if the $l_{i,j}$ entry of the L matrix is zero, since then the $l_{i,j}x_j$ value will be automatically zero and will not be used in the computation of x_i. Yet the computation of x_i should still wait the value of x_j $(j < i)$ if the $l_{i,j}$ entry is a nonzero.

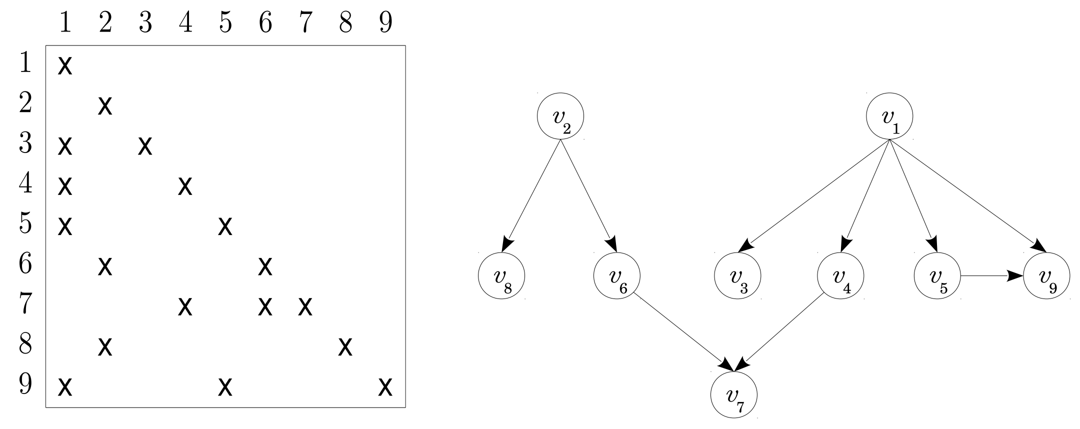
In other words, the i^{th} equation of (2.7) can not be solved before the j^{th} equation if $l_{i,j} \neq 0$ $(j < i)$. Hence for row-oriented forward substitution, row i

can not be processed before row j if $l_{i,j} \neq 0$ $(j < i)$. Row i is said to depend on row j if $l_{i,j} \neq 0$ with $j < i$. Conversely, if $l_{i,j} = 0$ with $j < i$, then row i is said to be independent from row j.

Similarly for column-oriented forward substitution, column i can not be processed before column j if $l_{i,j} \neq 0$ $(j < i)$. This dependency is expressed in terms of the columns of the matrix as follows: Column i is said to depend on column j if $l_{i,j} \neq 0$ and said to be independent from column j if $l_{i,j} = 0$ for $j < i$.

These dependencies can also be converted to a directed graph representation. According to the data dependencies in row-oriented forward substitution, a directed dependence graph is constructed corresponding to the lower triangular factor L such that vertex v_i represents row i and there is a directed edge e_{ji} from vertex j to i whenever $l_{i,j} \neq 0$ $(j < i)$. Theoretically speaking, for column-oriented forward substitution, vertex v_i represents column i but the edge definitions are the same and so the corresponding dependence graph is equivalent.

Figure 2.1(a) illustrates a sample lower triangular factor L of size 9×9. In Figure 2.1(b), the corresponding dependence graph of L factor is constructed. Directed edges e_{13}, e_{14}, e_{15}, e_{19}, e_{26}, e_{28}, e_{47}, e_{59}, e_{67} correspond to the nonzero entries $l_{3,1}$, $l_{4,1}$, $l_{5,1}$, $l_{9,1}$, $l_{6,2}$, $l_{8,2}$, $l_{7,4}$, $l_{9,5}$, $l_{7,6}$ of the L factor respectively.



(a) The lower triangular matrix L (b) The dependence graph of L factor

Figure 2.1: Directed dependence graph representation for row/ column oriented forward substitution

2.5 Sparse Matrix Storage Schemes

We will beriefly describe the compressed storage formats which are the most common sparse matrix storage schemes, and a variant of this format, namely the block compressed storage format, which improves the performance in most of the sparse matrix operations.

2.5.1 Compressed Storage Formats

Sparse matrices are often stored in a compressed format in which only the nonzeros of the sparse matrices and their localitions are stored. Mainly, each nonzero element of the matrix is stored into a linear array and some additional arrays are provided to describe the locations of the nonzeros in the matrix. This format is frequently preferred for sparse matrices since it does not store any unnecessary information about the zero elements that dominate the sparse matrices [22, 23].

There are two main compressed storage schemes, namely *Compressed Storage by Rows (CSR)* and *Compressed Storage by Columns (CSC)*. CSR scheme stores the nonzeros in a row-major format, i.e. stores the nonzeros of a row consecutively while CSC scheme stores the nonzeros in a column-major format, i.e., stores the nonzeros of columns consecutively.

2.5.2 Block Compressed Storage Formats

Block Compressed Storage by Rows (BSCR) and Block Compressed Storage by Columns (BCSC) are the modified versions of CRS and CCS formats to exploit dense block patterns respectively. The matrix is partitioned into small blocks whose sizes evenly divide the dimensions of the matrix. Each block is treated as a dense matrix even if it has some zeros in these blocks but the blocks consisting of only zero elements are not stored. In BSCR (BCSC) format, the column (row) indices are stored block-by-block and the pointers reference to rows (columns) of blocks.

For arithmetic operations with sparse matrices having dense sub-matrices, using block storage formats is considerably more efficient than using regular compressed sparse storage formats. Especially for matrices with large block dimensions, the block compressed storage formats significantly reduce the time spent in performing indirect addressing and the memory requirements for storage locations with respect to the usual compressed storage formats.

Determining the dimensions of dense block patterns which leads to the fastest implementation is referred as *tuning*. This process should be handled at run-time automatically (which is called *autotuning*) for sparse matrices since the best data structure depends on the sparsity pattern of the matrix. The Optimized Sparse Kernel Interface (OSKI) [9] library provides an autotuning framework for SpTS.

2.6 Data Locality in Forward Substitution

Here, we will briefly discuss how data locality can be achieved in the forward substitution $Lx = b$. The row-oriented forward substitution algorithm puts all of the previously computed entries x_j into the equation just before computing x_i where $(j < i)$. On the other hand, the column-oriented forward substitution algorithm updates x_i entry immediately after computing x_j for each $j < i$. These two algorithms seem to have the same computation order for the x vector entries and the same number of operations, but they process data in different order and hence access memory in different patterns.

There are mainly two ways to exploit cache locality: Temporal and spatial locality. *Temporal locality* refers to reuse of data which is previously fetched and still staying in the cache. *Spatial locality* allows reuse of data which was previously prefetched when a data from a nearby location was brought to the cache.

In row-oriented forward substitution, spatial locality in terms of the nonzeros of lower triangular factor can be automatically exploited if it is stored in a row-major (e.g. CSR, BCSR) format. Similarly for column-oriented substitution, we can automatically exploit spatial locality if we store the lower triangular factor in a column-major (e.g., CSC, BCSC) format. This is because in these cases the nonzero entries of the component matrix stored and processed consecutively. The temporal locality for the nonzeros of lower triangular factor is not feasible since each entry is accessed only once. The same reasons hold to say that the spatial locality is automatically achieved and the temporal locality cannot be exploited in accessing the entries of the b-vector.

The spatial locality in accessing x-vector entries is feasible since the entries are operated consecutively in the case of using a row-major compressed storage format for row-oriented forward substitution and using a column-major compressed storage format for column-oriented forward substitution. For row-oriented forward substitution, the temporal locality in accesing x-vector entries is feasible because of the reuse of previously stored data when processing former rows. For column-oriented forward substitution, the temporal locality in accesing x-vector entries is feasible since the previously operated data when processing former columns can be reused. Exploiting temporal locality for x vector is our major concern regarding data locality in sparse triangular solve.

2.7 Splitting Triangular Factor into Dense and Sparse Parts

We observed that the L factors obtained by Cholesky and LU factorization generally contain a dense submatrix in the lower right-most part of the L factor as mentioned in [8]. This dense part is referred as *trailing triangle* and accounts for a high fraction of the overall nonzeros. This structure can be exploited by splitting the L factor into sparse and dense components as in [8]. The triangular solve $Lx = b$ is then decomposed as:

$$\begin{bmatrix} L_1 & \\ L_2 & L_d \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix},$$

where L_1 is a sparse lower triangular submatrix, L_2 is a rectangular sparse submatrix and L_d is the dense trailing triangle. X_1 and X_2 (B_1 and B_2) are the components of x vector (b vector) according to the dimensions of the corresponding L factor components. With this decomposition, the triangular solve become equivalent to find the X_1 and X_2 subvectors by the following set of equations:

$$\begin{aligned} L_1 X_1 &= B_1, \\ L_2 X_1 + L_d X_2 &= B_2, \end{aligned}$$

which can be solved in three steps:

$$L_1 X_1 = B_1, \tag{2.13}$$

$$\tilde{B}_2 = B_2 - L_2 X_1, \tag{2.14}$$

$$L_d X_2 = \tilde{B}_2. \tag{2.15}$$

Here, equation 2.13 is a sparse triangular solve where Equation 2.14 is a sparse matrix-vector multiplication (SpMxV) and Equation 2.15 is a dense triangular solve. The process of splitting a lower triangular matrix into dense and sparse parts and solving them seperately can be further proceeded for the L_1 part. The second splitting on L_1 part is similarly performed by decomposing $L_1 X_1 = B_1$ as

$$\begin{bmatrix} L_{11} & \\ L_{12} & L_{1d} \end{bmatrix} \cdot \begin{bmatrix} X_{11} \\ X_{12} \end{bmatrix} = \begin{bmatrix} B_{11} \\ B_{12} \end{bmatrix}$$

which can be solved in three steps including one sparse triangular solve (SpTS), one sparse matrix vector multiplication (SpMxV) and one dense triangular solve as in the original splitting procedure.

2.8 Exploiting Cache Locality in SpMxV

Sparse matrix-vector multiplication (SpMxV) is another important kernel operation widely used in scientific computing. It requires to solve equations of the form $y = Ax$ where A is a given sparse matrix, x is a given dense vector and y is the dense solution vector. High performance gains are possible to be achieved in SpMxV operations if temporal and spatial localities can be exploited properly. A common method for exploiting cache locality in SpMxV operations is reordering the rows/ columns of A matrix.

Reordering the rows of A matrix such that rows with similar sparsity patterns are arranged nearby enables SpMxV to exploit temporal locality in accessing x-vector entries more. Reordering the columns of A matrix such that columns with similar sparsity patterns are arranged nearby enables SpMxV to exploit spatial locality in accessing x-vector entries more.

In a recent work [6], Akbudak et al handles such a reordering on the rows/ columns by representing A matrix as a hypergraph and partitioning it. It is shown that exploiting temporal locality is of primary importance in SpMxV operatons in terms of cache utilization. They utilize the column-net model of A matrix and construct a partition with the part sizes are upperbounded by the cache size. It is demonstrated that minimizing cutsize in this hypergraph partitioning model corresponds to reducing cache misses and enhancing the exploitation of temporal locality in accessing x-vector entries.

Chapter 3

Related Work

In literature there are several works for improving the performance of SpTS since it is an important kernel in scientific computing applications. Most of these works focus on the parallelization of SpTS to reduce runtimes in multicprocessors [24, 25, 26, 27, 28, 29].

A common method to parallelize SpTS is to construct the directed dependency graph based on the sparsity pattern of L factors and group the independent rows into levels to be processed by different processors [30, 31, 32]. The levels represent sets of row operations in SpTS which can be performed independently and they are obtained by a variant of BFS algorithm in general [31, 32]. In [33], directed graph representation is not used but a similar parallel algorithm for SpTS is developed which reorders the rows of the L factor by extracting independent rows concurrently and obeying the dependency rules among rows.

However parallel SpTS implementations have inherent limitations on performance due to the high communication rates with respect to the computation rates [27]. Hence on multiprocessors the forward and backward substitution steps may not perform its best efficiency and yield performance bottlenecks in applications that solve several systems with the same component matrix [29]. Still there is a limited number of studies proposed for uniprocessor implementations of SpTS.

Vuduc et al. propose a high-performance uniprocessor framework for auto-tuned SpTS in [8]. They introduce the splitting method upon the triangular factors into sparse and dense parts, which has been explained in Section 2.7 in detail. They also adopt the BCSR storage format and improve register reuse by register blocking optimization which was originally proposed by Im and Yelick [34]. Basically they select the block sizes for each matrix individually in a preprocessing step.

Another work [35] improves memory access and reduce cache misses by a trivial reorganization on the data structure in the matrix factorization phase. They simply store the L and U factors in the order of accessing by SpTS instead of the computation order from the LU factorization.

To the best of our knowledge, there has been no work using partitioning and reordering methods on triangular factors for cache utilization in SpTS. On the other hand, reordering techniques for cache utilization is widely used in other kernel operations like SpMxV [4, 5, 6].

In the most recent of these works [6], Akbudak et al. propose to use hypergraph partitioning models for reordering rows and columns in order to exploit cache locality in SpMxV. It is demonstrated that the hypergraph partitioning models precisely fulfill the requirements of the desired reordering methods for cache-aware SpMxV.

Chapter 4

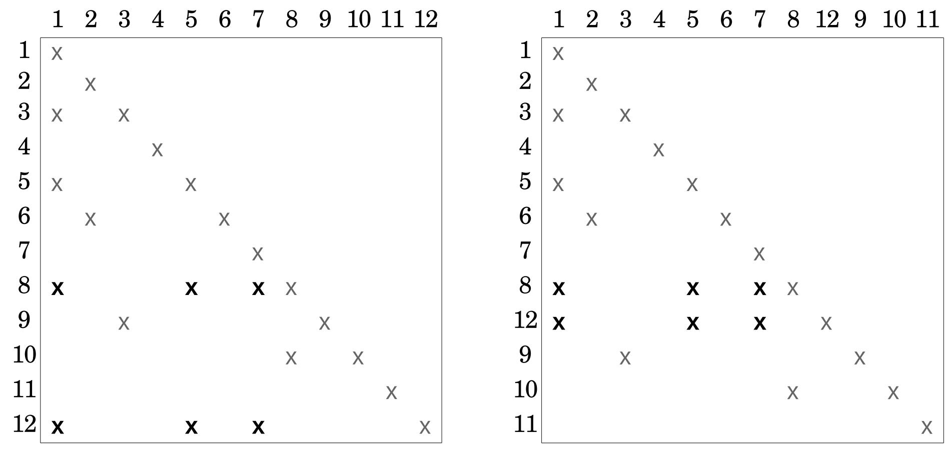
Directed Hypergraph Partitioning Model for Cache Aware Forward Substitution

We introduce the idea of reordering rows/ columns of the triangular matrices to exploit cache locality in Section 4.1. In Section 4.2, we show that partitioning of directed dependence graph does not fully correspond to partitioning problem of the lower triangular factor. The directed hypergraph representation of SpTS is explained in Section 4.3 and the ordered partitioning model on this directed hypergraph representation is represented in Section 4.4.

4.1 Reordering Triangular Matrix For Exploiting Cache Locality

The row-oriented forward substitution algorithm exploits the temporal locality in accesing previously computed x vector entries which were stored when processing previous rows. Hence for exploiting temporal locality, the rows using common x_i entries should be close enough to enable reusing these entries before they are evicted from the cache. In order to ensure this, we rearange the matrix so that the rows having similar sparsity pattern are ordered close to each other. Basically we determine the highly dependent (having large number of nonzeros on same columns) rows and apply a symmetric reordering (permuting rows and columns in the same order) to put these rows closer.

For instance, if we operate a row-oriented forward substitution on the L factor whose nonzero layout is given in Figure 4.1(a), the x_1, x_5 and x_7 entries are used for solving x_8. After completing the calculations for rows 9, 10 and 11, the x_1, x_5 and x_7 entries are again used for computing x_{12} because of the nonzeros $l_{12,1}$, $l_{12,5}$ and $l_{12,7}$. Assuming for this small example that the cache size is not enough to keep these x entries in the cache for reuse, we operate a symmetric reordering on the L matrix as in Figure 4.1(b) so that rows 8 and 12 become consecutive for exploiting temporal locality.

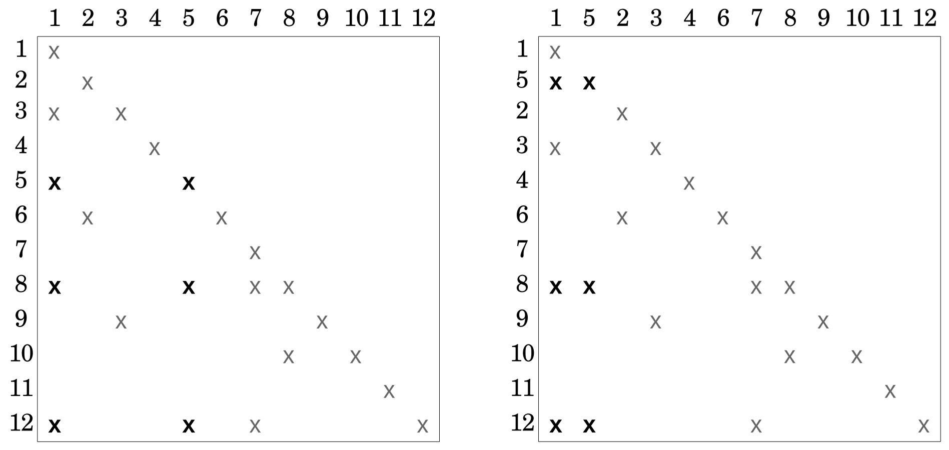

(a) The lower triangular matrix L

(b) L matrix after row-oriented symmetric reordering

Figure 4.1: Reordering rows of lower triangular factor L for exploiting temporal locality in row-oriented forward substitution

The column-oriented forward substitution algorithm exploits the temporal locality in accesing the x-vector entries which were previously calculated when processing previous columns. Therefore the columns using common x_i entries should be close enough to enable reusing these entries before evicted from the cache. In order to ensure this, we apply symmetric reordering to the matrix so that the columns having similar nonzero distribution become closer to fit into the cache.

For instance, if we operate a column-oriented forward substitution with the L factor in Figure 4.2(a), the x_5, x_8 and x_{12} entries are used for precessing column 1. After completing the calculations for columns 2, 3 and 4, the x_5, x_8 and x_{12} entries are again used for processing column 5. Assuming for this small example that the cache size is not enough to keep these x entries in the cache for reuse, we perform a symmetric reordering on the L matrix as illustrated in Figure 4.2(b) so that columns 1 and 5 become consecutive for exploiting temporal locality.



(a) The lower triangular matrix L

(b) L matrix after column-oriented symmetric reordering

Figure 4.2: Reordering columns of lower triangular factor L for exploiting temporal locality in column-oriented forward substitution

Figure 4.3(a) illustrates the situation if we move row 8 of the L factor in Figure 4.1(a) next to row 12 instead of moving row 12 next to row 8. The nonzero $l_{10,8}$ appearing in the upper triangular part destroys lower-triangularity and hence the process of forward substitution. The reason for this situation is that we broke the dependency rules by moving row 8 into 11^{th} position despite the fact that row 10 depends on row 8 because of the nonzero $l_{10,8}$.

A similar case occurs if we move column 1 of the L factor in Figure 4.2(a) next to column 5 instead of moving column 5 next to column 1 as illustrated in Figure 4.3(b). Here the nonzero $l_{3,1}$ appears in the upper triangular part because the dependency rules are broken by moving column 1 into 4th position despite the fact that column 3 depends on column 1 since $l_{3,1} > 0$.

$$\begin{array}{c|cccccccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 9 & 10 & 11 & 8 & 12 \\
\hline
1 & \mathrm{x} & & & & & & & & & & & \\
2 & & \mathrm{x} & & & & & & & & & & \\
3 & \mathrm{x} & & \mathrm{x} & & & & & & & & & \\
4 & & & & \mathrm{x} & & & & & & & & \\
5 & \mathrm{x} & & & & \mathrm{x} & & & & & & & \\
6 & & \mathrm{x} & & & & \mathrm{x} & & & & & & \\
7 & & & & & & & \mathrm{x} & & & & & \\
9 & & & \mathrm{x} & & & & & \mathrm{x} & & & & \\
10 & & & & & & & & & \mathrm{x} & & \mathrm{x} & \\
11 & & & & & & & & & & \mathrm{x} & & \\
8 & \mathbf{x} & & & & \mathbf{x} & & \mathbf{x} & & & & \mathrm{x} & \\
12 & \mathbf{x} & & & & \mathbf{x} & & \mathbf{x} & & & & & \mathrm{x} \\
\hline
\end{array}$$

(a) Reordering rows of lower triangular matrix L without considering row dependencies

$$\begin{array}{c|cccccccccccc|}
 & 2 & 3 & 4 & 1 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
2 & \mathrm{x} & & & & & & & & & & & \\
3 & & \mathrm{x} & & \mathrm{x} & & & & & & & & \\
4 & & & \mathrm{x} & & & & & & & & & \\
1 & & & & \mathrm{x} & & & & & & & & \\
5 & & & & \mathbf{x} & \mathbf{x} & & & & & & & \\
6 & \mathrm{x} & & & & & \mathrm{x} & & & & & & \\
7 & & & & & & & \mathrm{x} & & & & & \\
8 & & & & \mathbf{x} & \mathbf{x} & & \mathrm{x} & \mathrm{x} & & & & \\
9 & & \mathrm{x} & & & & & & & \mathrm{x} & & & \\
10 & & & & & & & & \mathrm{x} & & \mathrm{x} & & \\
11 & & & & & & & & & & & \mathrm{x} & \\
12 & & & & \mathbf{x} & \mathbf{x} & & \mathrm{x} & & & & & \mathrm{x} \\
\hline
\end{array}$$

(b) Reordering columns of lower triangular matrix L without considering column dependencies

Figure 4.3: Reordering rows/columns without obeying dependency rules would destroy lower-triangularity

Proposition 4.1.1. *Obeying dependency rules while doing symmetric reordering is the necessary and sufficient condition for preserving lower triangularity.*

Proof. We explained the necessity part in Section 2.4. For proving the sufficiency part, let us assume that lower triangularity is destroyed despite of applying symmetric reordering by obeying dependency rules. Let the nonzero $l_{i,j}$ be placed on the upper triangular part after reordering to the position (i', j'). Since $l_{i,j}$ is a nonzero before reordering, we know that $i > j$ and row/column i depends on row/column j. We also know that $i' < j'$ since position (i', j') stay on the upper triangular part. This means that row/column i is positioned before row /column j by reordering. Since we know that row/column i depends on row/column j, this fact contradicts with our assumption of obeying dependency rules and completes the proof. $\square$

Essentially we do not need to place highly dependent rows consecutively and may even not succeed this hard task every time because of the data dependencies. Theorically, it is enough to place them close enough to exploit temporal locality. Hence we adopt the idea of reordering and partitioning the rows/columns of the triangular factor into parts so that

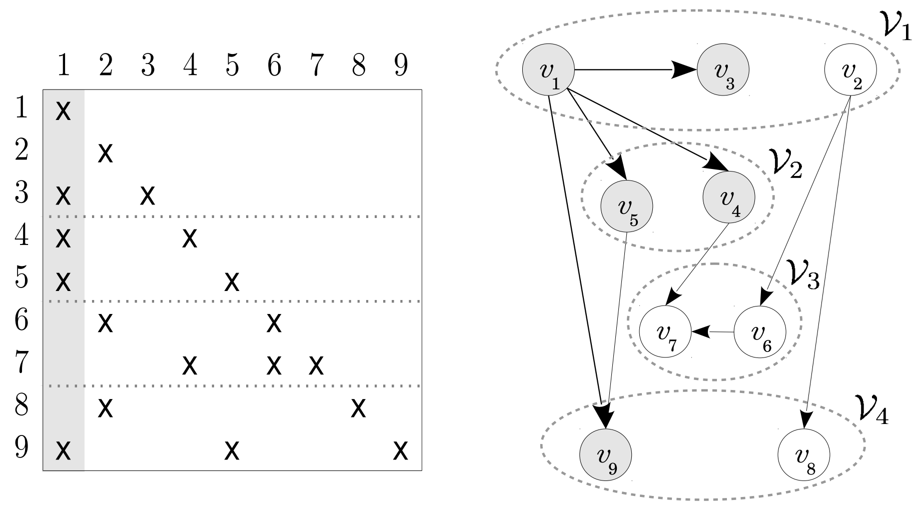
1. Each part fits into the cache, that is to say the size of the memory required to store the submatrix and the subvectors associated with a part is less than the size of the cache.
2. The data requirements between different parts is minimum, i.e. the number of cache misses induced by the need of accessing data which was previously used in another part is minimum.
3. Part elements obey the data dependency rules, i.e. a part cannot include a row (column) which depends on a row (column) from a subsequent part.

In our model we assume that the cache is fully associative, i.e. any data in the main memory can be stored in any cache location. By this assumption, we do not consider the conflict misses and only take the capacity misses into account.

The first property of this L-factor partitioning means that there will be no cache misses during processing the rows/ columns of the same part if the cache is fully associative. Hence the total number of cache misses is nothing more than the number of cache misses caused by the shared data between different parts. Therefore obeying the second property yields minimizing the total number of cache misses. This property enhances the probability of highly dependent rows / columns belonging to the same part.

4.2 Deficiencies of Directed Graph Partitioning Model for Reordering Triangular Matrices

In literature, dependence graph representation of lower triangular factor is used to reorder and levelize the rows for parallel implementations [31], [32]. First thing come to mind is that this representation can also be utilized for reordering the rows/ columns of the lower triangular factor by means of graph partitioning tools. We can partition the dependence graph into parts each having weights less than the cache size, and aim to minimize the cutsize which hopefully relates to minimize cache misses.



(a) Partitioning the lower triangular matrix L

(b) Partitioning dependence graph of lower triangular matrix L

Figure 4.4: Partitioning the L factor and corresponding dependence graph

However the cutsize definition of a standard dependence graph representation does not truely encode the cache miss counts. Consider the sample triangular matrix L whose rows are partitioned into 4 parts as shown in the Figure 4.4(a). We assume that each part fits into cache exactly, meaning that adding one more row to a part would end up with no longer fitting into cache. The corresponding dependence graph is constucted and partitioned as in Figure 4.4(b). The initial cut edges are e_{14}, e_{15}, e_{19}, e_{26}, e_{28} and e_{47}. If we assume that the cost of each edge is assigned to 1 as a standard graph partitioning setup, then the cutsize will be 6.

Let us consider the traffic of the x_1 entry in the cache. First, x_1 is calculated by processing row 1. Then row 3 directly uses x_1 value from the cache since row 1 and 3 belong to the same part which fits into the cache by construction. However row 4 has to fetch x_1 from main memory since it is no longer in cache. Then row 5 can directly get the x_1 value from cache since rows 4 and 5 belong to the same part. Finally row 9 needs x_1 value but a cache miss occurs since x_1 is ejected from the cache until reaching row 9 from row 5. Therefore there are 2 cache misses induced by data x_1 in total, for processing rows 4 and 9. However, the cutsize arised from vertex v_1 of the dependence graph corresponding to the L factor is 3, namely because of the cut edges e_{14}, e_{15} and e_{19}. The reason for this incompability is that the costs of vertices v_4 and v_5 are counted separately in the graph partitioning representation for calculating the cutsize although they were needed to be counted once in total.

We see that the graph representation is not adequate for truely encoding the cache-aware forward substitution problem. Yet we propose a novel directed hypergraph partitioning model which is precisely equivalent to lower triangular partitioning problem as described in Section 4.1.

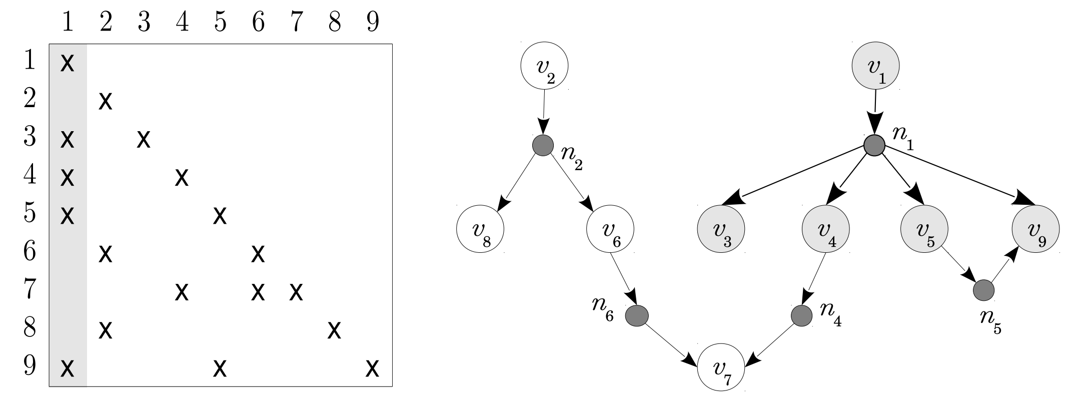
4.3 Directed Hypergraph Representation of Forward Substitution

Since we see that the graph representation is not adequate for encoding the forward substitution to minimize the cache misses, we propose a directed hypergraph model to encode this problem. We adopt a column-net directed hypergraph model for row-oriented forward substitution, and a row-net directed hypergraph model for column-oriented forward substion. The column-net hypergraph model takes its name by the property of the model which assigns the columns of a matrix to the nets of the hypergraph. The row-net hypergraph model is called so beacuse the rows of the matrix are represented by the nets of hypergraph.

For row-oriented forward substitution, we use a column-net directed hypergraph model, assigning the rows of the L factor to the vertices of the hypergraph since we want to permute rows for exploiting temporal locality. The nets are represented by columns of the L factor and net n_j consists of the vertices which correspond to the rows having a nonzero on column j. The directions on hyperedges are defined as follows: If row i depends on row j, then we say that the net n including v_i and v_j involve a direction from v_j to v_i. For the ease of presentation, we put an arrow going from v_j to net n and an arrow going from net n to v_i. Now let us show that the directions of these arrows on a net does not conflict.

Consider net n_j, corresponding to column j. For each nonzero $l_{i,j}$ on column j, row i depends on row j, hence net n_j involves a direction from v_j to v_i. Therefore net n_j involves directions from vertex v_j to all other vertices belonging to the net n_j. Then the direction representation is straightforward: There is an arrow going from v_j to net n_j and for each other vertex v_i in net n_j, there is an arrow from n_j to v_i. Since such v_j vertex is unique to the net n_j, we say that v_j is the unique *source node* of net n_j, and all the other vertices belonging to net n_j are called the *sink* nodes of net n_j. We denote the source node of a net n_j as $src(n_j)$ for this column-net directed hypergraph model.

Figure 4.5(b) illustrates the column-net directed hypergraph representation of the L factor given in Figure 4.5(a). The net n_1 and corresponding column 1 of the L factor are highlighted. Both rows 3, 4, 5 and 9 depend on row 1 because of the nonzeros $l_{3,1}$, $l_{4,1}$, $l_{5,1}$ and $l_{9,1}$. Hence net n_1 has vertex v_1 as a source node, and the vertices v_3, v_4, v_5, v_9 as sink nodes.



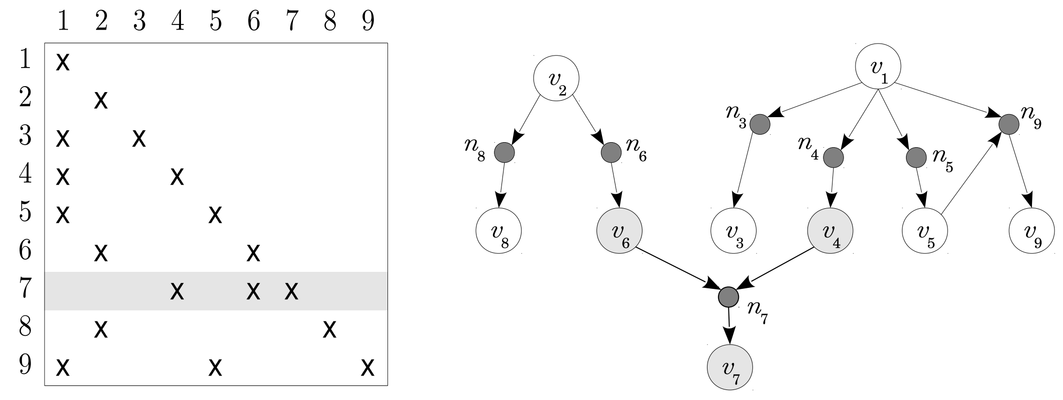
(a) The lower triangular factor L (b) Column-net hypergraph representation of L factor

Figure 4.5: Column-net directed hypergraph representation of forward substitution

A similar representation is adopted for column-oriented forward substitution, namely the row-net directed hypergraph model, in which the vertices of the hypergraph represent the columns of the L factor since we want to permute columns for exploiting temporal locality. The nets represent the rows of L factor and net n_j consists of the vertices which correspond to the columns having a nonzero on row j. The directions on hyperedges are defined the same as in the column-net model. However in this case we have multiple source nodes and a unique sink node for each net.

Consider net n_i, corresponding to row i. For each nonzero $l_{i,j}$ lying on row i, column i depends on column j, hence net n_i involves a direction from v_j to v_i. Therefore net n_i involves directions to vertex v_i from all other vertices in $Pins(n_i)$. Hence the direction representation is straightforward: There is an arrow going from net n_i to node v_i and there is an arrow from v_j to n_i for each $v_j \in Pins(n_i)$ where $i \neq j$. Vertex v_i is called the unique sink node of net n_i, and all the other vertices belonging to net n_i are called the source nodes of net n_j in the case of row-net directed hypergraph model.

Figure 4.6(b) illustrates the row-net directed hypergraph representation of the L factor given in Figure 4.6(a). The net n_7 and corresponding row 7 of the L factor are highlighted. Because of the nonzeros $l_{7,4}$ and $l_{7,6}$, column 7 depends on both columns 4 and 6 . Hence net n_7 has vertex v_7 as a sink node, and the vertices v_4 and v_6 as source nodes.



(a) The lower triangular factor L (b) Row-net hypergraph representation of L factor

Figure 4.6: Row-net directed hypergraph representation of forward substitution

4.4 Directed Hypergraph Partitioning (dHP)

In this section we introduce an ordered directed hypergraph partitioning (dHP) model which meets the requirements of the triangular solve partitioning problem. We define an ordered partition on the directed hypergraph to reorder the vertices and hence to permute the corresponding rows/columns of the L factor.

For row-oriented (column-oriented) forward substitution, we define the weight of vertex v_i as the number of nonzeros in row (column) i. We assign 1 to the cost of each net. Our aim is to obtain an ordered partition $\Pi = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$ on the set of vertices so that:

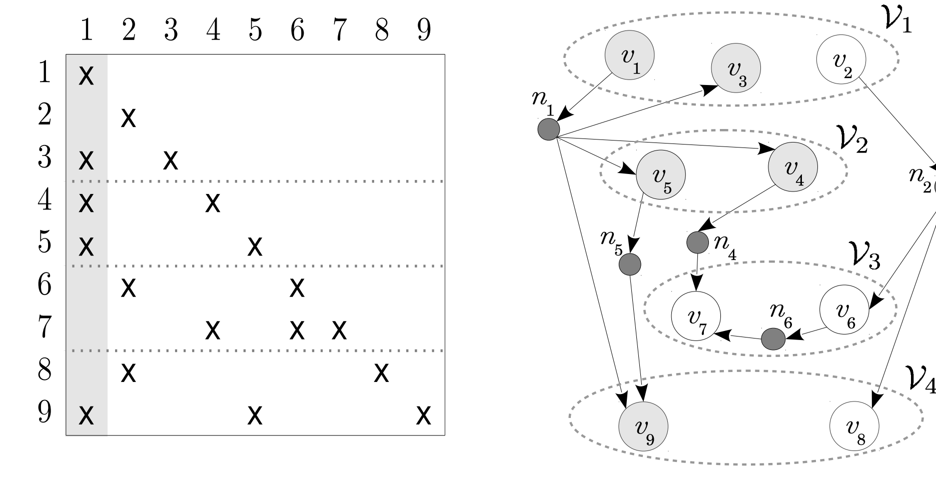
1. The weight of each part $\mathcal{V}_i$ is less than the cache size.
2. The cutsize of Π is minimum.
3. If a source of a net n belong to part $\mathcal{V}_i$ and a sink node of n belongs to part $\mathcal{V}_j$, then $i < j$ must hold.

Let us show that such an ordered partition on the vertices of the directed hypergraph induces a feasible reordering on the rows/ columns of the lower triangular factor. Note that these three properties of the aimed directed hypergraph partitioning has one-to-one correspondence with the properties of the desired lower triangular matrix partitioning given in Section 4.1.

Since we assign the number of nonzeros in rows/ columns to the weight of vertices, the weights of the parts in Π correspond to the sizes of the parts in L factor. Therefore making the part weights in Π less than the cache size is equivalent to fitting the parts of L factor into the cache. Hence obeying the first property of the ordered directed hypergraph partitioning induces to fulfill the first property of L factor partitioning defined in Section 4.1.

Minimizing cutsize of Π is equivalent to minimizing the sum of connectivities of cut nets. The connectivity of a net corresponds to the number of parts that the nonzeros of the corresponding row/column belongs to. It is actually the number of cache misses induced by the data requirements of a row / column between different parts if we assume the cache is fully associative. Hence minimizing cutsize of Π corresponds to minimize the number of cache misses arised between different parts, that is to obey the second property of L factor partitioning.

The 3rd property means that the parts should be aligned in the increasing order of their indices by obeying the dependency rules: A sink node of a net must come after a source node, and so do the parts where they belong. Let us denote the part where vertex v_i belongs to as $part(v_i)$, and the index (order) of a part $\mathcal{P}$ as $idx(\mathcal{P})$. If a net has a source node v_s and a sink node v_t, then $part(v_s)$ should come before $part(v_t)$, meaning that $idx(part(v_s)) < idx(part(v_t))$.By this setup we ensure that there exist no net which involves a direction from a vertex in $\mathcal{V}_j$ to a vertex in $\mathcal{V}_i$ whenever $i < j$. Thus this property is directly related to obey the data dependency rules represented for the L factor partitioning model.



(a) Rowwise partitioning the lower triangular matrix L

(b) Partitioning the column-net directed hypergraph corresponding to lower triangular matrix L

Figure 4.7: Rowwise partitioning the L factor and the corresponding column-net directed hypergraph

Figure 4.7(b) illustrates the column-net directed hypergraph partitioning corresponding to the initial rowwise partitioning of the L factor given in Figure 4.7(a). The connectivity set of net n_1 is $\{\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3\}$ and the connectivity of n_1 is $\lambda_1 = 3$. Therefore, net n_1 contributes $\lambda_1 - 1 = 2$ to the cutsize, which is equal to the number of cache misses induced by entry x_1. This equivalence confirms the applicability and validity of the directed hypergraph partitioning model for the triangular solve reordering problem defined in Section 4.1.

We refer the column-net directed hypergraph partitioning model proposed for exploiting temporal locality in row-oriented forward substitution as *row-wise dHP* since we define a partition on the rows of the corresponding triangular matrix. Similarly we refer the row-net directed hypergraph partitioning model proposed for exploiting temporal locality in column-oriented forward substitution as *column-wise dHP* since we define a partition on the columns of the corresponding triangular matrix.

4.4.1 Recursive Bipartitioning

We solved this K-way partitioning problem by doing recursive bipartitioning until the weight of each part become less than the cachesize. After each bipartitioning step, we apply a variant of FM algorithm which we develop for ordered directed hypergraph partitioning.

Since we are applying recursive bipartitioning, we only consider two parts at each step which we refer to as *left* and *right* parts, and denoted as $\mathcal{V}_L$ and $\mathcal{V}_R$, respectively. We denote the number of pins that net n_i has in left part as $Left(n_i)$ and the number of pins that n_i has in right part as $Right(n_i)$.

We assume that the initial partitioning of the vertices obeys the dependency rules, i.e. the arrows between two parts always have directions coming from left part and going to the right part. We preserve this rule by applying a modified FM algorithm that only allows the moves which respect these direction constraints. Note that the initial vertex order corresponding to the original row/ column order is feasible since a net can include a direction from vertex v_i to v_j only if $i < j$. Hence sorting the vertices with increasing order of indices and dividing from any point to two parts yields a partition obeying the dependency rules.

We can handle this initial partition trivially by just splitting the vertex sequence from the middle, meaning that the weights of each part is approximately equal. Instead, for a better initial bipartitioning, we search the γ neighbourhood of the middle point and split the sequence from the point which yields a minimum initial cutsize. In other words, we start with the first vertex, put it to the left part, and continue this procedure on vertices one by one until the weight of the left part is greater than $(50 - \gamma)\%$ of the total vertex weight. After this point we continue this procedure also by calculating the current initial cutsize and stop when the weight of the left part reaches $(50 + \gamma)\%$ of the total vertex weights. Among the interval that we calculate the initial cutsizes, we determine the vertex index which corresponds to a bipartition with lowest initial cutsize. Then we update the part informations of vertices and weight informations of parts according to the bipartitioning which splits the hypergraph from that point.

4.4.1.1 Cut-net Splitting

After completing the initial bipartitioning and our directed FM procedure, we construct two new hypergraphs corresponding to the right and left parts of the bipartition. We proceed these recursive bipartitioning steps onto these two hypergraphs that we obtain. At this new hypergraph construction step, we duplicate the nets which have nodes in both parts, i.e. the cut nets. This procedure is called *cut-net splitting* and exploited for capturing the partitioning cutsizes accurately.

After each bipartition, we construct two sub-hypergraphs $\mathcal{H}_L$ and $\mathcal{H}_R$ corresponding to the left and right parts of the bipartitioning respectively. The vertex set of $\mathcal{H}_L$ ($\mathcal{H}_R$) is equivalent to the vertex set of $\mathcal{V}_L$ ($\mathcal{V}_R$). The internal nets of left (right) part are added to the net set of $\mathcal{H}_L$ ($\mathcal{H}_R$). We split each cut net n_i to two nets n_i^L and n_i^R where $Pins(n_i^L) = Pins(n_i) \cap \mathcal{V}_L$ and $Pins(n_i^R) = Pins(n_i) \cap \mathcal{V}_R$.

Note that some nets may not possess a source node or any sink node after this splitting process. For instance assume that the source node v_1 of net n_1 remains in left part while the sink nodes v_2 and v_3 stay in right part after completing the bipartitioning and the directed FM procedure. Then in the left hypergraph $\mathcal{H}_L$ we have net n_1^L having the source node v_1 as its only pin, while in the right hypergraph $\mathcal{H}_R$ we have net n_1^R having the sink nodes v_2 and v_3. In this sample we observe that the net n_1^L has no sink nodes in the hypergraph $\mathcal{H}_L$ and the net n_1^R has no source node in the hypergraph $\mathcal{H}_R$. We handle such cases in our modified FM algorithm.

If we consider the dependencies of the row-net directed hypergraph model oppositely and reverse the directions of the arrows, we obtain nets having a single source and multiple pins just like our column-net directed hypergraph model. Hence we can apply our partitioning methods proposed for the column-net directed hypergraph model to the row-net directed hypergraph model with only a few alterations. For this reason, we assume that the operations are held on the column-net directed hypergraph model for the following algorithms without loss of generality.

4.4.2 Directed Fiduccia-Mattheyses (dFM) Algorithm

For reordering the vertices in order to reduce the cutsize, we developed a variant of FM algorithm which works for directed hypergraph model by taking the dependency rules into account. We refer this modified FM algorithm as directed FM, namely dFM algorithm.

As in the usual FM algorithm, we calculate the initial gains of each vertex at the beginning of each bisection for chosing the vertex whose move to other part reduce the cutsize most. For deciding to move the vertex with highest gain (namely the *base vertex*), usual FM algorithm only checks the balance constaint given in equation 2.10. However for directed hypergraph model we have two additional feasibility constraints:

1. If the base vertex is the source of net n, and if there exist sink nodes of n in the left part, then the base vertex cannot be moved from left to right part.
2. If the base vertex is a sink node of a net which has the source node in the right part, then the base vertex cannot be moved from right to left part.

If the chosen base vertex yields an unfeasible move, we discard moving this vertex and consider the vertex with the next highest gain value. In order to determine the vertex with highest gain, we use a max-heap assigning the gains of vertices as keys.

The pseudocode of the method $IsFeasible$ which checks the feasibility of moving vertex v to other part is shown in Algorithm 3. This procedure returns 1 if moving the corresponding vertex to other part is feasible in terms of dependency rules and 0 otherwise. We consider each net that a given vertex belongs and check if the dependency rules associated with that net are violated in the case of moving the given vertex. We discussed how a net may not have a source vertex in the previous section. The nets which have no source node anymore cannot force the feasibilty constraint and hence are skipped while searching the nets including the given vertex. There are two cases forcing the dependency rules:

1. If the given vertex belongs to the left part and it is the source node of a net that has other vertices in the left part, or
2. If the given vertex belongs to right part and it is a sink node of a net that has its source node in the right part

then we set moving the given vertex to other part as unfeasible and the procedure $IsFeasible$ returns 0, otherwise it returns 1.

Algorithm 3 Checking The Feasibility Constraints

```
1:  procedure IsFeasible($v$)
2:      for each $n \in Nets(v)$ do
3:          if $n$ has a source node then
4:              $s \leftarrow src(n)$
5:              if $v = s$ then
6:                  if $Part(v) = Left$ and $Left(n) > 1$ then
7:                      return 0
8:              else
9:                  if $Part(v) = Right$ and $Part(s) = Right$ then
10:                     return 0
11:     return 1
```

Before each move, we extract the vertex with highest gain from the heap and check its feasibility of moving regarding to the dependency constraints and the balance constraint. If moving this vertex is infeasible because of the balance constraint, we lock this vertex and do not bring it back to the heap. This attitude is applicable and does not restrict the solution space much bacause we observed that the proportion of unfeasible moves due to the balance constraint is highly low compared to the unfeasible moves due to the dependency violation. Morever, the feasibility conditions regarding the dependency constraints frequently change while the vertices moving and altering the node positions of nets. Hence we need a method to reinsert the previously extracted vertices into the heap if they previously extracted from the heap beacuse of the infeasibility regarding dependency constraints but become feasible after a certain move. For this purpose, we check the vertices whose feasibility condition might change from infeasible to feasible and update the max-heap after each move.

We do not update the heap for the vertices turning from feasible to infeasible after each move because we check the feasibility of the base vertex before each move and if it is infeasible we pass to the next element in the heap. Since we handle the detection of vertices which are infeasible to move before moving the base vertex, we do not need to update the heap for evicting the infeasible ones. Hence we do not force the heap to consist of only feasible vertices, but we guarantee that all the feasible vertices belong to the heap in each move.

Algorithm 4 Updating Max-Heap After Moving Base Vertex

```
1: procedure UpdateHeap($base$)
2:     for each $n \in Nets(base)$ do
3:         if $n$ has a source node then
4:             $s \leftarrow src(n)$
5:             if $base = s$ then
6:                 if $Part(base) = Left$ then
7:                     for each $v \in Pins(n)$ do
8:                         if $v$ is unlocked and $v$ is not in heap then
9:                             if IsFeasible ($v$) then
10:                                insert $v$ to heap
11:            else if $Part(s) = Left$ and $Left(n) = 1$ then
12:                if $s$ is unlocked and $s$ is not in heap then
13:                    if IsFeasible ($s$) then
14:                        insert $s$ to heap
```

Algorithm 4 shows how the max-heap is updated after a move in our dFM algorithm. After each move we consider the vertices whose feasibility condition might change. In general it is enough to check the vertices which share at least one net with the moved (base) vertex. We further restrict our search of the candidate vertices which are possible to turn into feasible condition from infeasible condition. There are two cases that moving a vertex may turn from infeasible to feasible after the move of the base vertex:

1. If base vertex is the source node of net n, and it moved from right part to left part, then the other vertices in $Pins(n)$ which were previosly infeasible might become feasible to move.

2. If base vertex is a sink node of net n which has its source node in the left

part and does not have any other nodes in that part, then the source node of n become feasible to move with respect to the net n.

We continue moving the vertices with highest gain if they are feasible until the heap is empty or no feasible move remains. Then we find the point when the cutsize is minimum and construct the left and right sub-hypregraphs corresponding to the vertex distribution at that point. This set of procedures is called a pass of FM algorithm and we apply these passes until the total cutsize gains drop below zero.

4.4.3 Clustering in dHP

We also include a clustering method in our dFM algorithm to ensure placing the highly-dependent rows/columns successively. We construct supernodes for the initial hypergraph and we make the uncoursening after completing all bipartitioning steps and obtaining the final permutation vector.

Basically we determine the vertices having almost the same set of nets with at most one exception. We connect two vertices v_i and v_j if the sets $Nets(v_i)$ and $Nets(v_j)$ are same except the nets n_i and n_j for the initial hypergraph, i.e. if $Nets(v_i) \cup \{n_i, n_j\} = Nets(v_j) \cup \{n_i, n_j\}$.

Because of the dependency constraints, we cannot merge each pair of vertices satisfying the above condition. For instance, assume that vertices v_i and v_j $(i < j)$ have almost the same set of nets but there exists another vertex v_k with $i < k < j$ such that v_k depends on vertex v_i and v_j depends on v_k. Then v_i and v_j can belong to the same supernode only if v_k is in that supernode too, beacuse otherwise we encounter with a case that the supernode containing v_i and v_j both depends on and is depended by the vertex v_k. In order to avoid such cases that force the dependency constraints, we only allow consecutive vertices to form a supernode.

For each vertex, we consider its adjacent vertex having the next index for checking our clustering criteria. If they satisfy the condition to be connected, then we embed them in a supernode. We continue this way and add more subsequent vertices to the same supernode until the next vertex does not satisfy the condition to be clustered with the current vertex.

A coarsened hypergraph is constructed such that disjoint subsets of vertices of the original hypergraph which are coalesced into supernodes form a single vertex of this new hypergraph. The weight of a vertex in the coarsened hypergraph is determined as the sum of the weights of the vertices that constitute the respective supernode in the original hypergraph. To put it more formally, let the vertices $v_i, v_{i+1}, \dots v_j$ of the original hypergraph $\mathcal{H}$ form a supernode which corresponds to the vertex v'_{i-j} in the coarsened hypergraph $\mathcal{H}'$. Then we set $w(v'_{i-j}) = \sum_{k=i}^{j} w(v_k)$. Similarly the net set of vertex v'_{i-j} in $\mathcal{H}'$ is equalized to be the union of the net sets of the constituent vertices $v_i, v_{i+1}, \dots v_j$ of $\mathcal{H}$, i.e. $Nets(v'_{i-j}) = \bigcup_{k=i}^{j} Nets(v_k)$.

For the uncoarsening, we project the final partition found on $\mathcal{H}'$ back to a partition on the original hypergraph $\mathcal{H}$. We assign each vertex in $\mathcal{H}$ forming a supernode to the part of the corresponding vertex in $\mathcal{H}'$.

In row-net (column-net) directed hypergraph model context, the vertices v_i and v_j with almost the same set of nets corresponds to the columns (rows) i and j having nonzeros in the same rows except the i^{th} and j^{th} rows. In other words, columns (rows) i and $i+1$ are coalesced when $l_{k,i} \neq 0$ holds if and only if $l_{k,i+1} \neq 0$ for each $k > i+1$, whether $l_{i,i+1}$ is nonzero or not. Hence in row-net (column-net) directed hypergraph model, clustering stands for combining columns (rows) with similar sparsity patterns.

Chapter 5

Experimental Results

Throughout the previous chapters, we examine the methods for exploiting data locality in triangular solve by partitioning and reordering the L factor. In this chapter, we present the cutsize and runtime improvements on real-world datasets obtained by the proposed model and framework.

5.1 Experimental Setup

The proposed directed hypergraph partitioning model is implemented and tested by comparing the performance of SpTS on the reordered matrices with the one on the original matrices. We set the imbalance ratio ϵ to 0.2 in hypergraph bipartitioning which means that a part weight cannot exceed 60% of the total vertex weight. We determine the γ neighbourhood to be 10%, meaning that we search the point to bipartite the hypergraph by ensuring the initial part weights remain between 40% and 60%.

Thoughout testing our directed hypergraph partitioning model, we observe that the mobility of vertices are restricted by the high dependencies between vertices and high vertex weights. This problem is directly related to the density of the corresponding L factor. Our model works better for sparser lower triangular matrices and its performance is effected by the dense parts having high number of nonzeros which arise in the L factor.

For fully exploiting the advantages of compressed storage formats and the reordering methods for cache utilization, we need to work on real sparse matrices. However we observe that most of the L factors obtained by LU or Cholesky factorization have a dense part in the lower rightmost side of the lower triangle. To exploit this structure and obtain a higher flexibility of movements in our directed hypergraph partitioning model, we adopt the idea of splitting the L factor into dense and sparse parts as proposed in [8].

To determine the point where to divide the L factor, namely the *switch point*, Vuduc et al. starts from the diagonal element of the last row and scans only the last row of the L factor until reaching two consecutive zero elements [8]. On the other hand, we do not restrict our search to just one row but we select the switch point by considering the density of the lower rightmost dense part i.e. the trailing triangle. We start from the last column index and proceed backwards until the submatrix staying rightside of the current column has a density below a threshold δ and the subarray containing the current column along with the previous and next columns has density below a threshold σ. The *density* of a submatrix is defined as the fraction of the submatix entries occupied by nonzero elements. The first condition provides the high density of the trailing triangle while the latter accounts for coupling highly connected matrix sections together. We set $\delta = 0.3$ and $\sigma = 0.1$ as the most appropriate thresholds for determining the switch point by investigating the sparsity patterns of various lower triangular matrices.

The idea of splitting the lower triangular matrix further upon the sparse lower-triangular submatrix is suggested in [8] but was not employed and experimentalized in that work. We conduct experiments for examining the effectiveness of our directed hypergraph partitioning model on both the original L factor, the L_1 part obtained by first splitting and the L_{11} part obtained by second splitting performed on L_1.

We developed a framework for cache utilization in sparse triangular solve including the matrix splitting method and directed hypergraph partitioning-based reordering on lower triangular matrices obtained by this splitting. We also apply the reordering method proposed in [6] on rectangular matrices obtained by splitting procedure in order to exploit cache locality for SpMxV which is used for solving SpTS.

We exploit autotuning provided by OSKI to reduce performance degradation which incurs due to the gap between processors and memory speeds. For storing the matrices, block compressed storage schemes are used since they outperform usual compressed storage schemes for sparse triangular solve [34]. We apply BCSR scheme for row-oriented forward substitution and BCSC scheme for column-oriented forward substitution for exploiting spatial locality in accessing L factor and b vector entries.

The experiments are conducted on a dual-core CPU with 2 MB cache size. The cache line size is of size 8 doubles and the set-associativity is 8. The data type used for storing the matrices is double precision floiting point number of size 8 bytes except the index array using integers of size 4 bytes. The implementations are done in C programming language and the files are compiled with the gcc -O3 optimization flag enabled.

5.2 Datasets

Various matrices are collected from the University of Florida Sparse Matrix Collection [36] to test the effectiveness of the proposed framework. Upon each matrix, the Cholesky factorization is carried by the CHOLMOD package included in SuiteSparse Collection.

The number of rows and nonzeros of the generated L factors are given in Table 5.1. The densities of L factors in percentage terms and switch point values which are equal to the row count of L_d parts are also shown. We present the densities of the constructed L_1, L_2 and L_d parts and the fraction of their nonzero counts over the total number of nonzeros of L factors in percentage terms. Similarly the switch point of second splitting (the dimension of L_{1d} part) along with the density and fraction of nonzero informations for the submatrices obtained by second splitting are given in Table 5.2.

	L factor				L_1 part		L_2 part		L_d part	
Matrix Name	Row Cnt	Nnz Cnt	Density	Switch Point	Density	Total Nnz %	Density	Total Nnz %	Density	Total Nnz %
aft01	8,205	355,870	1.1	446	0.8	68.3	2.0	19.0	45.2	12.7
bcsstk17	10,974	1,124,822	1.9	816	1.5	69.1	1.9	14.3	55.8	16.5
bcsstk18	11,948	795,066	1.1	5	-	-	-	-	-	-
bcsstk25	15,439	1,657,470	1.4	888	1.2	75.6	1.6	12.5	49.9	11.9
bcsstk28	4,410	359,209	3.7	444	3.2	70.4	3.2	15.9	49.7	13.7
bcsstk38	8,032	805,686	2.5	722	1.9	61.5	2.8	18.2	62.5	20.2
bodyy4	17,546	762,129	0.5	555	0.4	69.9	1.5	18.1	59.3	12.0
bodyy6	19,366	883,911	0.5	564	0.4	70.9	1.5	18.4	59.5	10.7
bundle1	10,581	464,166	0.8	460	0.2	19.5	6.3	62.9	76.8	17.6
crystm01	4,875	379,944	3.2	397	3.0	80.2	2.0	9.5	49.6	10.3
Dubcova1	16,129	735,850	0.6	283	0.5	84.5	1.8	11.1	80.5	4.4
fv1	9,604	397,261	0.9	495	0.6	62.8	1.9	21.7	50.2	15.5
fv2	9,801	406,470	0.8	505	0.6	62.8	1.9	21.6	49.7	15.6
gyro	17,361	1,326,348	0.9	547	0.8	88.0	0.9	6.4	49.7	5.6
ins2	309,412	1,585,753	0.0	31	0.0	54.0	7.6	46.0	37.7	0.0
jnlbrng1	40,000	1,603,752	0.2	652	0.2	80.7	0.8	12.6	50.7	6.7
minsurfo	40,806	1,593,385	0.2	842	0.1	72.4	0.8	16.5	50.0	11.1
msc04515	4,515	254,785	2.5	411	1.9	63.6	3.0	19.9	49.7	16.5
Muu	7,102	239,912	1.0	205	0.8	84.3	1.7	9.9	65.8	5.8
nasa4704	4,704	320,195	2.9	545	1.5	40.5	4.5	31.8	59.7	27.7
obstclae	40,000	1,558,505	0.2	800	0.1	71.7	0.8	17.1	54.6	11.2
s1rmq4m1	5,489	663,434	4.4	719	3.3	57.2	4.5	23.3	49.8	19.4
s1rmt3m1	5,489	559,367	3.7	703	2.5	51.4	4.4	26.6	49.8	22.0
s2rmq4m1	5,489	663,434	4.4	719	3.3	57.2	4.5	23.3	49.8	19.4
s2rmt3m1	5,489	559,367	3.7	703	2.5	51.4	4.4	26.6	49.8	22.0
s3rmq4m1	5,489	663,434	4.4	719	3.3	57.2	4.5	23.3	49.8	19.4
s3rmt3m1	5,489	559,367	3.7	703	2.5	51.4	4.4	26.6	49.8	22.0
s3rmt3m3	5,357	447,179	3.1	633	2.2	53.9	3.5	23.7	49.8	22.4
sts4098	4,098	181,808	2.2	218	1.9	77.4	3.1	14.4	63.2	8.3
t2dah_e	11,445	586,530	0.9	518	0.7	71.4	1.8	17.1	49.8	11.4
torsion1	40,000	1,558,505	0.2	800	0.1	71.7	0.8	17.1	54.6	11.2
AVERAGE	22,730	775,900	1.9	550	1.4	65.0	2.8	20.8	54.1	14.1

Table 5.1: Sparsities of submatrices obtained by 1. splitting

Note that for some matrices, the dimension of the trailing triangle is too low to alter the performance results. For the matrices with the switch point value below 30, we do not perform splitting procedure. Hence the properties and performance of the corresponding submatrices are not evaluated for such matrices.

	L_1 part				L_{11} part		L_{12} part		L_{1d} part	
Matrix Name	Row Cnt	Nnz Cnt	Density	Switch Point	Density	Total Nnz %	Density	Total Nnz %	Density	Total Nnz %
aft01	7,759	243,089	0.8	158	0.8	97.3	0.2	1.1	29.6	1.5
bcsstk17	10,158	777,705	1.5	112	1.5	99.7	0.0	0.0	29.2	0.2
bcsstk25	14,551	1,253,481	1.2	53	1.2	99.9	0.0	0.0	28.9	0.0
bcsstk28	3,966	252,933	3.2	134	3.4	98.1	0.4	0.8	29.6	1.1
bcsstk38	7,310	495,790	1.9	57	1.9	99.9	0.0	0.0	29.0	0.1
bodyy4	16,991	533,019	0.4	121	0.4	99.5	0.0	0.1	29.2	0.4
bodyy6	18,802	626,671	0.4	142	0.4	99.4	0.0	0.1	29.4	0.5
bundle1	10,121	90,708	0.2	12	-	-	-	-	-	-
crystm01	4,478	304,620	3.0	21	-	-	-	-	-	-
Dubcova1	15,846	622,117	0.5	48	0.5	99.9	0.0	0.0	28.7	0.1
fv1	9,109	249,383	0.6	127	0.6	98.7	0.1	0.3	29.4	1.0
fv2	9,296	255,126	0.6	202	0.6	94.7	0.4	2.9	29.7	2.4
gyro	16,814	1,166,784	0.8	15	-	-	-	-	-	-
ins2	309,381	856,286	0.0	12	-	-	-	-	-	-
jnlbrng1	39,348	1,294,013	0.2	108	0.2	99.7	0.0	0.1	29.5	0.1
minsurfo	39,964	1,153,320	0.1	208	0.1	98.9	0.1	0.6	29.5	0.6
msc04515	4,104	162,106	1.9	174	2.0	97.2	0.0	0.0	29.6	2.8
Muu	6,897	202,147	0.8	24	-	-	-	-	-	-
nasa4704	4,159	129,645	1.5	15	-	-	-	-	-	-
obstclae	39,200	1,117,584	0.1	133	0.1	99.6	0.0	0.2	29.3	0.2
s1rmq4m1	4,770	379,809	3.3	316	3.6	93.8	0.6	2.3	29.8	3.9
s1rmt3m1	4,786	287,285	2.5	226	2.6	94.6	0.8	2.9	28.1	2.5
s2rmq4m1	4,770	379,809	3.3	316	3.6	93.8	0.6	2.3	29.8	3.9
s2rmt3m1	4,786	287,285	2.5	226	2.6	94.6	0.8	2.9	28.1	2.5
s3rmq4m1	4,770	379,809	3.3	316	3.6	93.8	0.6	2.3	29.8	3.9
s3rmt3m1	4,786	287,285	2.5	226	2.6	94.6	0.8	2.9	28.1	2.5
s3rmt3m3	4,724	241,073	2.2	29	-	-	-	-	-	-
sts4098	3,880	140,630	1.9	18	-	-	-	-	-	-
t2dah_e	10,927	419,018	0.7	28	-	-	-	-	-	-
torsion1	39,200	1,117,584	0.1	133	0.1	99.6	0.0	0.2	29.3	0.2
AVERAGE	22,181	532,296	1.4	120	1.5	97.5	0.3	1.1	29.2	1.5

Table 5.2: Sparsities of submatrices obtained by 2. splitting

We see that the densities of the L_1 part is less than the densities of L factors in all matrices. This fact justifies our motivation of splitting the triangular factor into sparse and dense parts since it brings flexibility in our dHP model. However we cannot argue the same for the 2. splitting phase, because the densities of L_{11} parts are not less than the densities of L_1 parts. We presume that this fact is directly related to the sparsity pattern of the L factors abtained by Cholesky decomposition. High amount of nonzeros in the L factors squeeze in the L_d part and no apparent dense parts remain in the L_1 part to seperate more. This situation actually indicates the validity of our switch point selection method for L factors.

5.3 Results

The row-wise and column-wise directed hypergraph partitioning models are evaluated by comparing the performance of ordered matrices with unordered ones. The row-wise dHP model is exploited for reordering the lower triangular matrices which are used in row-oriented forward substitution. The column-wise dHP model is exploited for reordering the lower triangular matrices used in column-oriented forward substitution.

We observe that in row-wise dHP model no supernodes are found due to the structure of L factors obtained by Cholesky decomposition. Thus we present the performance of the clustering procedure only for the column-wise dHP model. Its performance on SpTS is tested by using column-oriented forward substitution.

Table 5.3 illustrates the number of parts in the final partitioning, namely the K value, the initial cutsize counts and the cutsize reduction fraction over the initial cutsize in percentage terms for L factor partitioning both with row-wise, column-wise and clustered column-wise dHP models. The last two columns of this table shows the supernode counts and the total weights of supernodes (total number of vertices belonging to supernodes) in the clustered column-wise dHP model. These results for L_1 and L_{11} submatrices are also shown in Tables 5.4 and 5.5

with the same order, respectively.

We obtain a fulfilling improvement in terms of cutsize reduction which verifies the effectiveness of our dHP model and its applicability for other problems which can be represented as the directed hypergraph model as well. We obtain the best cutsize reduction ratios for rerdering the L_1 part, so we expect the best runtime ratios for this part and it will be indeed demonstrated by the runtime results as shown in the rest of this section.

	Row-wise dHP			Column-wise dHP			Column-wise dHP (with Clustering)				
Matrix Name	K	Initial Cutsize	Cut Imp%	K	Initial Cutsize	Cut Imp%	K	Initial Cutsize	Cut Imp%	S.n. Cnt	S.n. Wgh
aft01	51	26,365	33.1	50	5,963	23.3	48	29,655	52.0	63	760
bcsstk17	156	86,813	23.9	157	36,493	27.7	83	59,156	39.2	33	304
bcsstk18	114	106,553	14.5	110	26,846	13.4	129	130,550	28.7	15	47
bcsstk25	239	183,554	17.6	228	59,584	16.3	256	204,750	25.6	13	50
bcsstk28	49	19,962	19.7	50	8,272	27.7	54	24,421	32.5	1	12
bcsstk38	117	69,870	19.9	111	26,027	19.1	28	22,615	31.3	136	1,600
bodyy4	117	71,049	15.8	110	15,533	21.4	121	87,300	34.7	0	0
bodyy6	131	76,153	14.5	124	18,645	19.8	139	98,150	35.4	4	31
bundle1	69	115,686	21.4	62	7,708	8.9	1	0	0.0	3,324	10,580
crystm01	53	23,302	19.7	51	7,859	28.0	1	0	0.0	372	4,872
Dubcova1	104	75,331	23.7	106	14,915	23.8	118	95,014	41.4	1	3
fv1	54	27,771	27.0	56	6,991	26.0	59	35,530	45.5	1	3
fv2	61	31,463	23.4	60	7,565	26.2	64	41,170	47.5	1	3
gyro	179	92,236	22.9	184	34,272	35.4	203	112,790	36.1	1	3
ins2	195	669,859	17.3	256	3,486	6.1	201	840,646	40.6	0	0
jnlbrng1	238	151,834	20.9	224	31,614	21.6	252	196,950	40.1	1	5
minsurfo	236	149,052	19.0	226	33,672	19.7	248	180,669	34.5	1	5
msc04515	35	12,962	25.7	37	4,812	25.6	39	16,754	45.0	24	248
Muu	38	16,064	34.6	32	3,052	24.4	39	18,204	45.0	44	521
nasa4704	44	32,044	14.3	41	7,894	22.7	52	40,871	31.4	4	48
obstclae	228	138,363	15.9	225	34,169	19.1	241	178,793	36.4	1	5
s1rmq4m1	91	52,658	15.2	92	20,736	15.1	91	55,859	24.1	47	552
s1rmt3m1	74	38,340	15.2	77	16,381	23.7	1	0	0.0	403	5,488
s2rmq4m1	91	52,658	15.2	92	20,736	15.1	91	55,859	24.1	47	552
s2rmt3m1	74	38,340	15.2	77	16,381	23.7	1	0	0.0	403	5,488
s3rmq4m1	91	52,658	15.2	92	20,736	15.1	91	55,859	24.1	47	552
s3rmt3m1	74	38,340	15.2	77	16,381	23.7	1	0	0.0	403	5,488
s3rmt3m3	66	29,251	20.9	63	12,127	26.4	14	5,240	49.5	321	3,929
sts4098	27	17,430	14.0	29	3,013	29.3	13	14,972	37.7	51	449
t2dah_e	82	41,928	24.7	81	11,957	23.0	90	54,004	45.7	11	129
torsion1	228	138,363	15.9	225	34,169	19.1	241	178,793	36.4	1	5
AVERAGE	110	86,331	19.7	110	18,322	21.6	97	91,438	31.1	186	1,346

Table 5.3: Cutsize reduction by directed hypergraph partitioning on L factors

	Row-wise dHP			Column-wise dHP			Column-wise dHP (with Clustering)				
Matrix Name	K	Initial Cutsize	Cut Imp%	K	Initial Cutsize	Cut Imp%	K	Initial Cutsize	Cut Imp%	S.n. Cnt	S.n. Wgh
aft01	34	11391	38.4	32	2,728	38.9	37	16,817	63.3	3	33
bcsstk17	109	46,892	24.9	111	19,112	27.5	115	56,791	41.6	7	82
bcsstk25	178	122,176	19.1	179	43,838	22.2	194	137,619	27.1	13	50
bcsstk28	35	10,593	25.2	36	4,937	31.4	33	10,930	41.3	31	369
bcsstk38	71	34,747	20.9	67	11,631	23.3	77	39,737	33.3	22	246
bodyy4	77	33,270	31.7	73	7,211	29.4	84	41,419	47.9	3	23
bodyy6	85	35,080	23.4	85	9,344	28.4	99	44,156	38.6	1	9
bundle1	15	16,215	22.5	16	754	11.8	1	0	0.0	3,313	7,587
crystm01	41	16,603	24.0	41	6,247	32.1	7	3,764	36.4	301	3,766
Dubcova1	87	51,931	21.1	84	10,208	28.3	98	68,949	44.8	1	3
fv1	35	11,593	30.1	35	2,523	36.9	35	13,859	45.5	5	47
fv2	36	11,908	30.3	33	2,464	22.5	43	16,994	53.3	1	3
gyro	165	75,556	22.7	160	27,625	35.8	179	88,978	35.3	2	6
ins2	148	2,697	100.0	128	208	43.8	195	7,089	100.0	0	0
jnlbrng1	182	101,694	28.9	188	20,344	21.5	199	119,881	44.1	1	5
minsurfo	163	76,931	28.0	165	16,426	25.3	176	98,444	48.0	1	5
msc04515	23	6,866	31.5	24	2,363	32.0	21	7,170	51.0	24	248
Muu	32	11,783	31.5	30	2,538	21.9	32	13,009	51.0	44	521
nasa4704	17	5,286	21.7	18	1,503	38.3	18	7,023	51.8	37	407
obstclae	153	69,122	23.5	154	15,170	26.6	175	85,616	40.8	1	5
s1rmq4m1	52	21,173	23.9	52	8,641	25.9	56	25,465	37.4	29	353
s1rmt3m1	40	12,020	22.1	41	5,685	29.2	43	14,263	45.5	13	120
s2rmq4m1	52	21,173	23.9	52	8,641	25.9	56	25,465	37.4	29	353
s2rmt3m1	40	12,020	22.1	41	5,685	29.2	43	14,263	45.5	13	120
s3rmq4m1	52	21,173	23.9	52	8,641	25.9	56	25,465	37.4	29	353
s3rmt3m1	40	12,020	22.1	41	5,685	29.2	43	14,263	45.5	13	120
s3rmt3m3	32	9,284	25.5	34	4,565	35.1	27	6,680	38.6	78	939
sts4098	18	7,342	19.9	18	1,696	40.7	23	12,742	47.8	4	38
t2dah_e	61	23,648	24.8	58	6,180	33.3	63	28,098	42.2	11	129
torsion1	153	69,122	23.5	154	15,170	26.6	175	85,616	40.8	1	5
AVERAGE	74	32,044	27.7	73	9,259	29.3	80	37,686	43.8	134	532

Table 5.4: Cutsize reduction by directed hypergraph partitioning on L_1 part

	Row-wise dHP			Column-wise dHP			Column-wise dHP (with Clustering)				
Matrix Name	K	Initial Cutsize	Cut Imp%	K	Initial Cutsize	Cut Imp%	K	Initial Cutsize	Cut Imp%	S.n. Cnt	S.n. Wgh
aft01	35	11,825	36.3	32	2,716	28.7	36	16,869	62.6	3	33
bcsstk17	107	46,062	23.9	109	19,101	28.8	116	56,221	39.9	7	82
bcsstk25	179	122,213	19.8	178	43,193	23.0	197	0	0.0	13	50
bcsstk28	34	10,598	24.7	33	4,481	36.4	34	11,254	42.0	31	369
bcsstk38	70	34,653	20.7	70	12,494	18.7	76	39,883	32.5	22	246
bodyy4	77	32,712	29.5	69	6,781	24.9	79	40,537	48.9	3	23
bodyy6	88	36,263	23.7	86	9,379	29.9	99	46,978	43.7	1	9
Dubcova1	87	51,907	21.2	83	10,202	27.8	98	70,750	47.0	1	3
fv1	33	10,797	30.9	32	2,264	29.3	35	13,221	51.5	5	47
fv2	33	10,542	33.8	33	2,455	28.1	37	13,637	50.6	1	3
jnlbrng1	184	91,426	21.8	187	20,003	23.9	200	0	0.0	1	5
minsurfo	158	70,615	24.9	158	15,671	25.9	176	89,811	44.7	1	5
msc04515	22	6,633	35.1	22	2,274	27.8	21	7,183	50.6	24	248
obstclae	153	71,037	23.7	159	15,152	25.2	170	84,795	41.6	1	5
s1rmq4m1	48	19,948	24.1	50	8,393	24.6	53	24,434	35.3	29	353
s1rmt3m1	39	11,522	23.7	38	5,888	35.5	42	14,558	46.6	13	120
s2rmq4m1	48	19,948	24.1	50	8,393	24.6	53	24,434	35.3	29	353
s2rmt3m1	39	11,522	23.7	38	5,888	35.5	42	14,558	46.6	13	120
s3rmq4m1	48	19,948	24.1	50	8,393	24.6	53	24,434	35.3	29	353
s3rmt3m1	39	11,522	23.7	38	5,888	35.5	42	14,558	46.6	13	120
torsion1	153	71,037	23.7	159	15,152	25.2	170	84,795	41.6	1	5
AVERAGE	80	36,797	25.6	80	10,674	27.8	87	32,996	40.1	11	122

Table 5.5: Cutsize reduction by directed hypergraph partitioning on L_{11} part

For the rest of the results, the improvements are given in percentage terms. The positive values stands for real improvements and negative values means worsening in runtime. We refer the row-wise, column-wise and clustered column-wise dHP models as row-dHP, column-dHP and clustered-dHP for simplicity. Let us denote the runtime of row-oriented forward substitution with a matrix M as $t_r(M)$, and the runtime of column-oriented forward substitution with matrix M as $t_c(M)$.

The runtime improvements of the dHP model for solving triangular solve with L, L_1 and L_{11} matrices are given in Tables 5.6, 5.7 and 5.8, respectively. The second column of these tables gives the improvement of just applying column-oriented instead of row oriented forward substitution, i.e. the ratio $\frac{t_c(L)-t_r(L)}{t_r(L)}$ in percent for a comparison of original row-oriented and column-oriented runtimes. The third column gives the row-oriented forward substitution runtime improvements gained by row-dHP model over the original $t_r(L)$ time. That is, if we represent the ordered version of L factor as $L^{'}$, this column gives the ratio $\frac{t_r(L)-t_r(L^{'})}{t_r(L)}$ in percent.

In the forth and sixth columns, the column-oriented forward substitution runtime improvements gained by column-dHP and clustered-dHP model over the $t_c(L)$ time are given, respctively. In order to compare the results over the same bases, we also present the time reduction ratio of the column-dHP and clustered-dHP models over the original $t_r(L)$ time in fifth and seventh columns. The best of the improvements gained by row-dHP, column-dHP and clustered-dHP with respect to $t_r(L)$ is shown in the last column of each table.

Along with performing the directed hypergraph partitioning-based reordering on lower triangular L, L_1 and L_{11} matrices, we also apply the hypergraph partitioning-based reordering method proposed in [6] on rectangular L_2 and L_{12} matrices for cache utilization in SpMxV. Since the column order of L_2 and L_{12} are determined by the permutation vector obtained by dHP on L_1 and L_{11} parts respectively, we could not reorder the columns of the rectangular L_2 and L_{12} matrices. Hence we apply a row reordering on these rectangular matrices with the method proposed in [6].

	$t_c(L)$ imp% over	Row-dHP imp% over	Column-dHP imp% over		Clustered-dHP imp% over		Best dHP imp% over
Matrix	$t_r(L)$	$t_r(L)$	$t_c(L)$	$t_r(L)$	$t_c(L)$	$t_r(L)$	$t_r(L)$
aft01	3.8	7.2	7.1	10.6	4.8	8.4	7.2
bcsstk17	-1.9	0.8	1.5	-0.3	0.6	-1.3	0.8
bcsstk18	3.3	3.2	1.6	4.8	0.3	3.5	4.8
bcsstk25	-0.1	-0.2	0.1	0.0	-0.5	-0.7	0.0
bcsstk28	2.2	0.8	-1.4	0.8	-1.8	0.4	0.8
bcsstk38	3.7	5.5	-0.3	3.4	-0.7	3.0	5.5
bodyy4	0.8	1.2	3.1	3.8	-1.2	-0.5	3.8
bodyy6	0.8	0.6	2.1	2.9	-0.7	0.1	2.9
bundle1	-4.2	-0.7	1.9	-2.2	-0.6	-4.8	-0.7
crystm01	0.7	0.2	0.8	1.4	5.1	5.8	5.8
Dubcova1	1.0	1.6	2.6	3.6	0.1	1.1	3.6
fv1	0.3	1.8	5.1	5.4	3.5	3.8	5.4
fv2	0.7	2.9	0.7	1.4	1.7	2.4	2.9
gyro	-2.4	-0.1	-0.2	-2.6	-8.1	-10.7	-0.1
ins2	16.8	2.8	-0.5	16.4	0.6	17.3	17.3
jnlbrng1	0.5	-0.3	0.9	1.4	0.0	0.4	1.4
minsurfo	1.4	1.4	0.7	2.1	-0.4	1.0	2.1
msc04515	-0.1	4.7	3.4	3.4	-7.0	-7.1	4.7
Muu	4.7	5.0	7.0	11.4	1.1	5.7	11.4
nasa4704	6.5	1.7	1.8	8.2	0.0	6.5	8.2
obstclae	-0.3	-0.2	0.7	0.4	-0.4	-0.7	0.4
s1rmq4m1	-1.2	-0.5	0.5	-0.8	-1.2	-2.5	-0.5
s1rmt3m1	1.2	4.2	1.2	2.4	0.9	2.1	4.2
s2rmq4m1	-1.6	-2.1	-0.9	-2.6	-1.1	-2.7	-2.1
s2rmt3m1	-3.5	0.6	1.0	-2.4	0.2	-3.2	0.6
s3rmq4m1	-0.5	1.6	1.5	1.0	-1.6	-2.1	1.6
s3rmt3m1	3.0	4.7	0.3	3.3	0.4	3.3	4.7
s3rmt3m3	-1.2	1.4	-0.2	-1.4	2.0	0.8	1.4
sts4098	14.9	6.4	3.8	18.1	4.6	18.8	18.8
t2dah_e	0.5	-0.1	-0.7	-0.2	-0.4	0.1	0.1
torsion1	1.7	1.6	-0.7	1.1	0.0	1.7	1.7
AVERAGE	1.7	1.9	1.4	3.0	0.0	1.6	3.8

Table 5.6: Runtime improvement of dHP model on L factor

Matrix	$t_c(L_1)$ imp% over $t_r(L_1)$	Row-dHP imp% over $t_r(L_1)$	Column-dHP imp% over $t_c(L_1)$	Column-dHP imp% over $t_r(L_1)$	Clustered-dHP imp% over $t_c(L_1)$	Clustered-dHP imp% over $t_r(L_1)$	Best dHP imp% over $t_r(L_1)$
aft01	10.9	10.4	5.5	15.9	0.8	11.7	15.9
bcsstk17	2.6	6.0	-0.4	2.2	-1.3	1.3	6.0
bcsstk25	1.0	-0.3	-0.9	0.1	-0.5	0.5	0.5
bcsstk28	0.1	7.3	2.3	2.4	-16.0	-15.9	7.3
bcsstk38	-8.3	0.6	4.9	-3.0	4.6	-3.3	0.6
bodyy4	6.7	3.5	-1.2	5.6	-0.3	6.4	6.4
bodyy6	-5.6	1.4	0.2	-5.4	7.0	1.8	1.8
bundle1	27.5	20.7	1.8	28.9	5.4	31.4	31.4
crystm01	8.5	3.9	-3.7	5.0	-0.2	8.3	8.3
Dubcova1	1.1	1.9	2.1	3.2	0.1	1.2	3.2
fv1	6.4	5.1	2.0	8.3	-1.3	5.2	8.3
fv2	-0.6	2.8	2.4	1.8	2.9	2.4	2.8
gyro	-1.4	5.2	2.9	1.5	-7.2	-8.7	5.2
ins2	-25.3	7.6	0.7	-24.5	1.1	-24.0	7.6
jnlbrng1	1.9	0.3	-0.2	1.7	-0.5	1.4	1.7
minsurfo	0.2	1.0	-0.3	-0.2	0.8	1.0	1.0
msc04515	-1.9	3.1	3.7	1.9	15.3	13.7	13.7
Muu	5.6	1.6	5.4	10.7	2.5	8.0	10.7
nasa4704	13.4	9.2	-0.2	13.2	0.1	13.5	13.5
obstclae	-0.4	0.2	1.9	1.5	-0.2	-0.7	1.5
s1rmq4m1	-3.6	1.9	3.8	0.4	3.7	0.2	1.9
s1rmt3m1	0.6	4.2	1.6	2.2	0.3	0.8	4.2
s2rmq4m1	-0.6	2.7	4.2	3.6	3.1	2.5	3.6
s2rmt3m1	-4.5	0.3	3.9	-0.5	-0.1	-4.6	0.3
s3rmq4m1	-0.6	7.9	4.8	4.2	3.0	2.5	7.9
s3rmt3m1	-0.6	3.1	-0.7	-1.3	1.3	0.8	3.1
s3rmt3m3	-1.0	3.7	1.4	0.4	-3.6	-4.7	3.7
sts4098	12.8	3.3	7.5	19.3	4.1	16.5	19.3
t2dah_e	1.5	-0.7	2.8	4.3	0.5	2.1	4.3
torsion1	-0.2	2.4	8.8	8.7	-0.2	-0.4	8.7
AVERAGE	1.5	4.0	2.2	3.7	0.8	2.4	6.8

Table 5.7: Runtime improvement of dHP model on L_1 part

Matrix	$t_c(L_{11})$ imp% over $t_r(L_{11})$	Row-dHP imp% over $t_r(L_{11})$	Column-dHP imp% over $t_c(L_{11})$	Column-dHP imp% over $t_r(L_{11})$	Clustered-dHP imp% over $t_c(L_{11})$	Clustered-dHP imp% over $t_r(L_{11})$	Best dHP imp% over $t_r(L_{11})$
aft01	16.9	13.4	2.8	19.3	1.3	18.0	19.3
bcsstk17	-0.8	0.4	-0.7	-1.6	-0.3	-1.2	0.4
bcsstk25	0.7	0.2	0.4	1.0	-0.4	0.3	1.0
bcsstk28	-14.6	10.0	16.9	4.8	5.1	-8.7	10.0
bcsstk38	-2.8	-1.3	-1.2	-4.1	-1.6	-4.5	-1.3
bodyy4	1.6	1.9	0.2	1.9	1.6	3.2	3.2
bodyy6	-0.5	0.3	2.9	2.4	-1.1	-1.6	2.4
Dubcova1	1.7	2.4	1.9	3.6	-0.6	1.2	3.6
fv1	2.4	12.6	6.8	9.0	4.4	6.7	12.6
fv2	-1.2	0.8	3.3	2.1	2.4	1.2	2.1
jnlbrng1	1.0	0.8	0.6	1.6	0.2	1.1	1.6
minsurfo	1.1	0.3	0.5	1.6	-0.6	0.5	1.6
msc04515	-6.1	2.2	5.1	-0.7	0.2	-5.8	2.2
obstclae	-1.2	0.0	0.2	-1.1	2.8	1.6	1.6
s1rmq4m1	0.5	1.9	0.2	0.7	-0.4	0.1	1.9
s1rmt3m1	4.3	8.6	3.0	7.2	2.8	7.0	8.6
s2rmq4m1	-8.5	-2.5	3.2	-5.0	0.4	-8.0	-2.5
s2rmt3m1	8.9	7.6	3.6	12.2	9.8	17.8	17.8
s3rmq4m1	-2.8	-0.1	-1.2	-4.1	-0.7	-3.5	-0.1
s3rmt3m1	-5.2	1.8	1.5	-3.6	0.5	-4.7	1.8
torsion1	-2.4	0.3	1.4	-1.0	2.4	0.1	0.3
AVERAGE	-1.7	3.1	2.4	0.8	1.3	1.0	4.2

Table 5.8: Runtime improvement of dHP model on L_{11} part

Table 5.9 illustrates the overall improvement gained by the splitting procedure, reordering L_1 part with the dHP model and reordering L_2 part for exploiting cache locality in SpMxV. The ratios of the row-oriented forward substitution solving times for L_1, L_2 and L_d parts over the solving time for L factor, i.e. the ratios $t_r(L_1)/t_r(L)$, $t_r(L_2)/t_r(L)$ and $t_r(L_3)/t_r(L)$ in percentage are given in the second, third and forth columns of this table, respectively. Fifth column is constructed by substracting the sum of the ratios in these three columns from 100, which is equal to the improvement over $t_r(L)$ time in percentage obtained by just applying the 1$^{\text{st}}$ splitting procedure.

The best improvement over $t_r(L_1)$ gained by reordering L_1 part with our dHP models and the improvement in SpMxV runtime due to reordering L_2 part with respect to $t_r(L_2)$ are given in sixth and seventh columns respectively. The next column shows the effect of these improvements upon the splitted situation. In other words, if we denote the ordered version of L_1 as $L_1^{'}$ and the ordered L_2 matrix as $L_2^{'}$, it indicates the runtime reduction ratio of $t_r(L_1^{'}) + t_r(L_2^{'}) + t_r(L_d)$ sum over the original $t_r(L_1) + t_r(L_2) + t_r(L_d)$ time in percent. The last column gives the overall improvement of the first splitting, i.e. the acceleration ratio of $t_r(L_1^{'}) + t_r(L_2^{'}) + t_r(L_d)$ sum over the original $t_r(L)$ time.

We see that in average, the splitting method gives 7.1% runtime reduction on $t_r(L)$ while we obtain 6.8% improvement on $t_r(L_1)$ and 8.8% improvement for $t_(L_2)$ with reordering methods. We see that the original solving time ratio for L_1 part is higher than other parts, so an improvement on this part effects the overall improvement ratio most. With the reordering methods, we get 6.6% improvement over the splitted model and in total we reach 13.2% average runtime reduction.

A similar chart for the overall improvements in second splitting is constructed in Table 5.10. It is observed that the splitting procedure does not yield an improvement for this case because the L_{1d} part is not dense enough for exploiting this structure as in the first splitting. Yet our dHP model provides 4.2% improvement over the L_{11} part and we obtain 4.2% improvement over the $t_r(L_1)$ time.

Matrix	Org ratios over $t_r(L)$			Imp via	Improvements on		Imp upon	Imp on
Name	$t_r(L_1)$	$t_r(L_2)$	$t_r(L_d)$	Splitting	$t_r(L_1)$	$t_r(L_2)$	splitting	$t_r(L)$
aft01	69.7	9.8	7.9	12.5	15.9	-0.2	12.6	23.6
bcsstk17	73.8	12.9	14.7	-1.4	6.0	10.6	5.7	4.4
bcsstk25	77.1	9.8	10.4	2.7	0.5	3.6	0.8	3.4
bcsstk28	68.3	8.7	11.3	11.7	7.3	0.2	5.6	16.7
bcsstk38	60.0	19.0	21.2	-0.2	0.6	12.9	2.8	2.6
bodyy4	72.2	14.6	7.3	5.8	6.4	24.0	8.7	14.0
bodyy6	74.4	12.3	6.8	6.4	1.8	3.5	1.9	8.2
bundle1	25.3	58.9	14.7	1.1	31.4	1.2	8.8	9.8
crystm01	79.8	5.8	7.0	7.4	8.3	2.0	7.2	14.1
Dubcova1	86.8	6.8	2.7	3.7	3.2	2.5	3.1	6.7
fv1	63.4	12.0	9.8	14.8	8.3	-10.2	4.8	18.9
fv2	65.9	18.4	10.0	5.8	2.8	15.3	4.9	10.4
gyro	89.4	4.8	4.3	1.6	5.2	16.9	5.6	7.1
ins2	46.4	54.6	0.0	-1.0	7.6	43.5	27.0	26.3
jnlbrng1	83.5	10.9	4.2	1.4	1.7	20.7	3.7	5.1
minsurfo	75.6	12.9	7.9	3.5	1.0	-4.7	0.2	3.7
msc04515	55.7	10.9	14.1	19.3	13.7	0.0	9.4	26.9
Muu	79.4	6.6	4.1	9.8	10.7	0.2	9.4	18.4
nasa4704	32.4	24.1	19.1	24.4	13.5	20.6	12.3	33.8
obstclae	75.2	14.2	8.9	1.7	1.5	0.2	1.2	2.9
s1rmq4m1	53.1	20.3	17.2	9.4	1.9	26.4	7.0	15.7
s1rmt3m1	47.4	22.4	19.4	10.8	4.2	11.3	5.1	15.4
s2rmq4m1	56.1	20.3	19.2	4.5	3.6	14.3	5.2	9.4
s2rmt3m1	47.9	19.8	18.6	13.6	0.3	7.7	1.9	15.3
s3rmq4m1	55.7	20.8	17.5	6.0	7.9	7.7	6.4	12.1
s3rmt3m1	50.1	27.6	23.9	-1.6	3.1	2.1	2.1	0.5
s3rmt3m3	51.1	18.7	19.9	10.3	3.7	30.6	8.5	18.0
sts4098	69.3	10.9	6.6	13.2	19.3	0.3	15.5	26.6
t2dah_e	68.6	9.4	7.3	14.8	4.3	0.1	3.4	17.7
torsion1	74.9	13.8	9.2	2.1	8.7	0.6	6.7	8.7
AVERAGE	64.3	17.1	11.5	7.1	6.8	8.8	6.6	13.2

Table 5.9: Runtime improvement of dHP by 1. splitting

Matrix	Org ratios over $t_r(L_1)$			Imp via	Improvements on		Imp upon	Imp on
Name	$t_r(L_{11})$	$t_r(L_{12})$	$t_r(L_{1d})$	Splitting	$t_r(L_{11})$	$t_r(L_{12})$	splitting	$t_r(L_1)$
aft01	101.2	0.9	1.4	-3.5	19.3	0.9	18.8	16.0
bcsstk17	95.6	0.1	0.3	4.1	0.4	0.1	0.4	4.4
bcsstk25	100.0	0.0	0.1	-0.1	1.0	0.0	1.0	0.9
bcsstk28	100.2	0.8	1.4	-2.4	10.0	0.8	9.8	7.6
bcsstk38	99.4	0.1	0.2	0.3	-1.3	0.1	-1.3	-1.1
bodyy4	99.9	0.1	0.4	-0.4	3.2	0.1	3.2	2.8
bodyy6	97.9	0.1	0.4	1.5	2.4	0.1	2.4	3.9
Dubcova1	94.3	0.0	0.2	5.5	3.6	0.0	3.6	8.9
fv1	99.8	0.3	1.0	-1.1	12.6	0.3	12.4	11.4
fv2	95.2	1.9	1.8	1.1	2.1	1.9	2.1	3.2
jnlbrng1	99.7	0.1	0.1	0.1	1.6	0.1	1.6	1.6
minsurfo	98.5	0.4	0.4	0.7	1.6	0.4	1.6	2.4
msc04515	95.0	0.4	3.9	0.7	2.2	0.4	2.1	2.9
obstclae	99.9	0.1	0.2	-0.3	1.6	0.1	1.6	1.4
s1rmq4m1	95.9	1.6	3.5	-1.0	1.9	1.6	1.8	0.9
s1rmt3m1	100.9	2.0	2.4	-5.3	8.6	2.0	8.2	3.4
s2rmq4m1	90.1	1.6	3.5	4.8	-2.5	1.6	-2.3	2.6
s2rmt3m1	98.0	2.0	2.5	-2.5	17.8	2.0	17.1	15.0
s3rmq4m1	96.1	1.5	3.4	-1.0	-0.1	1.5	0.0	-1.1
s3rmt3m1	96.1	1.9	2.2	-0.2	1.8	1.9	1.7	1.5
torsion1	100.6	0.1	0.2	-0.9	0.3	0.1	0.3	-0.6
AVERAGE	97.8	0.8	1.4	0.0	4.2	0.8	4.1	4.2

Table 5.10: Runtime improvement of dHP by 2. splitting

Finally, Table 5.11 presents the effects of reordering L, L_1 and L_{11} matrices with our dHP model over the original runtime results. The direct improvements of dHP on $t_r(L)$, $t_r(L_1)$ and $t_r(L_{11})$ obtained by reordering L, L_1 and L_{11} matrices are given in second, third and fifth columns respectively in percentage terms. The forth column shows the imrovement of reordering L_1 over the original $t_r(L)$ time while the sixth and seventh columns give the improvement gained by reordering L_{11} with respect to the $t_r(L_{11})$ and $t_r(L)$ times respectively.

We observe that our dHP model does not reach its best performance on the original L factors. However the first splitting method favours the dHP model noticeably. Reordering L_1 matrix is able to accelerate the forward substitution time up to 31.4% and in average 6.8%. Speedups of up to 33.8% and 13.2% in average are observed when the first splitting scheme and the reordering methods are applied. Yet we cannot argue the same for the second splitting. Although reordering on L_{11} matrix yields 4.2% improvement over $t_r(L_{11})$ and 10.3% speedup with respect to $t_r(L)$ in average, we see that it still does not beat the improvements gained by the first splitting along with the reordering techniques. Hence we propose using our reordering methods after splitting the triangular matrix once in order to exploit cache locality in SpTS.

Matrix	Reordering L	Reordering L_1		Reordering L_{11}		
	Imp% over	Imp% over		Imp% over		
Name	$t_r(L)$	$t_r(L_1)$	$t_r(L)$	$t_r(L_{11})$	$t_r(L_1)$	$t_r(L)$
aft01	10.6	15.9	23.6	19.3	16.0	23.6
bcsstk17	0.8	6.0	4.4	0.4	4.4	3.2
bcsstk18	4.8	-	-	-	-	-
bcsstk25	0.0	0.5	3.4	1.0	0.9	3.7
bcsstk28	0.8	7.3	16.7	10.0	7.6	16.9
bcsstk38	5.5	0.6	2.6	-1.3	-1.1	1.6
bodyy4	3.8	6.4	14.0	3.2	2.8	11.4
bodyy6	2.9	1.8	8.2	2.4	3.9	9.7
bundle1	-0.7	31.4	9.8	-	-	-
crystm01	1.4	8.3	14.1	-	-	-
Dubcova1	3.6	3.2	6.7	3.6	8.9	11.6
fv1	5.4	8.3	18.9	12.6	11.4	20.8
fv2	2.9	2.8	10.4	2.1	3.2	10.7
gyro	-0.1	5.2	7.1	-	-	-
ins2	16.4	7.6	26.3	-	-	-
jnlbrng1	1.4	1.7	5.1	1.6	1.6	5.1
minsurfo	2.1	1.0	3.7	1.6	2.4	4.7
msc04515	4.7	13.7	26.9	2.2	2.9	20.9
Muu	11.4	10.7	18.4	-	-	-
nasa4704	8.2	13.5	33.8	-	-	-
obstclae	0.4	1.5	2.9	1.6	1.4	2.8
s1rmq4m1	-0.5	1.9	15.7	1.9	0.9	15.2
s1rmt3m1	4.2	4.2	15.4	8.6	3.4	15.0
s2rmq4m1	-2.1	3.6	9.4	-2.5	2.6	8.9
s2rmt3m1	0.6	0.3	15.3	17.8	15.0	22.3
s3rmq4m1	1.6	7.9	12.1	-0.1	-1.1	7.0
s3rmt3m1	4.7	3.1	0.5	1.8	1.5	-0.3
s3rmt3m3	1.4	3.7	18.0	-	-	-
sts4098	18.1	19.3	26.6	-	-	-
t2dah_e	-0.1	4.3	17.7	-	-	-
torsion1	1.6	8.7	8.7	0.3	-0.6	1.7
AVERAGE	3.7	6.8	13.2	4.2	4.2	10.3

Table 5.11: Overall runtime improvement of dHP

Chapter 6

Conclusion

We propose a novel directed hypergraph partitioning model for reordering the sparse triangular matrices to minimize cache misses in SpTS. The model primarily aims to exploit temporal locality of the solution vector. We represent the data localities arised in SpTS as a directed hypergraph and define an ordered partitioning on this directed hypergraph. Our motivation behind adopting a hypergraph representation is to accurately capture the cache miss counts in SpTS by the cutsize metric of hypergraph partitioning models.

We develop a framework for cache utilization in SpTS which consist of the following steps. First we split the triangular factor into dense and sparse components, which produces a sparse lower triangular matrix, a rectangular matrix and a dense lower triangular matrix. This procedure decomposes SpTS as a collection of a SpTS, a SpMxV and a dense triangular solve. Then we apply our directed hypergraph partitioning model on the sparse triangular submatrix for exploiting data locality. We also reorder the rows of the rectangular submatrix for cache utilization in SpMxV by applying the model proposed in [6].

We conduct experiments for examining the effectiveness of our directed hypergraph partitioning model on both the original lower triangular factor and the lower triagular submatrices obtained by the splitting method along with further splitting procedures. We conclude that the best improvement is obtained with one splitting procedure whose framework is explained above.

The effectiveness of the dHP model is demonstrated by the cutsize gain ratios. The relative lowliness in the performance improvements might be related to conduct experiments on a cache which is not fully associative. Furthermore, just partitioning the hypergraph without applying an accurate clustering may yield highly connected parts to fall into different parts. We see that the clustering algorithm in dHP does not enhance the cutsize gain improvements with respect to the original dHP model, which means that it needs further attention to become more effective. Our future research will involve improving the clustering procedure in dHP, by augmenting the cases which yields vertices to form supernodes and by applying clustering in more than one bipartitioning levels.

Bibliography

[1] W. F. Tinney and J. W. Walker, “Direct solutions of sparse network equations by optimally ordered triangular factorization,” *Proceedings of the IEEE*, vol. 55, no. 11, pp. 1801–1809, 1967.

[2] G. H. Golub and C. F. Van Loan, *Matrix Computations*, vol. 3. JHU Press, 2012.

[3] Y. Saad, *Iterative Methods for Sparse Linear Systems.* SIAM, 2003.

[4] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix–vector multiplication on emerging multicore platforms,” *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[5] A. Pinar and M. T. Heath, “Improving performance of sparse matrix-vector multiplication,” in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, p. 30, ACM, 1999.

[6] K. Akbudak, E. Kayaaslan, and C. Aykanat, “Hypergraph partitioning based models and methods for exploiting cache locality in sparse matrix-vector multiplication,” *SIAM Journal on Scientific Computing*, vol. 35, no. 3, pp. C237–C262, 2013.

[7] C. M. Fiduccia and R. M. Mattheyses, “A linear-time heuristic for improving network partitions,” in *Design Automation, 1982. 19th Conference on*, pp. 175–181, IEEE, 1982.

[8] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, and K. A. Yelick, "Automatic performance tuning and analysis of sparse triangular solve," International Continence Society, 2002.

[9] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, p. 521, IOP Publishing, 2005.

[10] G. Dahlquist and Å. Björck, "Numerical methods in scientific computing," *Volume I, SIAM, Philadelphia*, 2007.

[11] B. J. Wall and B. A. Conway, "Genetic algorithms applied to the solution of hybrid optimal control problems in astrodynamics," *Journal of Global Optimization*, vol. 44, no. 4, pp. 493–508, 2009.

[12] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified np-complete graph problems," *Theoretical Computer Science*, vol. 1, no. 3, pp. 237–267, 1976.

[13] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley & Sons, Inc., 1990.

[14] Ü. V. Catalyürek and C. Aykanat, "Patoh: a multilevel hypergraph partitioning tool, version 3.0," *Bilkent University, Department of Computer Engineering, Ankara*, vol. 6533, 1999.

[15] G. Karypis and V. Kumar, "hmetis: A hypergraph partitioning package, version 1.5. 3," *user manual*, vol. 23, 1998.

[16] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[17] A. Dasdan and C. Aykanat, "Two novel multiway circuit partitioning algorithms using relaxed locking," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 16, no. 2, pp. 169–178, 1997.

[18] R. Battiti, A. Bertossi, and R. Rizzi, "Randomized greedy algorithms for the hypergraph partitioning problem," *Randomization Methods in Algorithm*

Design (P. Pardalos, S. Rajasekaran, and J. Rolim eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 43, pp. 21–35, 1998.

[19] Ü. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 7, pp. 673–693, 1999.

[20] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek, "Permuting sparse rectangular matrices into block-diagonal form," *SIAM Journal on Scientific Computing*, vol. 25, no. 6, pp. 1860–1879, 2004.

[21] Ü. V. Çatalyürek and C. Aykanat, "Decomposing irregularly sparse matrices for parallel matrix-vector multiplication," in *Parallel Algorithms for Irregularly Structured Problems*, pp. 75–86, Springer, 1996.

[22] G. E. Blelloch, M. A. Heroux, and M. Zagha, "Segmented operations for sparse matrix computation on vector multiprocessors," tech. rep., DTIC Document, 1993.

[23] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations," 1994.

[24] F. L. Alvarado and R. Schreiber, "Optimal parallel solution of sparse triangular systems," *SIAM Journal on Scientific Computing*, vol. 14, no. 2, pp. 446–460, 1993.

[25] J. Mayer, "Parallel algorithms for solving linear systems with sparse triangular matrices," *Computing*, vol. 86, no. 4, pp. 291–312, 2009.

[26] E. Rothberg and A. Gupta, "Parallel ICCG on a hierarchical memory multiprocessor addressing the triangular solve bottleneck," *Parallel Computing*, vol. 18, no. 7, pp. 719–741, 1992.

[27] E. Rothberg, "Alternatives for solving sparse triangular systems on distributed-memory multiprocessors," *Parallel Computing*, vol. 21, no. 7, pp. 1121–1136, 1995.

[28] M. V. Joshi, A. Gupta, G. Karypis, and V. Kumar, "A high performance two dimensional scalable parallel algorithm for solving sparse triangular systems," in *Proceedings of the Fourth International Conference on High-Performance Computing*, pp. 137–143, IEEE, 1997.

[29] P. Raghavan, "Efficient parallel sparse triangular solution using selective inversion," *Parallel Processing Letters*, vol. 8, no. 01, pp. 29–40, 1998.

[30] J. H. Saltz, "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM Journal on Scientific and Statistical Computing*, vol. 11, no. 1, pp. 123–144, 1990.

[31] M. M. Wolf, M. A. Heroux, and E. G. Boman, "Factors impacting performance of multithreaded sparse triangular solve," in *High Performance Computing for Computational Science–VECPAR 2010*, pp. 32–44, Springer, 2011.

[32] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU," *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011-001*, 2011.

[33] E. Totoni, M. T. Heath, and L. V. Kale, "Structure-adaptive parallel solution of sparse triangular linear systems," 2012.

[34] E.-J. Im and K. Yelick, "Optimizing sparse matrix computations for register reuse in SPARSITY," in *Computational Science–ICCS*, pp. 127–136, Springer, 2001.

[35] B. Smith and H. Zhang, "Sparse triangular solves for ilu revisited: data layout crucial to better performance," *International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 386–391, 2011.

[36] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

Appendix A

Pictures of Reordered Matrices

In this appendix we provide the pictures of the original matrix *bcsstk*17, its L factor, L_1 part and L_{11} part. We also show the pictures of L factor, L_1 part and L_{11} part after applying symmetric reordering with row-wise dHP model. We see that the nonzero distribution of these matrices are altered by this reordering so that the rows having similar sparsity patterns are located successively.

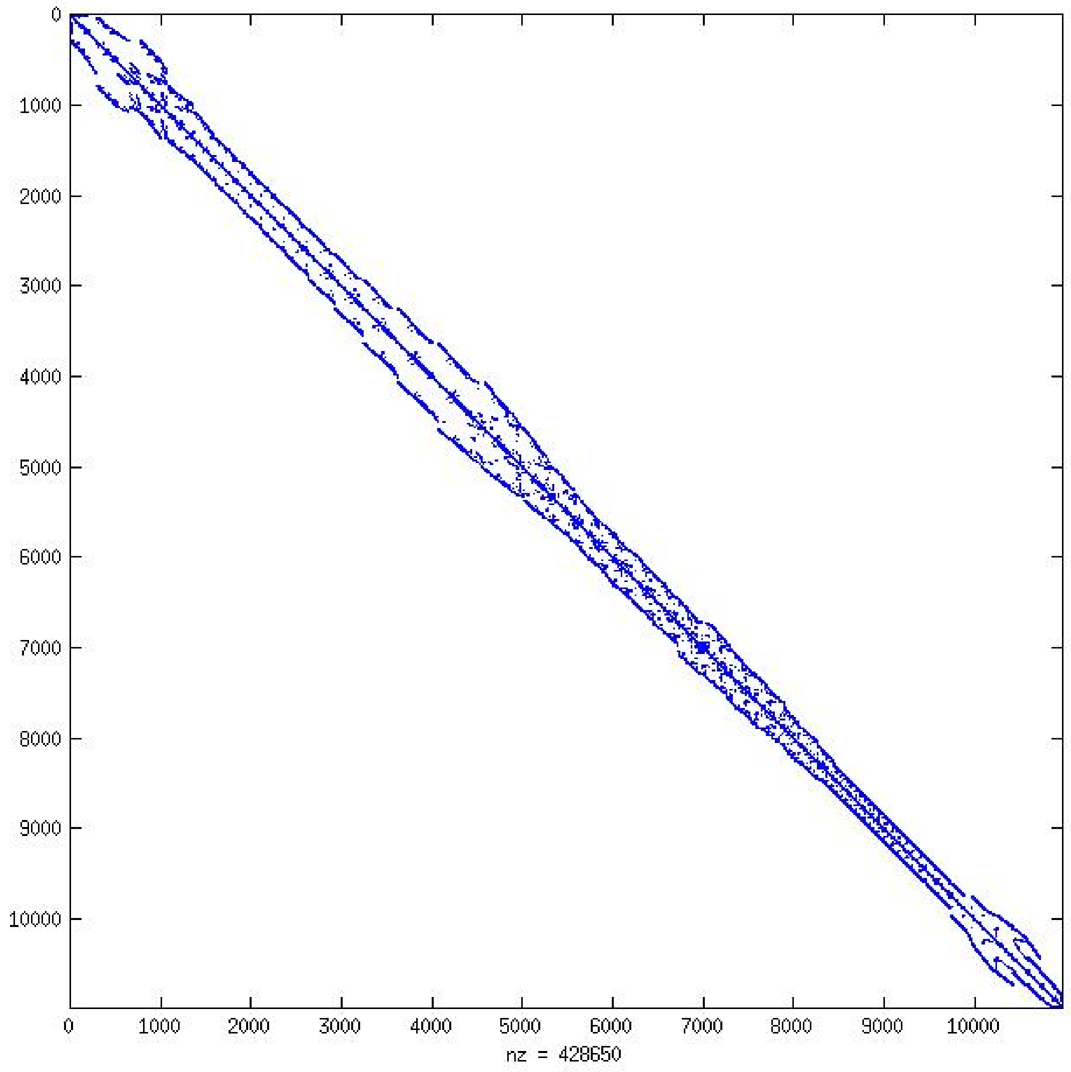


Figure A.1: Original matrix *bcsstk*17

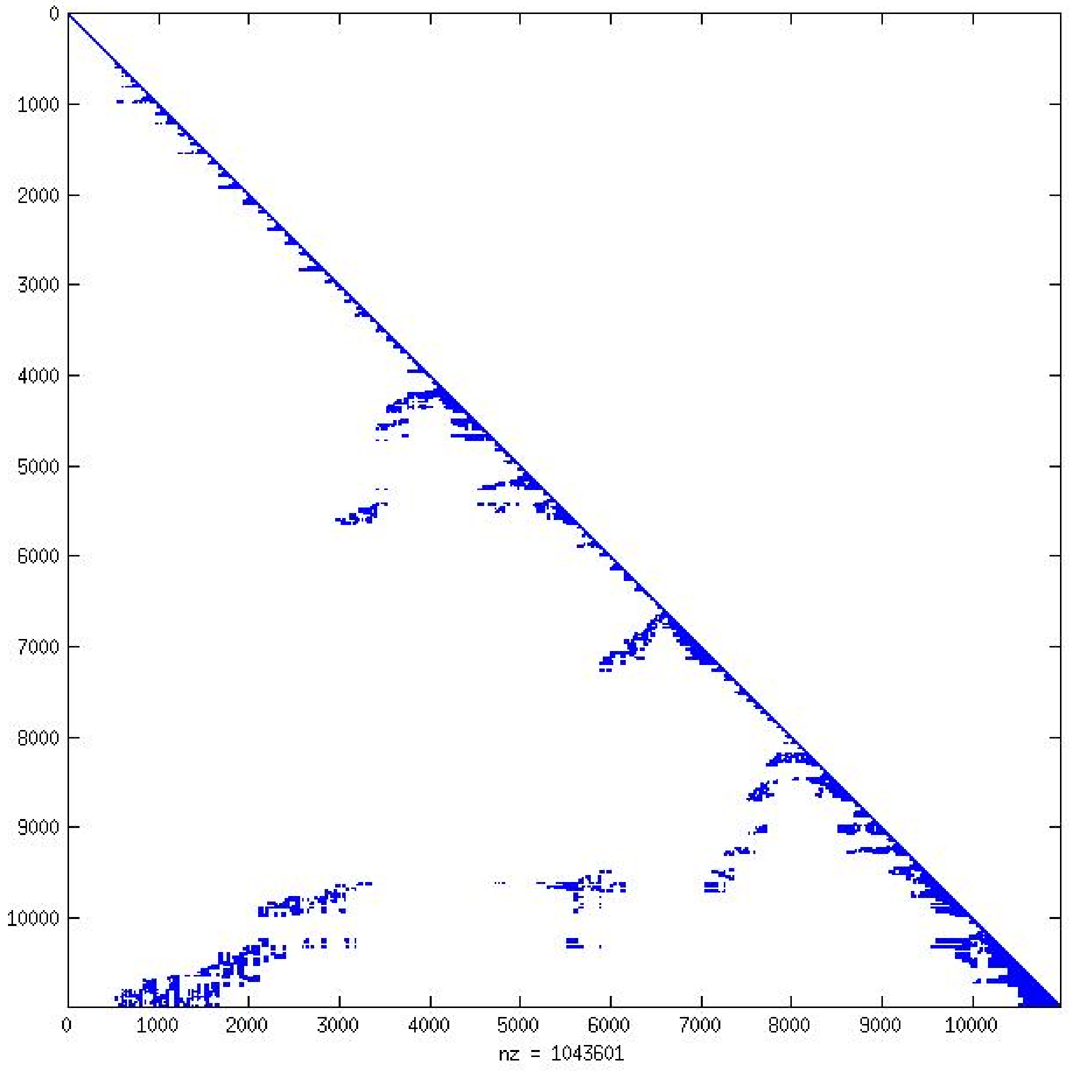


Figure A.2: L factor of matrix $bcsstk17$ obtained by Cholesky Factorization

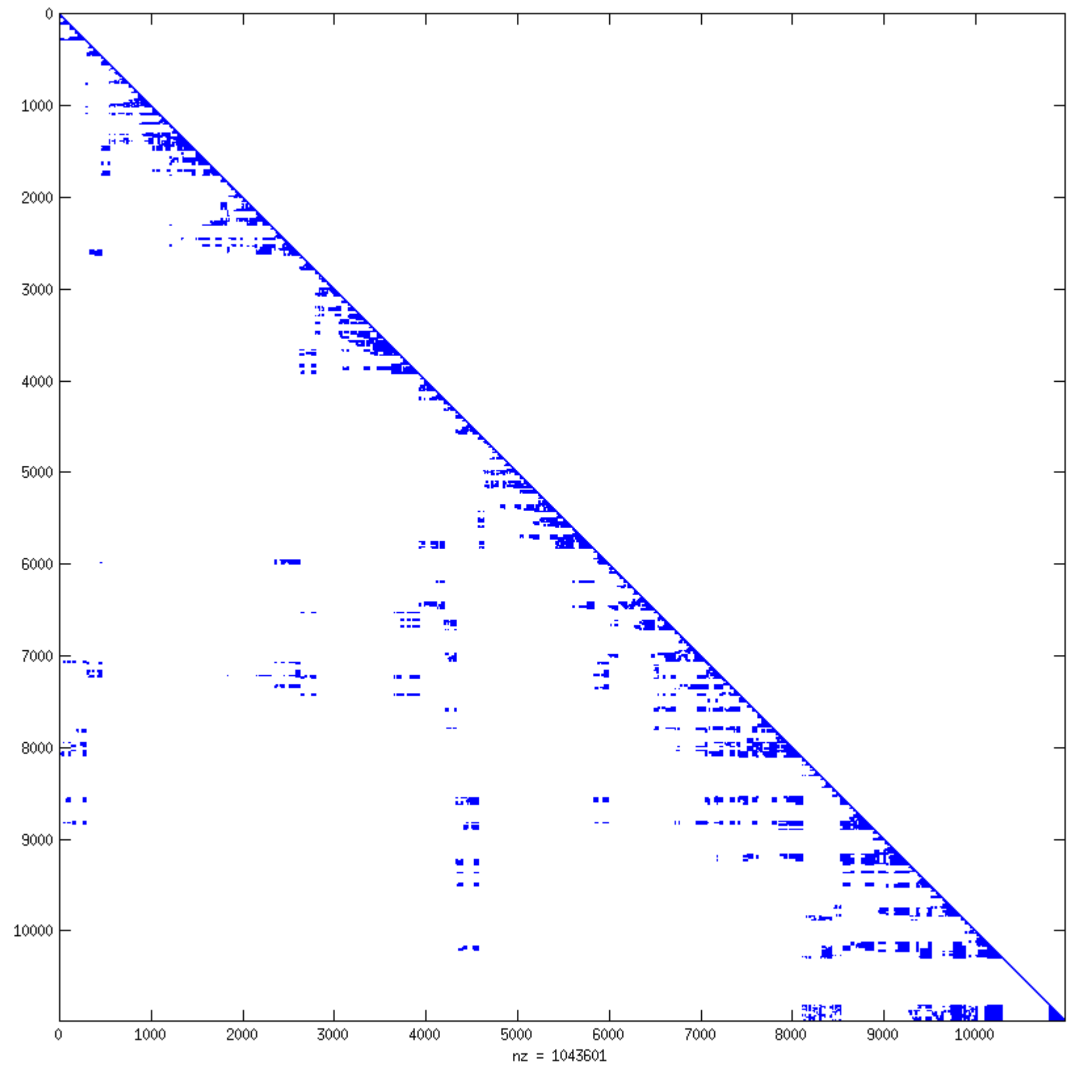


Figure A.3: L factor of matrix *bcsstk*17 after reordering with row-wise dHP

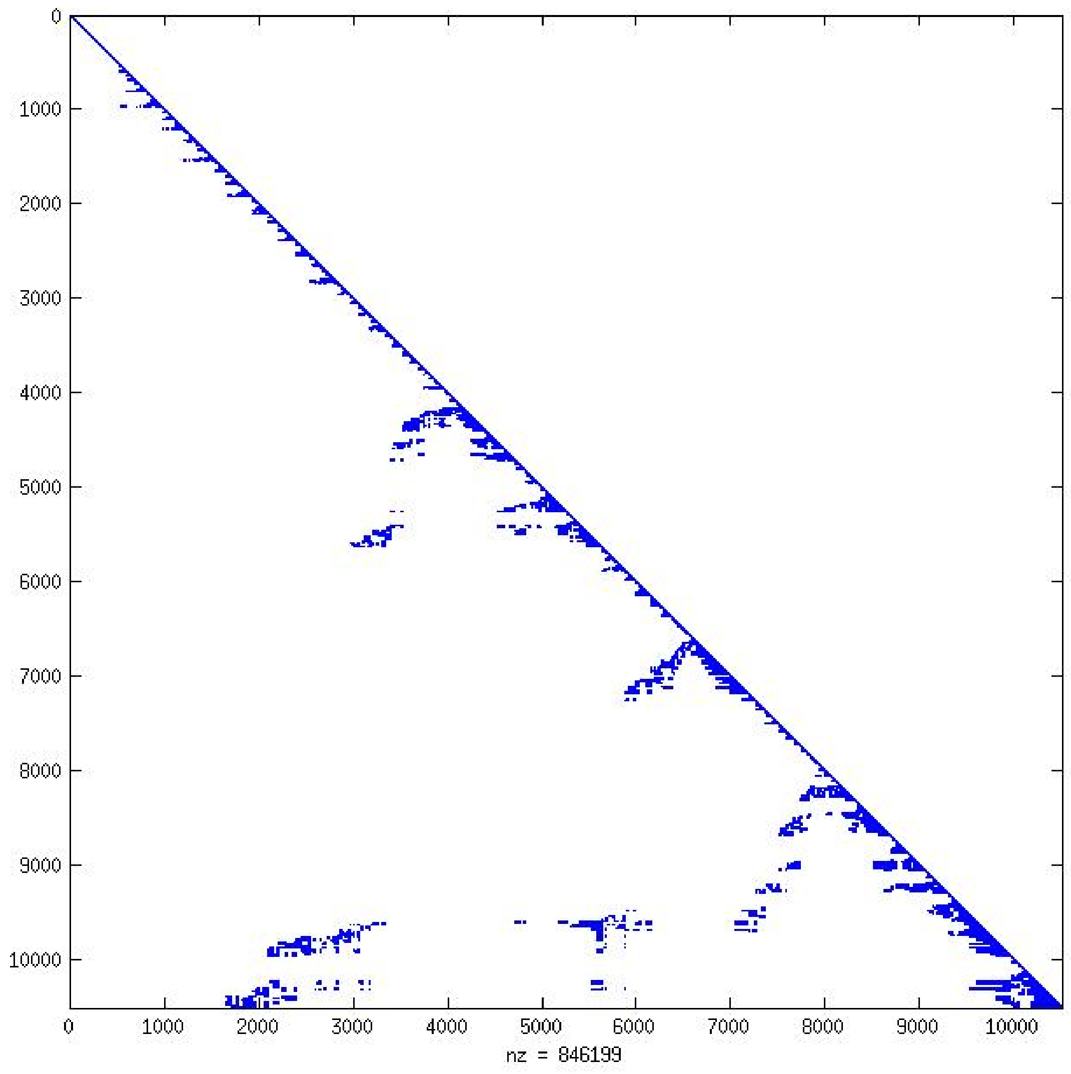


Figure A.4: L_1 part of matrix *bcsstk*17

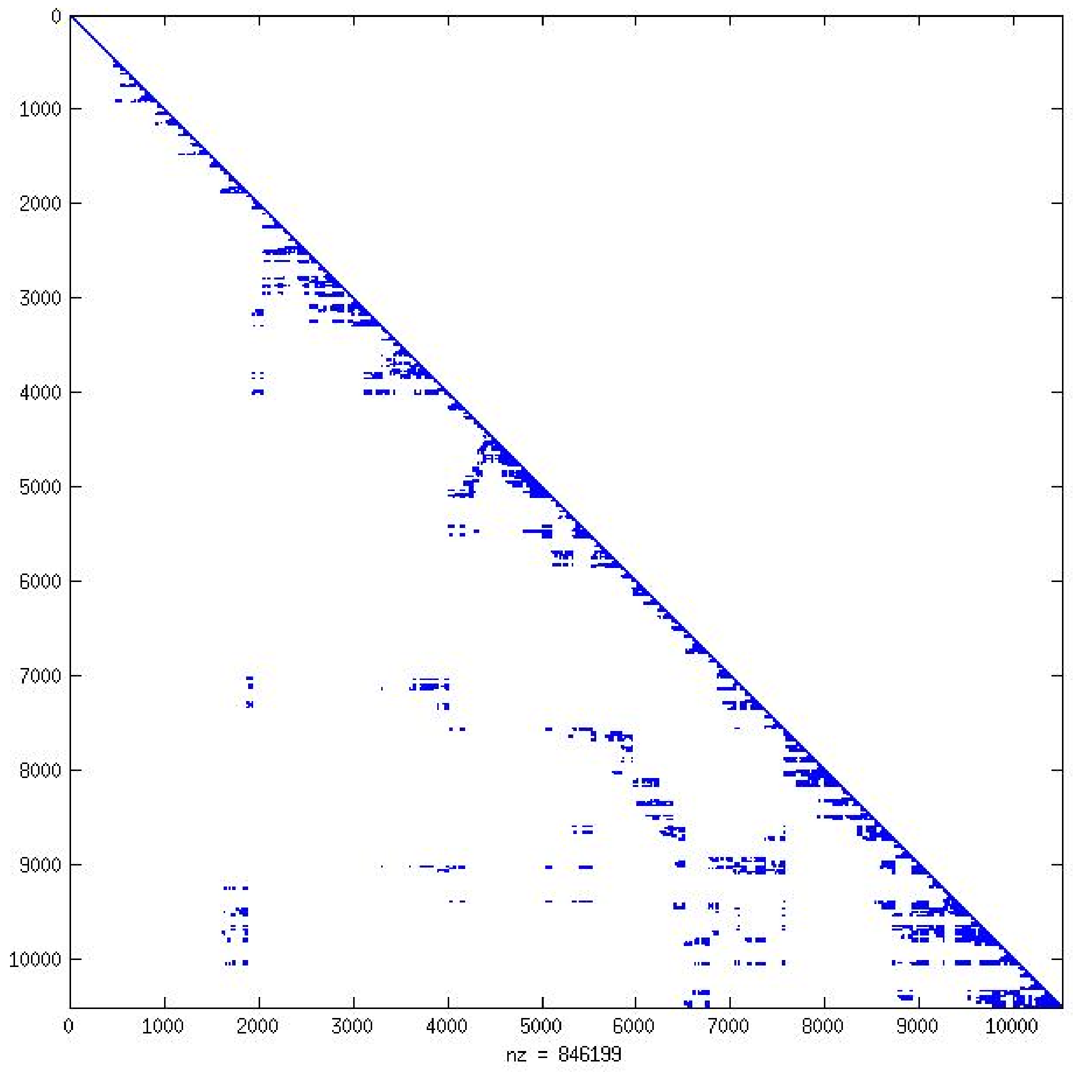


Figure A.5: L_1 part of matrix *bcsstk*17 after reordering with row-wise dHP

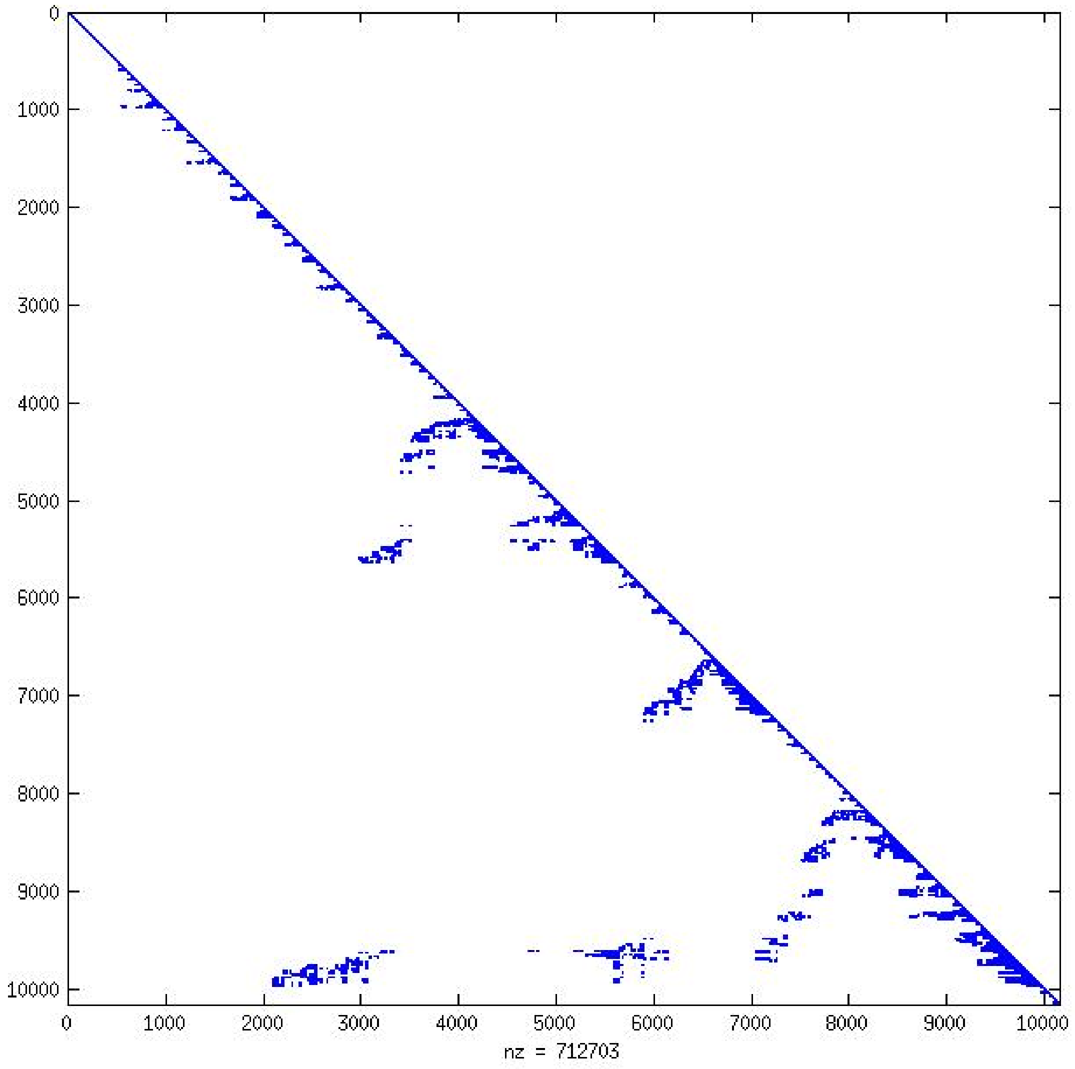


Figure A.6: L_{11} part of matrix *bcsstk*17

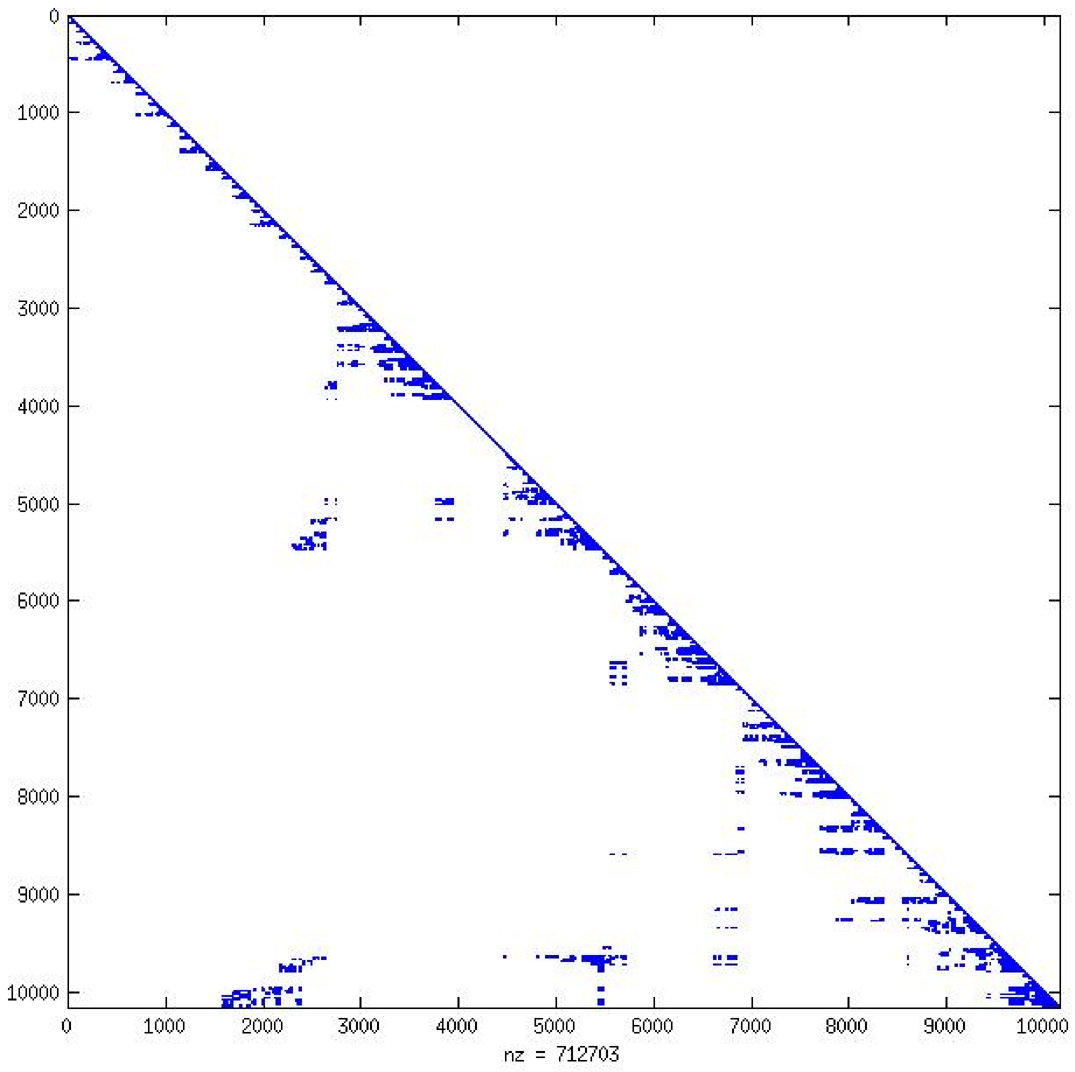


Figure A.7: L_{11} part of matrix $bcsstk17$ after reordering with row-wise dHP