PARALLEL SPARSE MATRIX VECTOR MULTIPLICATION TECHNIQUES FOR SHARED MEMORY ARCHITECTURES

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

Mehmet Başaran

September, 2014

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

Prof. Dr. Cevdet Aykanat(Advisor)

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

Assist. Prof. Dr. Can Alkan

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

Assist. Prof. Dr. Kayhan İmre

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

PARALLEL SPARSE MATRIX VECTOR MULTIPLICATION TECHNIQUES FOR SHARED MEMORY ARCHITECTURES

Mehmet Başaran
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
September, 2014

SpMxV (Sparse matrix vector multiplication) is a kernel operation in linear solvers in which a sparse matrix is multiplied with a dense vector repeatedly. Due to random memory access patterns exhibited by SpMxV operation, hardware components such as prefetchers, CPU caches, and built in SIMD units are under-utilized. Consequently, limiting parallelization efficieny. In this study we developed;

- an adaptive runtime scheduling and load balancing algorithms for shared memory systems,
- a hybrid storage format to help effectively vectorize sub-matrices,
- an algorithm to extract proposed hybrid sub-matrix storage format.

Implemented techniques are designed to be used by both hypergraph partitioning powered and spontaneous SpMxV operations. Tests are carried out on Knights Corner (KNC) coprocessor which is an x86 based many-core architecture employing NoC (network on chip) communication subsystem. However, proposed techniques can also be implemented for GPUs (graphical processing units).

Keywords: SpMxV, parallelization, KNC, Intel Xeon Phi, many-core, GPU, vectorization, SIMD, adaptive scheduling and load balancing, Work stealing, Distributed Systems, Data Locality.

ÖZET

PAYLAŞIMLI HAFIZA SİSTEMLERİ İÇİN PARALEL SEYREK MATRİS - DİZİ ÇARPİM TEKNİKLERİ

Mehmet Başaran
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Eylül, 2014

Seyrek matris dizi çarpımı, denklem çözücülerde kullanılan anahtar işlemdir. Seyrek matrix tarafından yapılan düzensiz hafıza erişimleri nedeniyle, buyruk ön yükleyicisi, işlemci ön belleği ve dizi buyrukları gibi bir çok donanım etkili bir şekilde kullanılamamaktadır. Buda paralel verimliliğin düşmesine neden olur. Bu çalışmada, paylaşımlı hafıza sistemlerinde kullanılmak üzere,

- Öğrenme yetisine sahip planlayıcı ve yük dengeleyici algoritmalar,
- Dizi buyruklarını etkili bir şekilde kullanmaya olanak sağlayan melez bir seyrek veri yapısı ve
- Bu veri yapısını oluşturmada kullanılan bir algoritma

geliştirilmiştir.

Bu alşmada belirtilen teknikler, hem ön yapılandırmalı hemde direkt olarak seyrek matrix-dizi çarpımında kullanılabilir. Testler Intel tarafından üretilen Xeon Phi adlı, x86 tabanlı çekirdeklere ve bu çekirdekleri birbirine bağlayan halka ağ protokolüne sahip, yardımcı kartlar üzerinde yapılmış tır. Önerilen teknikler ekran kartlarında da kullanılabilir.

Anahtar sözcükler: Seyrek matris-dizi çarpımı, KNC, Intel Xeon Phi, çok çekirdekli ilemciler, vektörizasyon, SIMD, öğrenebilen planlayıcı ve y¨k dengeleyiciler, i çalma, dağıtık sistemler, veri yerelliği.

Acknowledgement

I acknowledge that . . .

Contents

1 Introduction **1**

1.1 Preliminary 1

1.2 Problem Definition 2

1.3 Thesis Organization 4

2 Background **5**

2.1 Terms and Abbreviations 6

2.2 Background Infomation on SpMxV 8

2.2.1 Sparse-Matrix Storage Schemes 9

2.2.2 Space Requirements 12

2.2.3 Task decomposition techniques for SpMxV 13

2.3 Partitioning & Hypergraph model explored 15

2.3.1 Column-Net Model & Interpretation 15

2.3.2 Row-Net Model & Interpretation 16

2.3.3 Recursive Bipartitioning 17

2.4 A high level overview of Intel's Xeon Phi High Performance Computing Platform . . . 19
2.4.1 5 key features of Xeon Phi Coprocessors . . . 20
2.4.2 Thread Affinity Control . . . 24

3 Partitioning, Scheduling, and Load Balancing Algorithms 26
3.1 Ordering Only Routines . . . 28
3.1.1 Sequential Routine . . . 28
3.1.2 Dynamic OMP Loop Routine . . . 28
3.2 Ordering & blocking routines . . . 30
3.2.1 Chunk & Scatter Distribution Methods . . . 32
3.2.2 Static Routine . . . 34
3.2.3 OpenMP Task Routine . . . 35
3.2.4 Global Work Stealing Routine (GWS) . . . 36
3.2.5 Distributed Work Stealing Routine (DWS) . . . 37

4 Implementation deltails, Porting, and Fine tuning 40
4.1 Test sub-set . . . 41
4.2 General template of application behaviour . . . 42
4.3 Application Kernels . . . 43
4.3.1 CSR format . . . 43
4.3.2 JDS format . . . 44

4.4 Runtime Scheduling and Load Balance 47

4.4.1 Generic Warm-up Phase 47

4.4.2 Improving Application Scalability 47

4.4.3 Adaptiveness: Reasoning 48

4.5 Overcoming deficiencies related to storage format 51

4.5.1 Hybrid JDS-CSR storage format 52

4.5.2 Laid Back Approach 53

4.5.3 Possible Improvements & Performance Analysis 59

4.5.4 Choosing optimum partition size and decomposition algorithm 60

4.5.5 Effect of Hypergraph Partitioning: Analyzed 61

5 Experimental Results and Future Work 64

5.1 Test Environment 65

5.2 Hardware Specifications 65

5.3 Test Data 66

5.4 Future Work 91

5.4.1 Experiments with hybrid JDS-CSR format 91

5.4.2 Choosing optimum partition size and decomposition algorithm 94

5.4.3 GPU implementation 94

6 Conclusion **95**

List of Figures

2.1 Sample matrix in 2D array representation. 9

2.2 CSR representation of SpM in figure 2.1. 10

2.3 Matrix rows are sorted by their non-zero count in descending order. 10

2.4 All non-zeros are shifted to left. 11

2.5 JDS representation of SpM in Figure 2.1. 11

2.6 Rowwise 1-D Block Partitioning algorithm for 4 PEs. 13

2.7 Columnwise 1-D Block Partitioning algorithm for 4 PEs. 14

2.8 2-D Block Partitioning algorithm for 16 PEs. 14

2.9 Column-net interpretation incurs vertical border. 16

2.10 Row-net interpretation incurs horizontal border. 17

2.11 A SpM is recursively reordered and divided into four parts using column-net recursive bipartitioning scheme. And sub-matrix to PE assignment is done using rowwise 1-D partitioning algorithm. 18

2.12 An SpM is recursively reordered and divided into 4 parts using row-net recursive bipartitioning scheme. Sub-matrix to processor assignment is done by using columnwise 1-D partitioning algorithm. 18

2.13 High level view of on-die interconnect on Xeon Phi Coprocessor cards. TD (tag directory), MC (Memory Channel), L2 (Level 2 Cache), L1-d (Level 1 Data Cache), L1-i (Level 1 Instruction Cache), HWP (hardware prefetcher). 21

2.14 Compact thread affinity control. 24

2.15 Scatter thread affinity control. 25

2.16 Balanced thread affinity control. 25

3.1 Sample bipartitioning tree for column-net interpretation using CSR/JDS schemes on 5 EXs. 30

3.2 Sample matrix ordered by column-net model using rowwise 1-D partitioning algorithm for task to PE assignment. There are 5 EXs and collective reads on X vector by tasks are pointed out by density of portions' color. 31

3.3 Bipartitioning tree for 2 blocks. 33

3.4 Sample matrix ordered by column-net model using rowwise 1-D partitioning using scatter distribution method. Collective reads by task on X vector are shown by the density of potions' color. . . 33

3.5 Static routine sub-matrix distribution among 5 execution contexts. 34

3.6 Global queue and the order in which sub-matrices line up. 36

3.7 Ring work stealing scheme making efficient use of bidirectional communication ring for 60 PEs. 37

3.8 Tree work stealing scheme in action. 38

4.1 Task decomposition for CSR format. 43

4.2 Task decomposition for JDS format. 44

4.3 Constraint violation in LBA is described. 55

4.4 LBA in action. 56

4.5 Sample output for CSR implementation of LBA. 59

4.6 Possible improvement for hybrid JDS-CSR format to avoid extra prefetch issues. 60

5.1 Comparison of JDS-heavy CSR and CSR-heavy JDS in certain senarios. Hybrid format doesn't necessarily contain both JDS and CSR everytime. In this case, cost of JDS and CSR are same. Therefore, one of them is chosen depending on the implementaion. 92

5.2 Comparison of JDS-heavy CSR and CSR-heavy JDS in certain senarios. Here, at certain points costs are the same. Depending on version of LBA, one of them will be chosen. 92

List of Tables

2.1 Properties of Atmosmodd, it holds only 7 times the memory space of a dense vector with same column count. 8

4.1 Stats of choosen data sub-set. Row & column count, NNZ, and max-avg-min NNZ per row/column are given. 41

4.2 Effect of discarding permutation array from JDS format. Results are measured in GFlops and belong to single precision spontaneous static routine with 32KB sub-matrix sizes. Gains are substantial for matrices that are efficiently vectorizable. 46

4.3 Row & Column count, NNZ, and max, avg, min NNZ per row/column are given for as-Skitter and packing-500x100x100-b050. 49

4.4 Time measured (in seconds) for 100 SpMxV operations performed using as-Skitter and packing-500x100x100-b050 with Static and DWS routines both 'sp' (spontaneous) and 'hp' (hypergraph) modes. 'spc' stands for spontaneous mode partition count while 'hpc' for hypergraph mode patition count. In third row, how many times faster packing-500x100x100-b050 executes compared to as-Skitter is shown. (A hybrid storage format, proposed later in this chapter, is used to take these runs). 49

4.5 Storage format performance (measured in GFLOPs) comparison for single precision SpMxV. For CSR format, the routine that performs best is chosen to include OpenMP Dynamic Loop implementation. For JDS and CSR formats, results belong to DWS routine. 53

4.6 Hypergraph partitioning effect is observed on different schedulers. 'sp' stands for spontanenous mode, while 'hp' for hypergraph partitioning mode. 'su' is the speed up of hypergraph partitioning based sequential routine over spontaneous sequential run. Out of 3 tree work stealing routines, only chunk distribution method is tested. Results are presented in GFlops. 63

5.1 CPU and memory specifications of Xeon Phi model used in tests. 65

5.2 Cache specifications of Xeon Phi model used in tests. 65

5.3 CPU and memory specifications of Xeon model used in tests. . . . 66

5.4 For all the SpMs used to test proposed routines, row & column count, NNZ, min/avg/max NNZ per row/col, and their matrix structures are grouped by their problem types. 67

5.5 Performance is measured in GFlops and compared between MKL and default DWS (DWS-RING) routine. 'spc' stands for spontaneous mode partition (/ sub-matrix) count, while 'hppc' for hypergraph-partitioning mode partition count. 'su' is speed up of hypergraph-partitioning use over spontanenous use while running sequential routine. If, for a matrix, in spontaneous mode, DWS routine performs worse than MKL in either single or double precision versions, that matrix's row is highlighted in red, statistics for problem groups are highlighted in blue, and overall statistics in green. 77

5.6 Different variants of LBA are compared. Results are taken from single precision Static routine. Performance and measured in GFlops. Partition size is 32KB. 93

List of Algorithms

4.1 General template of application behaviour. 42
4.2 Kernel function for CSR format. 44
4.3 Kernel function for JDS format. 45
4.4 Kernel function for hybird JDS-CSR format. 52
4.5 Hybird JDS-CSR sub-matrix extraction. 54
4.6 JDS implementation of LBA to find the optimum cut for infinite vector unit length. 57
4.7 CSR implementation of LBA to find the optimum cut for infinite vector unit length. 58

Chapter 1

Introduction

1.1 Preliminary

Advancements in manufacturing technology made it possible to fit billions of transistors in a single processor. At the time of this writing, a 60+ core Xeon Phi coprocessor card has 5 billion transistor count. For a processor, higher transistor count means more computational power. But unfortunately, more computational power doesn't necessarily result in better performance. Since computations are carried out on data, processor has to keep data blocks nearby or bring it from memory / disk when needed.

Increase in clock frequency implies an increase in data transfer penalty as well. In current era, the time it takes to transfer data blocks from memory dominates the time it takes to for processors to perform calculations on those data blocks. Because of the latency caused by transfer, processors has to be stalled frequently. Fortunately, most applications do not make entirely idenependent memory accesses. In general, memory access patterns express some degree of locality (classified under either temporal or spatial) [2]. Therefore CPU caches along with prefetchers are introduced in an effort to keep data nearby for certain senarios. Substantially reducing average time to access memory.

On the other hand, sequential execution model reaching a point of diminishing returns, paved the way for parallel programming paradigm. With this new paradigm, problems are expressed in terms of smaller sub-problems which are solved simultaneously. The process of dividing a workload into smaller chunks is called task decomposition. Task decomposition takes several metrics into account, such as load balance, data locality, and communication overhead.

Due to certain applications expressing definitive characteristics (such as predictable access patterns, being embarassingly parallel...), specialized hardware structures are developed. In particular, vector extensions such as SSE (Streaming SIMD extensions), operates on chunks of data instead of individual elements in an effort to improve performance. This is referred as SIMD (single instruction multiple data) in Flynn's taxonomy [19, 20].

In this context, scalability of an application is measured by simultaneously running threads at its peak performance. Increasing thread count further after this point reduces overall performance which is measured in GigaFLOPs (one billion floating point operations per second). Scalability depends on application itself and measured on the hardware it's running on. Therefore, effectively using;

- prefetchers and CPU cache to hide latency,
- task decomposition to improve load balance, data locality, and to relax communication overhead, and
- hardware components built in for specific use,

increases the scalability of parallel applications.

1.2 Problem Definition

In computer science alone, linear solvers are used in various, seemingly irrelevant, areas. Whether the aim is to dynamically animate a character while satisfying

certain constraints [16] or to find the best possible placement for millions of circuit elements [17], the need to use a linear solver remains intact.

In an application, the routines that dominate the runtime, form its kernels. Improving the performance of an application kernel, stands for improving the application performance itself. And in linear solvers, the kernel operation is SpMxV (sparse-matrix vector multiplication).

In this study,

- A hybrid storage format to increase the efficient usage of SIMD components,
- A heuristic based algorithm to extract proposed storage format,
- An adaptive & architecture aware runtime scheduling & load balancing algorithms that respect data locality,

are developed.

Techniques implemented in this work are designed to be used for both hypergraph partitioning powered and spontaneous SpMxV operations.

Hypergraph model is used to

- implement cache blocking techniques to reduce the number of capacity misses.
- create
  - elegant task decomposition,
  - data locality awareness in runtime scheduler and load balancer.

Tests are carried out on Intel's brand new Xeon Phi High Performance Computing platform which gathers NoC (network on chip) and NUMA (non-uniform memory access) paradigms in single hardware. However, proposed routines can also be implemented for GPUs (Graphical Processing Units).

1.3 Thesis Organization

The rest of this thesis is organized as follows;

- Chapter 2 provides the definition of terms and abbreviations used in this document, background information on SpMxV, and reviews Xeon Phi High Performance Computing Platform briefly.
- Chapter 3 describes proposed scheduling algorithms and ideas behind their implementations.
- Chapter 4 discusses optimization techniques and proposes a data structure & algorithm to effectively utilize SIMD components for wide spectrum of matrices.
- Chapter 5 presents experimental results and possible areas for future work.
- Chapter 6 summarizes contributions of this work.

Chapter 2

Background

In this chapter,

- definitions of terms and abbreviations
- background information on SpMxV
- basic concepts of and motives behind partitioning algorithms
- an overview of Xeon Phi Architecture

are provided.

2.1 Terms and Abbreviations

Below are the definitions of terms and abbreviations as they are used throughout this document.

- **M, N:** Row, column count of a matrix.
- **SpM:** Sparse-matrix.
- **SpMxV:** Sparse-matrix vector multiplication.
- **DMxV:** Dense-matrix vector multiplication.
- **NUMA:** Non-uniform memory-access.
- **NoC:** Network on Chip commutnication subsystem.
- **Flops:** Number of floating point operations per second.
- **GFlopss:** GigaFlops, main metric used for measuring performance in this study.
- **MKL:** Intel's Math Kernel Library [24].
- **icc:** Intel C Compiler [23].
- **gcc:** GNU C Compiler [22].
- **nnz:** number of non-zero elements in a sparse-matrix.
- **Cache capacity miss:** Cache misses that occur due to cache's insufficient capacity and can be avoided if bigger cache is provided.
- **Cache blocking:** Converting a big computational work, into smallter chunks that can fit into cache to reduce capacity misses. In this work, L2 cache is chosen for cache blocking.
- **Recursive bipartitioning:** Dividing given data structure into two sub-parts recursively until a certain condition is met.

- **P, PE:** Processing element.
- **EX:** Execution context.
- **FIFO:** First in first out.
- **GPU:** Graphical processing unit.
- **GPGPU:** General purpose graphical processing unit.
- **SISD:** Single instruction single data.
- **SIMD:** Single instruction multiple data.
- **ALU:** Arithmetic logic unit.
- **SSE:** Streaming SIMD extensions.
- **Sub-Matrix:** A small part of SpM that can be used simultaneously with other sub-matrices.
- **SMT:** Simulatenous multi-threading.
- **LBA:** Laid back algorithm (a heuristic algorithm to find optimum cut for hybrid JDS-CSR format).

2.2 Background Infomation on SpMxV

A matrix is said to be sparse if the total number of nonzeros is much less than its row and column count multiplied *(M x N)*. In general any number of nonzeros per row/column remains constant. Below in Table 2.1, stats of an SpM, taken from University of Florida sparse-matrix collection [21], are presented.

Table 2.1: Properties of Atmosmodd, it holds only 7 times the memory space of a dense vector with same column count.

atmosmodd	
number of rows	1,270,432
number of columns	1,270,432
nonzeros	8,814,880
max nonzeros per row	7
average nonzeros per row	6.9
min nonzeros per row	4
max nonzeros per column	7
average nonzeros per column	6.9
min nonzeros per column	4

In both SpMxV and DMxV (Dense matrix vector multiplication), throughput is measured in FLOPs (number of floating operations) per second. If both routines were implemented in the same way, effective throughput of SpMxV will be much lower than the throughput of DMxV. Because elements with the value zero doesn't contribute to overall results in any way, the throughput of SpMxV calculated in terms of non-zero elements. Using the same storage format as dense matrix will result in wasted memory, memory bandwidth, and CPU cycles. As a result, sparse-matrices are generally stored in compact data strucutres (only keeping track of non-zero values) which allows traversing non-zeros in a certain way, instead of using traditional 2D array format.

2.2.1 Sparse-Matrix Storage Schemes

There are various storage schemes for sparse-matrices (most fundamental ones are explained in [15]), only 2 of those are implemented for this work and they are stated below.

1. CSR (Compressed Storage by Rows)
2. JDS (Jagged Diagonal Storage)

Both structures facilitate efficient use of sequential access and prefetchers. In Figure 2.1, a sample sparse-matrix with dimensions 8 x 8 is shown in 2D array representation to be converted into CSR, and JDS counterparts in the following two sections.

$$M_{5,5} = \begin{pmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 6 & 2 \\ 0 & 0 & -2 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -4 & 1 & 2 \end{pmatrix}$$

Figure 2.1: Sample matrix in 2D array representation.

2.2.1.1 CSR (Compressed Row Storage) Format

The CSR format consists of 3 arrays; values, columns, and row-ptr. The sparse-matrix in Figure 2.1 is presented in CSR format in Figure 2.2.

index	0	1	2	3	4	5	6	7	8
values	4	-1	6	2	-2	1	-4	1	2
colInd	0	1	3	4	2	0	2	3	4
rowPtr	0	1	4	5	6	9			

Figure 2.2: CSR representation of SpM in figure 2.1.

Given that an SpM has enough non-zero elements per row, CSR scheme can benefit from vectorization. If average non-zero elements per row is significantly smaller than the number of elements that can fit into SIMD unit, vectorization is inefficient since most SIMD slots will be left empty.

2.2.1.2 JDS (Jagged Diagonal Storage) Format

The JDS format is formed by 4 arrays and designed to be used by GPUs and SSE components. Conversion steps of matrix in 2.1 to JDS format is depicted in figures 2.3, 2.4, and 2.5.

$$M_{5,5} = \begin{pmatrix} 0 & -1 & 0 & 6 & 2 \\ 0 & 0 & -4 & 1 & 2 \\ 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 2.3: Matrix rows are sorted by their non-zero count in descending order.

$$M_{8,8} = \begin{pmatrix} -1 & 6 & 2 \\ -4 & 1 & 2 \\ 4 & & \\ -2 & & \\ 1 & & \end{pmatrix}$$

Figure 2.4: All non-zeros are shifted to left.

index	0	1	2	3	4	5	6	7	8
dj	-1	-4	4	-2	1	6	1	2	2
jdiag	1	2	0	2	0	3	3	4	4
idiag	0	5	7	9					
perm	1	4	0	2	3				

Figure 2.5: JDS representation of SpM in Figure 2.1.

In JDS format, much like in CSR, because they are continuous in memory, both y vector entries and dj & jdiag arrays can be brought into cache with single high performance load instruction (vector load). It differs from CSR in that, all non-zero elements are shifted to left regardless of their column indices to create longer chunks of SpM elements which will be traversed in the innermost for loop (see sections 2.4.1.3 and 4.3 for more information). Only x vector entries need additional gather and pack instructions. Also, for JDS, vectorization is more efficient for matrices that have similar amount of non-zeros per row (in this case, matrix assumes more of a rectangle rather than jagged array).

2.2.2 Space Requirements

Algorithms presented in this work aim to make use of data residing in cache and reduce the number of times processor has to go to memory to fetch blocks of data. Because of this, data structures are compact and their space requirements are crucial. The formulas that calculate matrix storage schemes' space requirements, for a single sub-matrix, are given below. Size of X vector entries is discarded because of no particular way to calculate it using rowwise 1-D partitioning algorithm (which is the partitioning algorithm utilized in this study).

- CSR Storage Scheme
  sizeof(REAL) * NNZ + // *values array*
  sizeof(INTEGER) * NNZ + // *colInd array*
  sizeof(INTEGER) * ROW-COUNT // *rowPtr array*
  sizeof(REAL) * Y_VECTOR_LENGTH // *y entries used by sub-matrix*

- JDS Storage Scheme
  sizeof(REAL) * NNZ + // *dj array*
  sizeof(INTEGER) * NNZ + // *jdiag array*
  sizeof(INTEGER) * LONGEST-ROW-LENGTH + // *idiag array*
  sizeof(INTEGER) * ROW-COUNT // *permutation array*
  sizeof(REAL) * Y_VECTOR_LENGTH // *y entries used by sub-matrix*

2.2.3 Task decomposition techniques for SpMxV

Depending on the computation, decomposition may be induced by partitioning the input, output, or intermediate data [1]. In following sections, 3 of the task decomposition schemes for SpMxV are explained.

2.2.3.1 Rowwise 1-D Block Partitioning

As shown in Figure 2.6, output vector Y is partitioned among 4 PEs which resulted in dividing matrix in row slices and broadcasting input vector X to all processing elements. Rowwise 1-D block partitioning algorithm incurs shared reads on X vector on shared memory architectures.

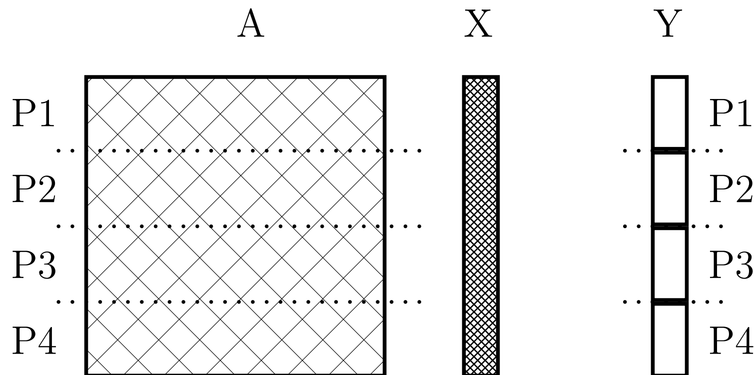

Figure 2.6: Rowwise 1-D Block Partitioning algorithm for 4 PEs.

2.2.3.2 Columnwise 1-D Block Partitioning

Parallel algorithm for columnwise 1-D block partitioning is similar to rowwise, except this time input vector X is partitioned among PEs which resulted in partitioning matrix in column slices and collective usage of output vector Y. Parallel SpMxV using this type of decomposition incurs conflicting writes and must provide a synchronization infrastructure to guarantee the correctness of results. Columnwise 1-D partitioning algorithm is depicted in Figure 2.7.

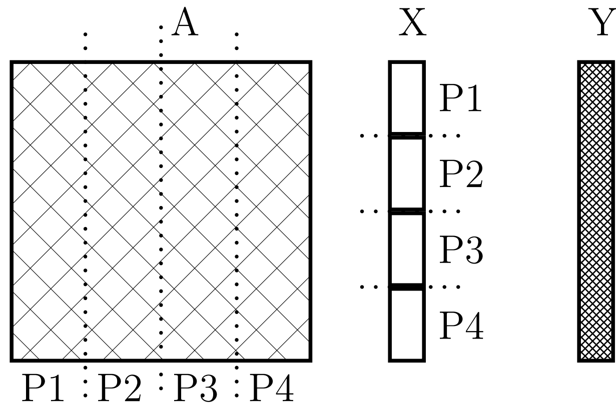

Figure 2.7: Columnwise 1-D Block Partitioning algorithm for 4 PEs.

2.2.3.3 2-D Block Partitioning

Also known as checkerboard, 2-D block partitioning algorithm directly divides given matrix into blocks in way that both input vector X and output vector Y can be partitioned among all PEs. Both shared reads and conflict writes incurred in this decomposition type. Figure 2.8 shows this scheme in action for 16 PEs.

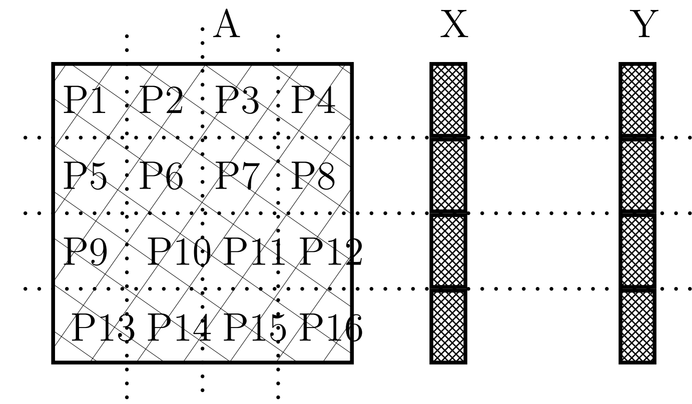

Figure 2.8: 2-D Block Partitioning algorithm for 16 PEs.

It is stated in [1] that DMxV multiplication is more scalable with 2-D partitioning algorithm. In addition, for SpMxV multiplication, cache blocking techniques can be used with checkerboard partitioning.

This work uses only rowwise 1-D decomposition algorithm.

2.3 Partitioning & Hypergraph model explored

In this study, hypergraph partitioning model [11] serves is used to sort matrix rows and columns so that case misses induced by X vector entries are reduced.

In a single iteration of SpMxV multiplication, SpM entries are used only once. On the other hand, dense vector entries are used multiple times. When combined SpM, Y, and X size is bigger than targeted cache size, vector entries can be evicted from and transferred back to cache due to its limited capacity. Hypergraph partitioning model is utilized to order SpM rows and columns such that vector entries are used multiple times before they are finally evicted from cache.

Secondly, in parallel systems where performance can be hindered by communication and uneven work distribution between PEs (processing elements), with the help of hypergraph partitioning model;

- elegant task decomposition which reduces inter-process comminication,
- locality aware scheduling and load balancing algorithms,

can be implemented. PATOH (Partitioning Tool for Hypergraphs) [11] is used throughout this work. Two of PATOH's partitioning models are explained below.

2.3.1 Column-Net Model & Interpretation

In the column-net model [11], matrix columns are represented as nets (hypernodes) and rows as vertices. Ordering & Partitioning decisions are made using cut nets which represent columns that cannot be fully assigned to single PE, therefore has to be shared.

In this work, column-net model is used with rowwise 1-D partitioning algorithm which is previously explained in this chapter. SpM entries on borders incur collective reads on shared memory architectures.

When using column-net interpretation, partitioning a matrix into 2 sub-matrices incurs one column border. In Figure 2.9, sample structure of a sparse matrix partitioned using column-net model is depicted. A1 and A2 are dense blocks (which will be distributed among PEs). B is border which has all the cut nets (columns that cannot be fully assinged to single PE).

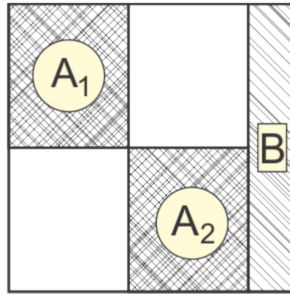

Figure 2.9: Column-net interpretation incurs vertical border.

2.3.2 Row-Net Model & Interpretation

In the row-net model [11], matrix rows are used as nets (hypernodes) and columns as vertices. Ordering & Partitioning decisions are made using cut nets which represent rows that cannot be fully assigned to single PE, therefore has to be shared.

When using row-net interpretation, partitioning a matrix into 2 sub-matrices incurs one row border. In Figure 2.10, sample structure of a sparse matrix partitioned using row-net model is depicted.

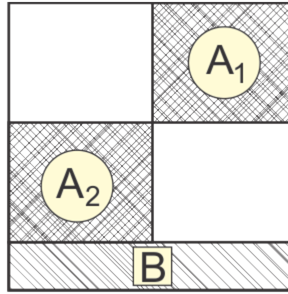

Figure 2.10: Row-net interpretation incurs horizontal border.

Row-net model is not used throughout this work.

2.3.3 Recursive Bipartitioning

PATOH recursively divides a matrix into two sub-matrices until the total size of data structures (required to multiply sub-matrix in question) falls below targeted size (determined by user input). Generally, targeted size is either below or equal to the local cache size of a processor core. This way, number of cache capacity misses are reduced.

Total size of the sub-matrix data structure denpends on the underlying storage format and explained in section 2.2.2

- Rowwise 1-D partitioning algorithm when used with column-net partitioning model & interpretation,
- Columnwise 1-D partitioning algorithm when used with row-net partitioning model & interpretation (not used in this work),

produces better load balance and parallel scalability.

In Figures 2.11 and 2.12, both algorithms are depicted in action accordingly. Matrices are partitioned using bipartitioning scheme explained earlier in this chapter.

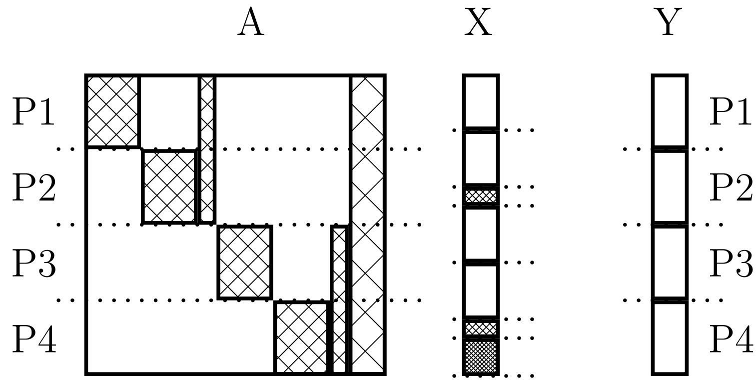

Figure 2.11: A SpM is recursively reordered and divided into four parts using column-net recursive bipartitioning scheme. And sub-matrix to PE assignment is done using rowwise 1-D partitioning algorithm.

As can be seen from Figure 2.11, using ordering, number of shared reads are reduced compared to Figure 2.6 (Shared portion of X vector are weaved denser).

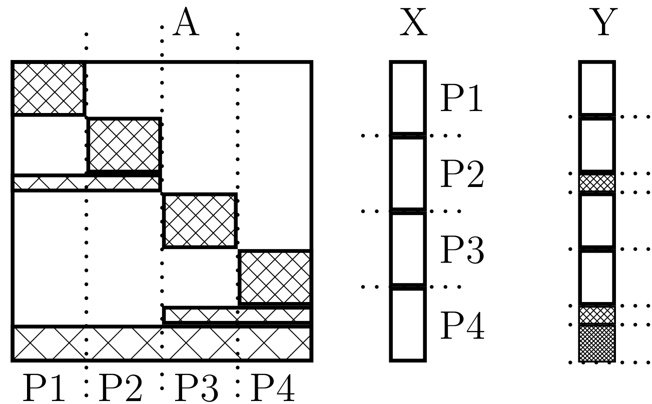

Figure 2.12: An SpM is recursively reordered and divided into 4 parts using row-net recursive bipartitioning scheme. Sub-matrix to processor assignment is done by using columnwise 1-D partitioning algorithm.

Column parallel SpMxV using row net partitioning scheme, as shown in Figure 2.12, reduces the number of conflicting writes (weaved denser), thus the synchronization overhead, compared to SpMxV in Figure 2.7 is minimized.

2.4 A high level overview of Intel's Xeon Phi High Performance Computing Platform

Covering the whole architecture of Xeon Phi cards is out of the scope of this document. Therefore, hardware is briefly inspected and only the parts that are crucial for this study are explained in detail.

Xeon Phi co-processor card [13, 14] is an external and independent hardware that works in conjunction with Xeon processor [25]. It's very similar to GPU in that sense.

Unlike a graphics card, Xeon Phi card has very similar structure to that of a traditional processor, making it easy to program and port existing code written for a conventional processors [3]. Porting is done by Intel's compiler [3, 23] (thus it is required to program the coprocessor). As a result, using the implementations of algorithms which are designed for traditional processors, speed-up can be attained on Xeon Phi card.

Xeon Phi coprocessor is intended for applications where runtime is dominated by parallel code segments. Because Xeon Phi cores have much lower frequency compared to Xeon cores, tasks whose runtime is dominated by serial execution segments can perform better on general purpose Xeon processors [3]. Specifications of these two products used throughout this work are given in are given in chapter 5.

In addition to 60+ cores, Xeon Phi coprocessor cards;

- use directory based cache coherency protocol compared to Xeon's bus based cache coherency protocol.
- has 512bit SIMD vector unit in each core, compared to Xeon's SSE / MMX instructions.
- has GDDR5 memory, compared to DDR3 of Xeon procesor.

2.4.1 5 key features of Xeon Phi Coprocessors

Covering features of Xeon Phi Coprocessors is out of the scope of this document. Here are the 5 aspects of this hardware which carries upmost importance for this work.

2.4.1.1 Number of threads per core

Each core has an inorder dual-issue pipeline with 4-way simultaneous multi-threading (SMT). For applications (except the ones that are heavily memory intensive), each core must have at least two threads to attain all the possible performance from coprocessor. This is the result of coprocessor cores having 2-stage pipeline and described in [3].

2.4.1.2 On die interconnect

All the processing elements on die are connected to each other with a bidirectional ring which uses store and forward communication scheme (making it possible that one or more messages can be on the ring at the same time). It is mentioned in [3] that, because of high quality design of interconnect, data locality beyond a single core (4 threads running on same core) usually doesn't make any difference. Meaning the physical distance between two communicating cores doesn't significantly affect overall performance. See Figure 2.13 for a high level hardware view.

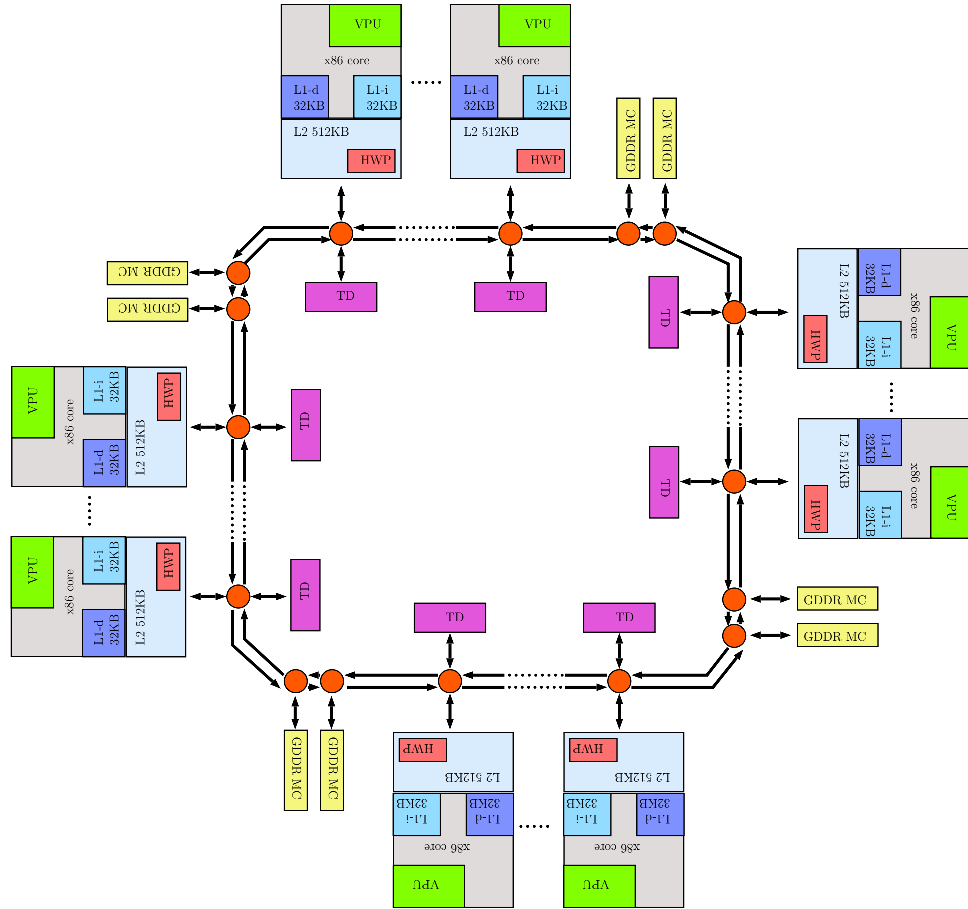

Figure 2.13: High level view of on-die interconnect on Xeon Phi Coprocessor cards. TD (tag directory), MC (Memory Channel), L2 (Level 2 Cache), L1-d (Level 1 Data Cache), L1-i (Level 1 Instruction Cache), HWP (hardware prefetcher).

There are 3 type of bidirectional rings, each of which is used for different purposes and operate independently from one another [27].

- Data block ring: Sharing data among cores.
- Address ring: Send/Write commands and memory addresses.
- Acknowledgement ring: Flow control and coherency messages.

2.4.1.3 Vector Unit

Each core has 512bit wide SIMD instruction unit. Which means 8 double or 16 single precision floating point operations may be carried out at the same time. It can be used in loop vectorization, however as explained in [4] loops must meet the following criterias in order to be vectorized.

1. **Countable:** Loop trip count must be known at entry of a loop at runtime and remain constant for the duration of the loop.
2. **Single entry and single exit:** There should be only one way to exit a loop once entered (Use of breaks and data-dependent exit should be avoided).
3. **Straight-line code:** It is not possible for different iterations to have different control flow, in other words use of branching statements should be avoided (However, there is an exception to this rule if branching can be implemented as masked assignments).
4. **The innermost loop of a nest:** Only the innermost loop will be vectorized (exception being outer loops transforming into an inner loop from through prior optimization phases).
5. **No function calls:** There shouldn't be any procedure call withing a loop (major exceptions being intrinsic math functions and functions that can be inlined).

2.4.1.3.1 Obstacles to vectorization There are certain elements that not necessarily prevent vectorization, but decrease their effectiveness to a point in which whether to vectorize is questioned. Some of these elements (illustrated detailly in [4]) are described below;

1. **Data alignment:** To increase the efficiency of vectorization loop data should be aligned by the size of architecture's cache line. This way, it can be brought into cache using minimum amount of memory accesses.

2. **Non-contiguous memory access:** Data access pattern of an application is crucial for efficient vectorization. Consecutive loads and stores can be accomplished using single high performance load instruction incorporated in Xeon Phi (or SSE instructions in other architectures). If data is not layed out continuously in memory, Xeon Phi architecture supports scatter and gather instructions which allow manipulation of irregular data patterns of memory (by fetching sparse locations of memory into a dense vector or vice-versa), thus enabling vectorization of algorithms with complex data structures [26]. However, as shown in Section 4.3.2, it is still not as efficient.

3. **Data Dependencies:** There are 5 cases of data dependency overall in vectorization.

   (a) **No-dependency:** Data elements that are written do not appear in other iterations of the loop.

   (b) **Read-after-write:** A variable is written in one iteration and read in a subsequent iteration. This is also known as 'flow dependency' and vectorization can lead to incorrect results.

   (c) **Write-after-read:** A variable is read in one iteration and written in a subsequent iteration. This is also known as 'anti-dependency' and it is not safe for general parallel execution. However, it is vectorizable.

   (d) **Read-after-read:** These type of situations are not really dependencies and prevent neither vectorization nor parallel execution.

   (e) **Write-after-write:** Same variable is written to in more than one iteration. Also refered to as 'output dependency' and its unsafe for vectorization and general parallel execution.

   (f) **Loop-carried dependency:** Idioms such as reduction are referred to as loop-carried dependencies. Compiler is able to recognize such loops and vectorize them.

As much as advatageous it may seem, significant amount of code is not data parallel, as a result it is quite rare to fill all of the SIMD slots. Considering SIMD instructions are slower than their regular counterparts, vectorization may deteriorate performance when heavily under-utilized.

2.4.1.4 Execution Models

There are 2 execution models for Xeon Phi co-processors.

1. **Native Execution Model:** In this model, execution starts and ends on co-processor card. This is usually better choice for applications that doesn't have long serial segments and IO operations.

2. **Offload Execution Model:** This is designed for applications with inconsistent behaviors throughout their execution. Application starts and ends on processor, but it can migrate to co-processor in between. Intends to execute only the highly parallel segments on co-processor. Using processor simultaneously along with co-processor is also possible in this model.

2.4.2 Thread Affinity Control

In this work, precise thread to core assignment is crucial for better spatial locality and architecture awareness. As mentioned in [9], there are 3 basic affinity types depicted in Figures 2.14, 2.15, and 2.16. Examples have 3 cores with each core having 4 hardware threads, and 6 software threads in total.

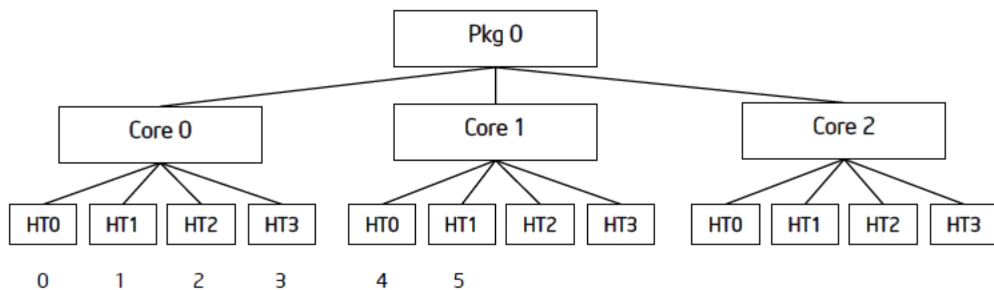

Figure 2.14: Compact thread affinity control.

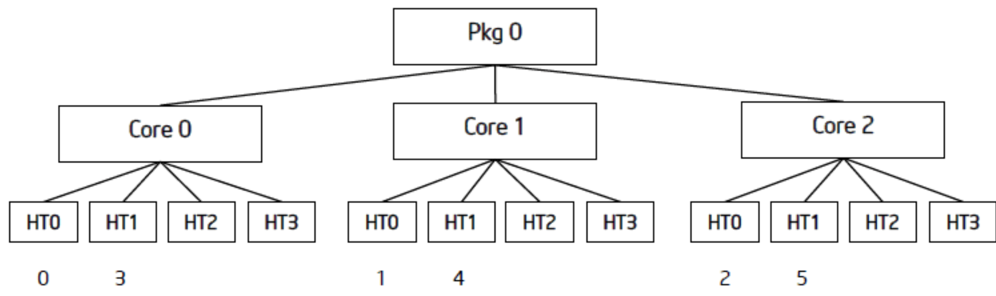

Figure 2.15: Scatter thread affinity control.

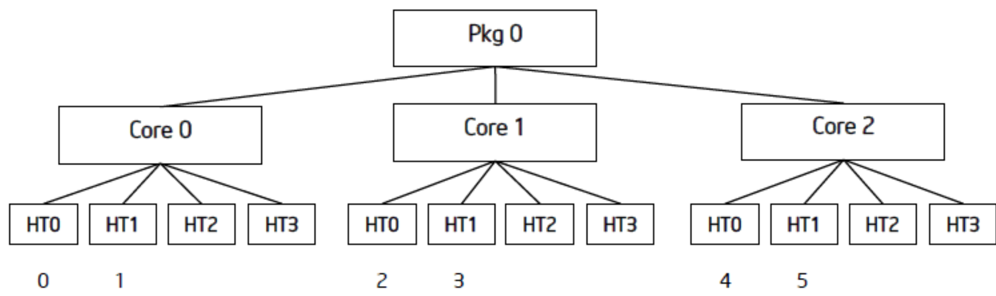

Figure 2.16: Balanced thread affinity control.

Chapter 3

Partitioning, Scheduling, and Load Balancing Algorithms

In this chapter,

- developed SpMxV routines,
- utilized task decomposition strategies, and
- implemented scheduling and load balancing algorithms

are all explained in detail.

All of the SpMxV routines developed for this study uses rowwise 1-D partitioning algorithm and utilizes PATOH's column net model to regulate memory access patterns of SpMs. They are listed below.

- Ordering only routines
    1. Sequential routine
    2. Dynamic OMP Loop routine
- Ordering & Blocking routines
    1. Static routine
        (a) Chunk distribution
        (b) Scatter distribution
    2. OpenMP task routine
        (a) Chunk distribution
        (b) Scatter distribution
    3. Global work stealing routine (GWS)
    4. Distributed work stealing routine (DWS)
        (a) Ring work stealing mode
            i. Chunk distribution
            ii. Scatter distribution
            iii. Shared queue implementation
        (b) Tree work stealing mode
            i. Chunk distribution
            ii. Scatter distribution
            iii. Shared queue implementation

In the following sections, clarifications about those routines on the way they work and the problems they aim to solve are made.

3.1 Ordering Only Routines

Routines presented in this category utilize only the ordering information passed on by hypergraph model.

3.1.1 Sequential Routine

Uses traditional sequential SpMxV algorithm on single Xeon Phi core using only one thread. This algorithm is used as a baseline to calculate speed up of other algorithms and forms application kernel for other routines. C style pseudo code of this routine is provided in Algorithm 4.2 and Algorithm 4.3 for both CSR and JDS formats in order.

3.1.2 Dynamic OMP Loop Routine

This is the parallelized version of sequential routine using OpenMP parallel for pragma. Routine distributes sparse-matrix rows by using OpenMP runtime load balancing algorithms (Dynamic scheduling type is chosen using 32, 64, 128 long trip counts). Schedule types are described in [12].

Problem with using OpenMP runtime load balancing algorithms is that

- they don't use the benefits of cache blocking done by partitioning algorithms. Thus may occasionally disturb data locality.
- they can fail to balance workload in cases where a matrix has dense rows & small row count (latter needs a change in scheduling trip count to get fixed).
- they allow limited control over the order in which SpM entries are traversed. Certain storage schemes cannot be implemented inside OpenMP's parallel for pragma.

However, unlike routines that use blocking, dynamic implementation doesn't have the extra for loop (which is used for traversing SpM row slices - also addressed as sub-matrices) overhead.

3.2 Ordering & blocking routines

In hypergraph partitioning powered use, routines in this category utilize cache blocking techniques along with ordering. Blocking information is either passed on by hypergraph model or created manually in spontaneous mode. Also, the load balancing decisions and initial partitioning in these routines are locality and architecture aware when using hypergraph model. In spontaneous mode, they only consider underlying architecture (physical placement of EXs) before making any scheduling decisions.

An SpM is partitioned using recursive bipartitioning algorithm explained in chapter 2. Then load balancing decisions are extracted from resulting bipartitioning tree. Sample bipartitioning tree and its corresponding matrix view (ordered using hypergraph model), shown in Figures 3.1 and 3.2, are used throughout this chapter for more clear explanation of partitioning and scheduling algorithms.

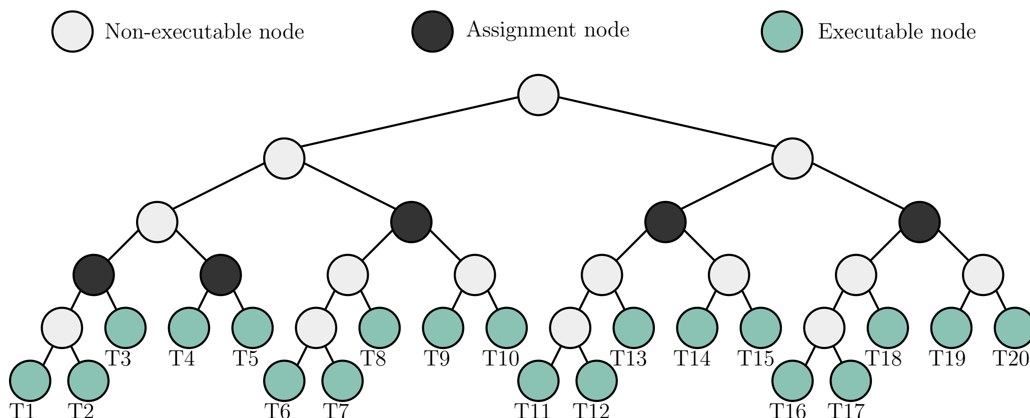

Figure 3.1: Sample bipartitioning tree for column-net interpretation using CSR/JDS schemes on 5 EXs.

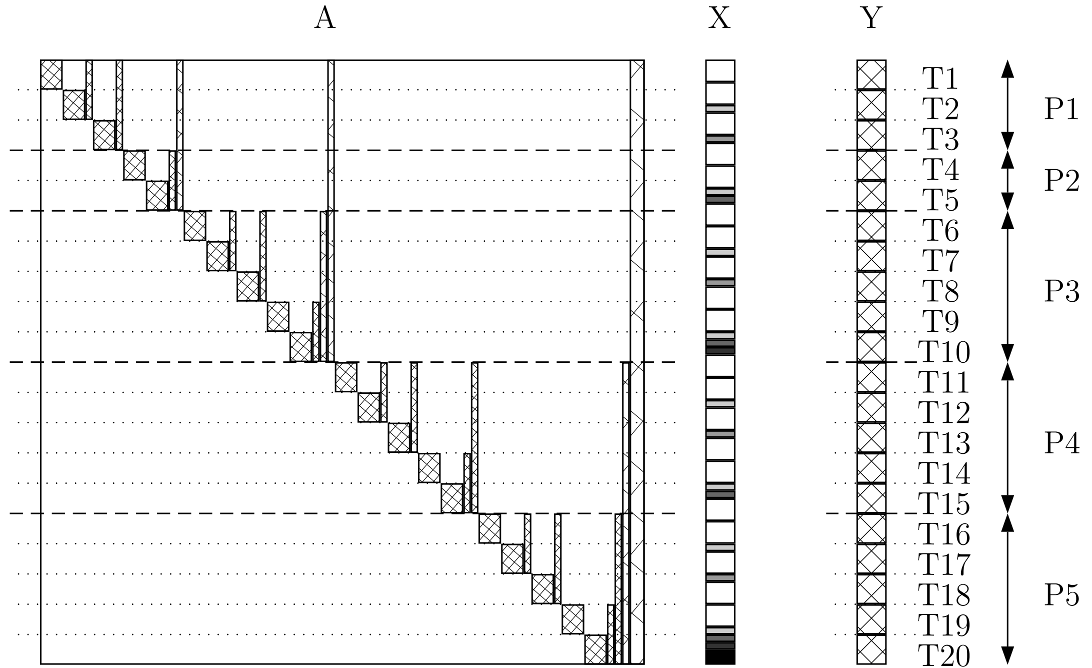

Figure 3.2: Sample matrix ordered by column-net model using rowwise 1-D partitioning algorithm for task to PE assignment. There are 5 EXs and collective reads on X vector by tasks are pointed out by density of portions' color.

As mentioned in chapter 2, column-net interpretation is used with rowwise 1-D partitioning. Roles played by each tree node depends on underlying storage scheme, patitioning algorithm, and execution context count as explained below.

- **Assignment Node** is attached to a PE. All child nodes, connected to this node, are assigned to that PE.
- **Non-Executable Node** contains a sub-matrix whose total space is bigger than targetted size. So, it is continued to be divided and ignored by PEs.
- **Executable Node** , in rowwise 1-D partitioning, contains a sub-matrix whose total space is smaller than targetted size. Therefore, it isn't divided anymore and ready for execution. Additionally, in implementations that have conflicting writes, inner nodes of the bipartitioning tree are also executable and require synchronization framework. In this study, matrices are

distributed by rowwise 1-D partitioning algorithm and underlying storage schemes are CSR and JDS, as a result, only the leafs in Figure 3.1 are executable.

Depicted in Figure 3.1 and 3.2, when using hypergraph model, nodes that have more common parents, share more input dense vector entries (borders). Thus, for each EX, executing groups of nodes that share this trait, will result in better performance and such approach is said to be locality aware.

3.2.1 Chunk & Scatter Distribution Methods

Before describing distribution methods, it is mandatory to define a block. In this study, **Block** is a group of EXs. It can have multiple EXs or single one.

3.2.1.1 Chunk Distribution

In chunk distribution, a block consists of single EX. Assignments to blocks occur as chunks of continuous sub-matrices. For 5 EXs, assignment is the same as the on depicted in Figures 3.1 and 3.2.

3.2.1.2 Scatter Distribution

In scatter distribution, a block can have multiple EXs. It is assumed that EXs on the same block is physically closer to each other than other EXs. Therefore, continuous sub-matrices are scattered among multiple EXs in a block. When execution starts, each EX executes the sub-matrices that share the most X vector entries, at the same time in an effort to improve temporal locality. In Figures 3.3 and 3.4 this distribution method is shown in action for 2 blocks, first containing 3 EXs, while the latter having only 2.

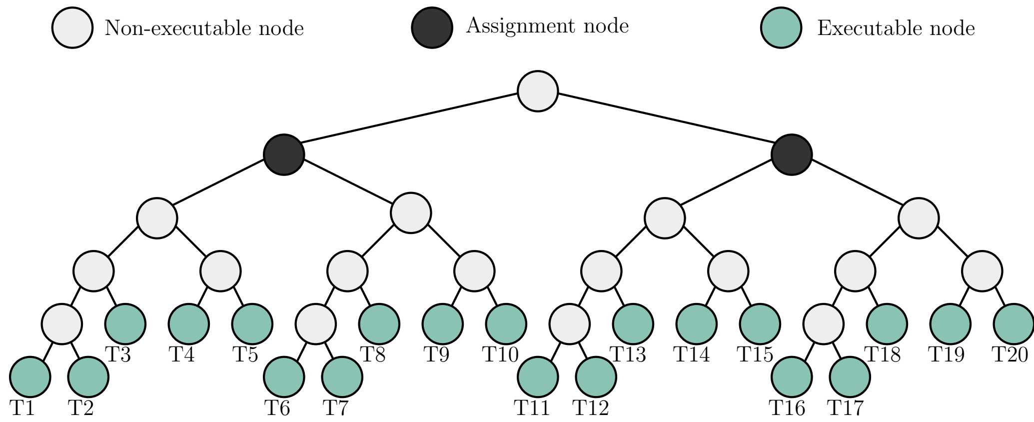

Figure 3.3: Bipartitioning tree for 2 blocks.

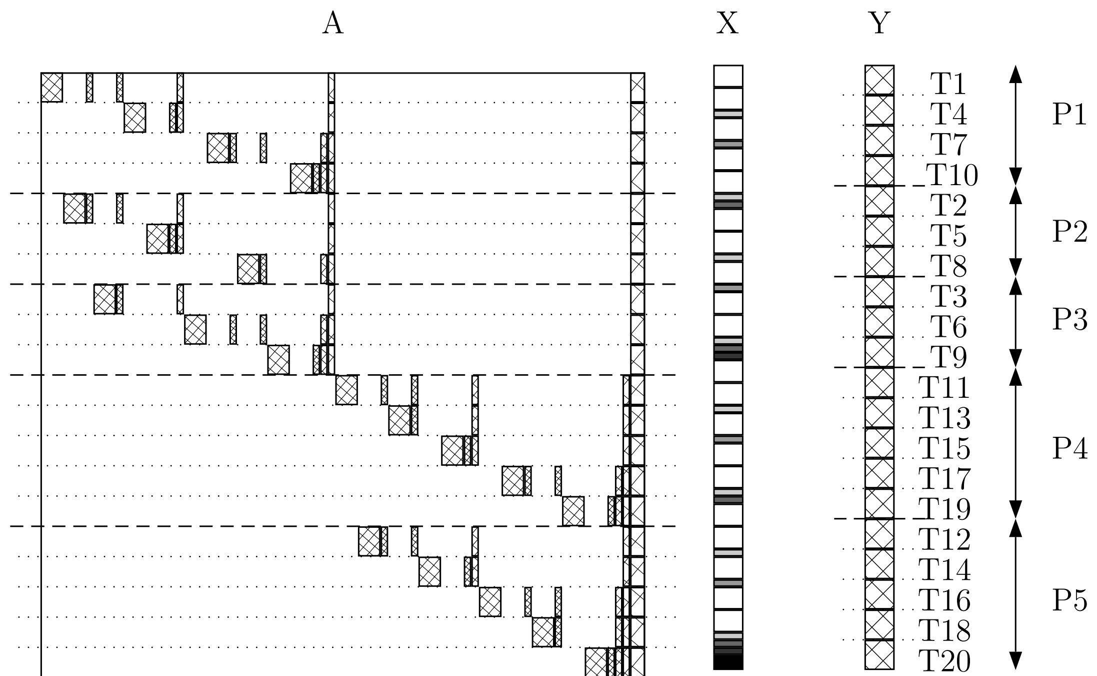

Figure 3.4: Sample matrix ordered by column-net model using rowwise 1-D partitioning using scatter distribution method. Collective reads by task on X vector are shown by the density of potions' color.

3.2.2 Static Routine

This routine makes use of cache blocking techniques to adjust size of sub-matrices which reduces the number of cache capacity misses. Figure 3.5 shows how initial work distribution is done in block parallel algorithm.

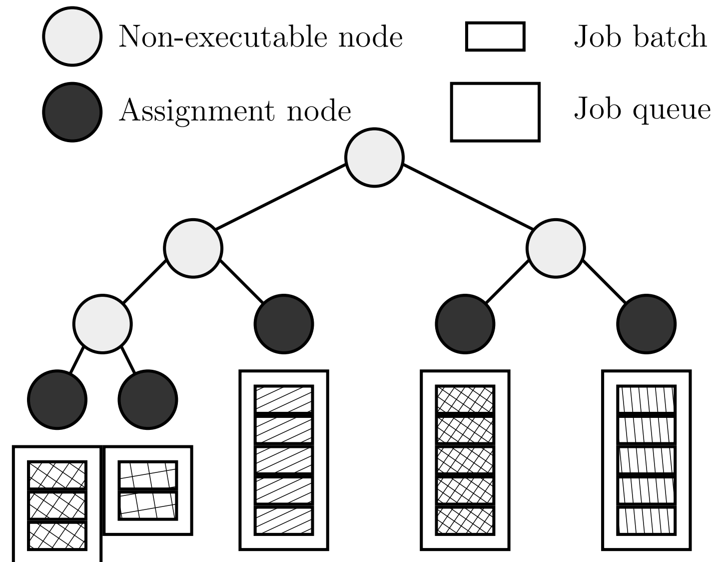

Figure 3.5: Static routine sub-matrix distribution among 5 execution contexts.

When used with hypergraph partitioning, tasks who share more borders (X vector entries) are assigned to a single PE as shown in Figures 3.2 and 3.5.

The downside of static routine is that it only uses initial work distribution to balance load. Throughout execution no scheduling decisions are made which causes load imbalance as shown in Figure 3.5.

3.2.3 OpenMP Task Routine

Tries to improve static routine by adding a dynamic load balancing component. After an EX finishes its share of load, it looks to steal from other EXs and choses the victim randomly. Since this routine is implemented using omp task pragma, control of the order in which sub-matrices are executed and the victim choice is left to OpenMP runtime.

Aside from locking schemes used by OpenMP runtime, this routine tries to improve load balance without destroying cache blocks. In this work, victim choice can improve performace (in both hypergraph partitioning and spontaneous modes). However, lacking a way to control execution order, steal count (how many matrices to steal at once), and choosing the victim randomly this methods doesn't allow further tuning.

3.2.4 Global Work Stealing Routine (GWS)

GWS uses a single FIFO job queue, accessed multiple times by and EX, as a means of dynamic load balancing structure. In GWS, all the sub-matrices are stored in a global queue, and each EX takes the first sub-matrix from queue as they finish the sub-matrix they are currently working on. Queue is protected by a single spin lock which has to be obtained in order to make a change in its state. In Figure 3.6 the way sub-matrices line up in global queue is depicted.

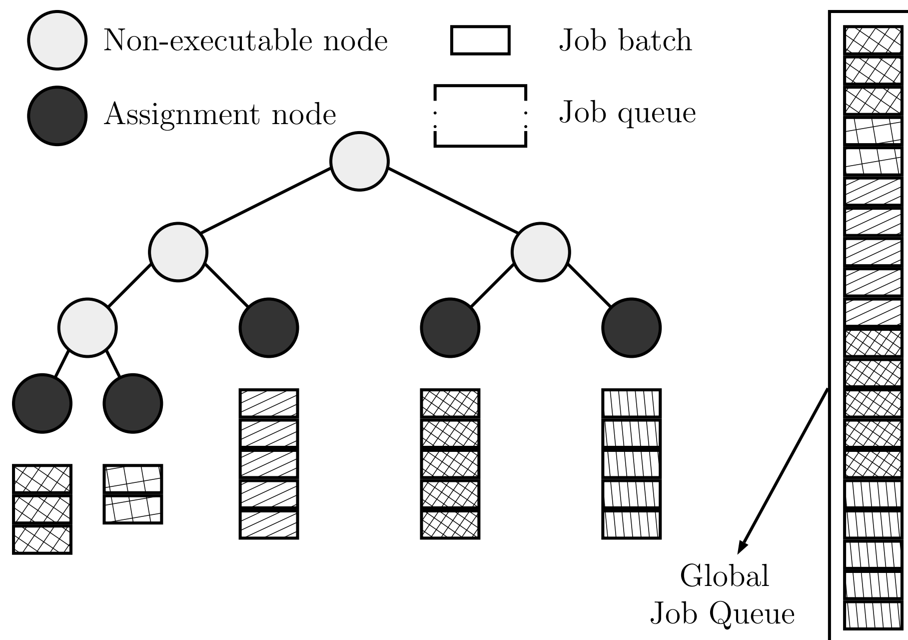

Figure 3.6: Global queue and the order in which sub-matrices line up.

Although using a global job queue provides perfect load balance, it also limits scalability since all EXs are racing to obtain the same lock. It can also be argued that it destroys data locality of initial work distribution.

3.2.5 Distributed Work Stealing Routine (DWS)

Instead of a single FIFO queue, this algorithm keeps multiple FIFO queues, 1 per EX, each protected by its own lock. A block can have either 1 or more EXs depending on different DWS implementations (See Chapter 4). Initial state of the queues before execution are same as Figure 3.5. To preserve data-locality in victim queues, successful steal attempts to a queue always removes tasks from the back. Not front, where owner of the queue detaches tasks for itself.

Stealing in this routine happens in 2 ways.

3.2.5.1 Ring Work Stealing Scheme

This scheme is designed to make better use of ring based communication interconnect and it is locality aware in a sense that it checks for nearby cores first. In Figure 3.7, it is shown in action.

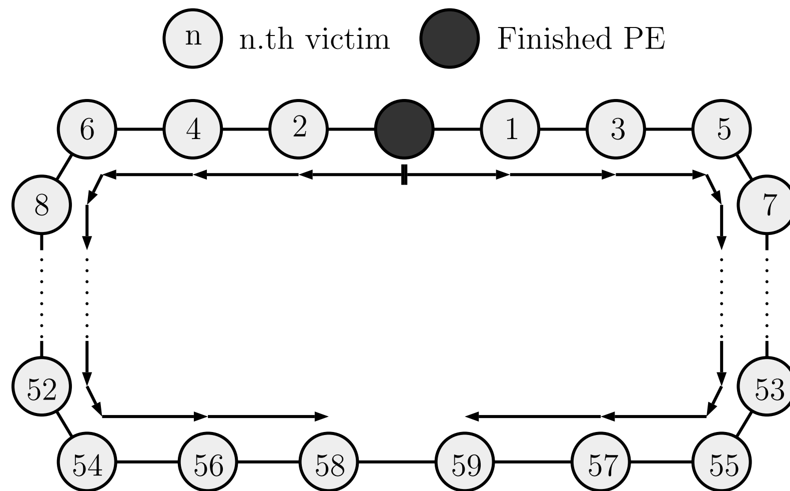

Figure 3.7: Ring work stealing scheme making efficient use of bidirectional communication ring for 60 PEs.

True data-locality awareness in ring stealing scheme comes from the hypergraph partitioning phase. Because of the way sub-matrices are distributed, there

is a strong chance that nearby EX carry sub-matrices using more common input dense vector entrice compared to a distant EX.

This algoirthm is also architecture aware since each EX prefers steal attempts on nearby EXs (in terms of physical placement) over distant ones in an effort to relax communication volume.

3.2.5.2 Tree Work Stealing Scheme

This scheme is more aggressive in a sense that it tries harder to steal the sub-matrices with more common input dense vector entries. It uses bi-partitioning tree to look for victims without any concerns for on-die interconnect. Figure 18 shows this scheme in action.

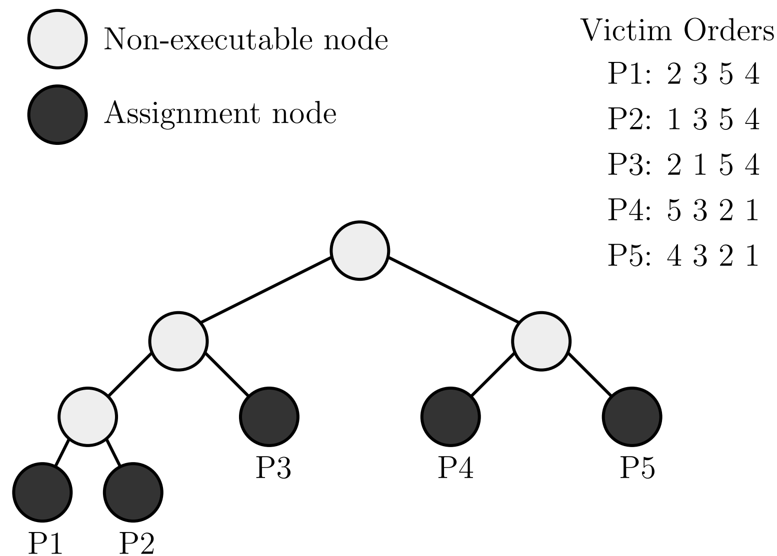

Figure 3.8: Tree work stealing scheme in action.

This algorithm too is locality aware since it prefers stealing sub-matrices with more common borders first.

Compared to GWS, DWS algorithm is more scalable because, contention for each EXs' lock is much less compared to global lock contention of GWS. On the

downside, EXs still have to go through a lock for accessing their local queues which limits application scalability.

In distributed schemes, after a victim is chosen, last half of the tasks in its local queue are stolen. However, execution contexts don't perform steals on queues which has less than certain number of entries. This is called steal treshold and is an adjustable parameter.

3.2.5.3 Shared Queue Implementation

Everything is same with chunk distribution except in shared queue implementation blocks have more than one EX sharing the same job queue and stealing occurs between blocks. The first EX that spots the job queue is empty, will look to steal from other blocks while other EXs in the same block stalls. After a successful steal attempt, EX that stole sub-matrices will get itself a single sub-matrix and free synchronization units for others to continue execution.

Much like scatter distribution, EXs on the same block are assumed be closer in terms of physical placement and temporal data locality is tried to be exploited. This implementation, however, is more strict from scatter distribution in that data-locality is restricted to single block until that blocks sub-matrices are all executed. After that, stealing can be accomplished according to both ring and tree work stealing designs.

This implementation of DWS is designed to be used with hypergraph partitioning which is employed to regulate memory access pattern of SpMs.

Chapter 4

Implementation deltails, Porting, and Fine tuning

In this chapter;

- high level execution course of application,
- detailed analysis of application kernels,
- optimization techniques used,
- new hybrid storage scheme for sub-matrices and an algorithm to extract it

are presented.

4.1 Test sub-set

Peak GFlops achieved for each proposed routine on test SpM set of this work are given in chapter 5. Because there are more than 200 test results, to demonstrate the effects of optimizations documented in this chapter, small but diverse sub-set of 15 matrices are chosen. Below in Table 4.1, stats of these matrices are presented.

Table 4.1: Stats of choosen data sub-set. Row & column count, NNZ, and max-avg-min NNZ per row/column are given.

				row			column		
Matrix	rows	columns	nnz	min	avg	max	min	avg	max
3D_51448_3D	51448	51448	1056610	12	20.5	5671	13	20.5	946
3dtube	45330	45330	3213618	10	70.9	2364	10	70.9	2364
adaptive	6815744	6815744	20432896	1	3	4	0	3	4
atmosmodd	1270432	1270432	8814880	4	6.9	7	4	6.9	7
av41092	41092	41092	1683902	2	41	2135	2	41	664
cage14	1505785	1505785	27130349	5	18	41	5	18	41
cnr-2000	325557	325557	3216152	0	9.9	2716	1	9.9	18235
F1	343791	343791	26837113	24	78.1	435	24	78.1	435
Freescale1	3428755	3428755	18920347	1	5.5	27	1	5.5	25
in-2004	1382908	1382908	16917053	0	12.2	7753	0	12.2	21866
memchip	2707524	2707524	14810202	2	5.5	27	1	5.5	27
road_central	14081816	14081816	19785010	0	1.4	8	0	1.4	8
torso1	116158	116158	8516500	9	73.3	3263	8	73.3	1224
webbase-1M	1000005	1000005	3105536	1	3.1	4700	1	3.1	28685
wheel_601	902103	723605	2170814	1	2.4	602	2	3	3

Optimizations in this chapter doesn't alter matrix structures in any way. They are, in general, related to (parallel) programming and geared towards effective usage of host hardware resources. To demonstrate their impact more clearly, results are presented in between.

4.2 General template of application behaviour

Explanations of sub-routines ,shown in Algorithm 4.1, are listed below;

1. **CREATE_SUB_MATRICES** procedure divides SpM into smaller parts and its implemention changes with underlying storage scheme.
2. **ASSIGN_TO_EXs** procedure performs the initial assignment has different implementation for each routine explained in chapter 3
3. **ADAPTIVE_WARM_UP** procedure adaptively balances workload between multiple execution contexts and is also differs for routines mentioned in chapter 3.
4. **EXECUTE_KERNEL** procedure performs SpMxV and varies depending on underlying storage scheme and task decompoisition as explained in section 4.3.

Algorithm 4.1 General template of application behaviour.

1: $A \leftarrow sparse_matrix$
2: ▷ Cache size for sub-matrices are calculated using total size of the partial spm strucutre and size of corresponding output vector entries
3: $cacheSize \leftarrow targettedsize$
4: $ex_count \leftarrow total_number_of_execution_contexts$
5: $sub_mtx_list \leftarrow ROWWISE_1D_PARTITION(A, cacheSize)$
6: $initial_assignments \leftarrow ASSIGN_TO_EXs(sub_mtx_list, ex_count)$
7: $assignments \leftarrow ADAPTIVE_WARM_UP(initial_assignments, ex_count)$
8:
9: $ParallelSection$
10: $ex_id \leftarrow ID_of_current_execution_context$
11: $ex_subMatrices \leftarrow assignments(ex_id)$
12: **while** ex-subMatrices is not empty **do**
13: $subMatrix \leftarrow ex_subMatrices.remove()$
14: EXECUTE_KERNEL(subMatrix, x, y)
15: **end while**

4.3 Application Kernels

Impelementation details and aplication kernels for CSR and JDS formats are given in next two sections.

4.3.1 CSR format

In CSR implementation, sub-matrix structure composed only of a descriptor which includes, starting row index, row count, starting column index, and column count of sub-matrix itself.

Task decomposition depicted in Figure 4.1 occurs in row slices and during multiplication process global SpM structure is used with sub-matrix descriptors.

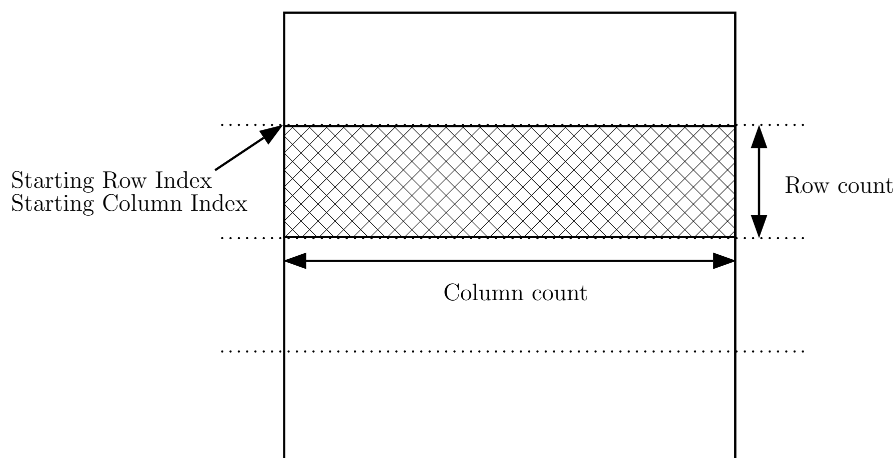

Figure 4.1: Task decomposition for CSR format.

Kernel function for CSR format is given in Algorithm 4.2. For SpMs having only a few average non-zero elements per row (smaller than SIMD length), CSR scheme suffers from vectorization since most of SIMD slots will be left unutilized. However, this scheme will benefit from vector component considering SpM rows are dense enough to effectively fill SIMD slots.

Algorithm 4.2 Kernel function for CSR format.

```
1: function CSR_KERNEL(csr, x, y)
2:     for i = 0; i ≤ csr.rowCount; ++i do
3:         sum ← 0
4:         for j = csr.rowPtr[i]; j ≤ csr.rowPtr[i + 1]; ++i do
5:             sub ← sum + csr.values[j] * x[csr.colInd[j]];
6:         end for
7:         y[i] ← sum;
8:     end for
9: end function
```

4.3.2 JDS format

As depicted in Figure 4.2 task decomposition is implemented as row slices. However, in addition to a descriptor, sub-matrix strucutres also has a part of SpM stored in JDS format (partial JDS). And during multiplication, this structure is used instead of global SpM.

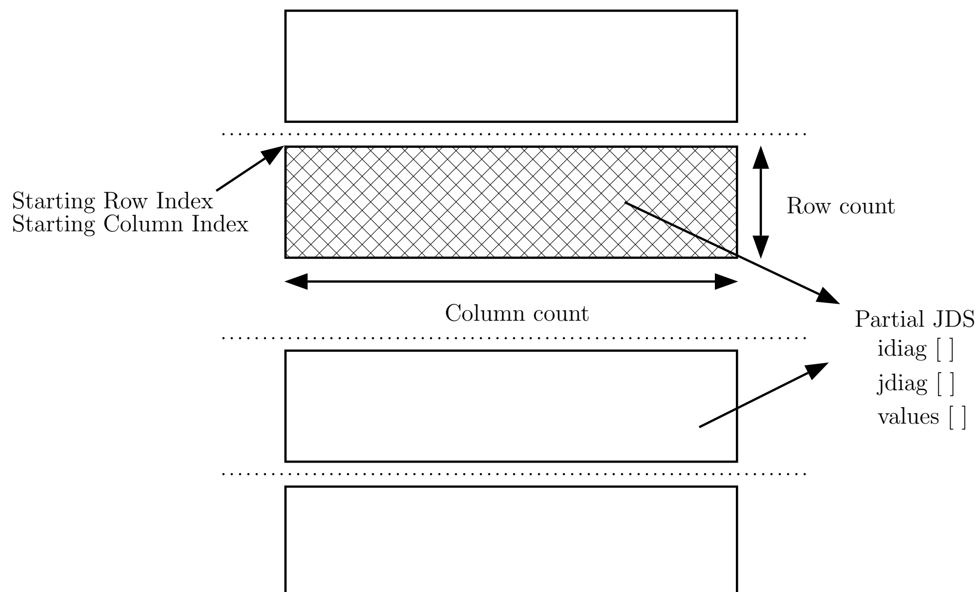

Figure 4.2: Task decomposition for JDS format.

Kernel function for JDS format is given in Algorithm 4.3. As mentioned in section 2.2.1, for well behaved matrices, JDS format can be vectorized efficiently. On the other hand, for SpMs that have occasional dense rows and significantly low average non-zero count per row, vectorization becomes inefficient. Because vectorization is carried out by unrolling the innermost loop of JDS kernel, loop will most likely have very few iterations when traversing dense rows. Therefore, the mojority of SIMD slots will be left empty.

Algorithm 4.3 Kernel function for JDS format.

1: **function** JDS_KERNEL(jds, x, y)
2:     **for** $i = 0; i \leq jds.idiagLength; ++i$ **do**
3:         **for** $j = jds.idiag[i]; j \leq jds.idiag[i+1]; ++j$ **do**
4:             $rowIndex \leftarrow j - jds.idiag[i];$
5:             $y[jds.perm[rowIndex] += jds.dj[j] * x[jds.jdiag[j]];$
6:         **end for**
7:     **end for**
8: **end function**

Application kernel for JDS can be further optimized by making memory accesses to y vector sequential. In optimized implementation permuataion array in JDS structure is eliminated by one time sort on y vector at the end of multiplication phases. Performance differences regarding the update are given in Table 4.2.

Table 4.2: Effect of discarding permutation array from JDS format. Results are measured in GFlops and belong to single precision spontaneous static routine with 32KB sub-matrix sizes. Gains are substantial for matrices that are efficiently vectorizable.

Matrix	jds_perm (GFlops)	jds_sort (GFlops)
3D_51448_3D	0.30	0.33
3dtube	1.64	1.97
adaptive	3.63	4.20
atmosmodd	13.89	16.48
av41092	0.67	0.34
cage14	5.47	6.33
cnr-2000	3.79	4.65
F1	2.58	2.90
Freescale1	3.00	3.23
in-2004	1.71	1.87
memchip	7.09	3.54
road_central	0.65	0.67
torso1	0.31	0.32
webbase-1M	0.26	0.28
wheel_601	1.46	0.65

4.4 Runtime Scheduling and Load Balance

4.4.1 Generic Warm-up Phase

Each routine, whether it is ordering only or uses both ordering and blocking, is run 10 times to warm up CPU caches. In this work, SpMxV operation usually takes up much more space than the space provided by CPU caches as a direct result of huge SpM size. Therefore, SpM, X, and Y vector entries cannot reside in cache between runs. However, generally, warm-up operation is not designed for data-sets, but for frequently used data-structures that are small enough to fit into cache, such as job queues. After warm-up stage, application simply settles down on hardware.

4.4.2 Improving Application Scalability

Routines implemented using 'OpenMP for loop' limits programmers' control in many ways as described in Chapter 3. Therefore, they do not allow any more tuning in warm-up phase.

However, routines with hand coded scheduling algorithms can be optimized during warm-up phase. When each routine is executed multiple times, frequently used job-queue data structures, thread descriptors, and other low level primitives such as scheduling data structures are brought into cache so that each routine can settle down.

Routines with dynamic scheduling rely on lock and other synchronization primitives defined by OpenMP library to ensrure correctness of results. Although it depends on routines itself, synchronization overhead introduced by locks is visible in every routine and significantly limits scalability of an application. Locks affect GWS the most because of contenttion caused by all EXs racing to obtain a single lock. As a result, it cannot scale up to 240 threads which is the most number of threads, Xeon Phi model used in this work can simultaneously run.

As for distributed routines, although contention per lock is greatly reduced, PEs still have to go through their own lock to access local queue which significantly hinders performance. There is litte improvement between 180 threads (3 per core) and 240 (4 per core). By discarding these locks and critical sections, hardware threads will not be stalled due to yields caused by them.

Also, locks and other sychronization primitives used by handcoded schedulers are defined at a relatively high level (also called as application-level), which incurs more overhead than sometimes needed (as they are designed for general use) [28].

For SpMxV, all this can be discarded through warm-up phase. Execution starts with initial task decompositions which are defined in Chapter 3. After a run, stolen tasks for each queue are recorded and job queues are reformed using that info. And for the next run, same thing happens on reformed job queues. This phase is repeated for 10 times, where each run building on the one before it. It has been observed that after 6 - 7 runs, job queues reach to an almost stable state, where task groups assigned to PEs, do not change despite actively working scheduler. Consequently, number of times software threads yield due to I/O are reduced and scalability is further enhanced. This is called adaptive warm-up phase, since it displays a form of learning.

4.4.3 Adaptiveness: Reasoning

Previously, when performance was limited by the transistor count (calculation power) of processor, load balance could be defined as ***"equally distributing computations to each processing element"***. However, today, where majority of applications' performance is determined by their memory access pattern, it is almost mandatory to alter the definition by adding ***"and minimizing processing element idle time"***. To further illustrate the point, below in Table 4.3, two matrices' stats are provided.

Table 4.3: Row & Column count, NNZ, and max, avg, min NNZ per row/column are given for as-Skitter and packing-500x100x100-b050.

Matrix	rows	columns	nnz	row			column		
				min	avg	max	min	avg	max
as-Skitter	1696415	1696415	20494181	0	12.1	35455	0	12.1	35455
packing-500x 100x100-b050	2145852	2145852	32830634	0	15.3	18	0	15.3	18

From Table 4.3, it can be seen that as-Skitter has lesser NNZ. It also has smaller row & column count which translates into Y & X vector sizes respectively. Bigger Y vector also means number of writes are higher. However, packing-500x100x100-b050 is more structured since there isn't big difference between max/avg/min NNZ per row & column. Below in Table 4.4 results of 100 SpMxV operations for these matrices are presented for both spontaneous & hypergraph powered uses for Static and DWS routines.

Table 4.4: Time measured (in seconds) for 100 SpMxV operations performed using as-Skitter and packing-500x100x100-b050 with Static and DWS routines both 'sp' (spontaneous) and 'hp' (hypergraph) modes. 'spc' stands for spontaneous mode partition count while 'hpc' for hypergraph mode patition count. In third row, how many times faster packing-500x100x100-b050 executes compared to as-Skitter is shown. (A hybrid storage format, proposed later in this chapter, is used to take these runs).

Matrix	spc	hpc	static		DWS	
			sp	hp	sp	hp
as-Skitter	3959	2606	1.589	0.577	0.531	0.372
packing-500x100x100-b050	8192	4139	0.378	0.279	0.227	0.255
comparison - speedup			**4.20**	**2.06**	**2.33**	**1.45**

Compared to packing-500x100x100-b050, as-Skitter also has much less partition count. However, because of its complex structure, as-Skitter's memory

access pattern cannot be captured (thus require more memory access / data fetch issues from processor). As shown in Table 4.4, despite being much bigger matrix, packing-500x100x100-b050 runs multiple times faster. In spontaneous mode, DWS runs almost 3 times faster than Static routine for as-Skitter. Which means it has severe load imbalance as well.

Problems are said to be memory bound, if the time it takes to complete is primarily determined by memory operations. SpMxV is a memory bound problem as well. And from the experiment above, it can be inferred that for applications which are memory bounded, load balancing doesn't necessarily mean physical load (computation) balance. It can also mean, cache miss count, cache miss rate, memory latency, write cost, read cost, how they are implemented, and so on. Therefore, for SpMxV, load balancing algorithms should also take many other factors into acount.

As stated in previous section, in warm-up phase each run is built on top of the one before it and overall workload is tried to be balanced in a limited time interval. It starts with physical workload sizes, however, because of its adaptive nature, after that, it's not the computation sizes that are tried to be balanced. It is the active runtime for each EX that is tried to be balanced and it includes all the things that are mentioned above.

4.5 Overcoming deficiencies related to storage format

Both CSR and JDS formats are compact and compressed. Therefore, they are naturally optimized to benefit from vectorization using high performance load & store instructions through data alignment and continuous memory accesses. However, loop vectorization occurs when innermost loops in CSR and JDS application kernels (shown in Figures 4.2 and 4.3 can be unrolled. And a loop can be unrolled as many times as there are iterations. In conclusion, iteration count of the innermost loops significantly effects efficiency of vectorization (Other factors to consider for vectorization are described in chapter 2).

In CSR, average/max/min non-zero count per row can be used as a metric to illustrate effectiveness of vectorization, the higher they are the more items to fill ALUs in vector unit. However, max-column non-zero count of a SpM or 'not-so-rare' dense columns can equally effect vectorization of a CSR structure. In such situtations, vector unit will be severely under utilized most of the time, sometimes even operating on only 1 element.

In JDS, similarly, higher average/max/min non-zero count per column can be thought as a possitive sign for vectorization, while high max-row non-zero count of an SpM or 'not-so-rare' dense rows can hinder it.

Above, 'not-so-rare' idiom means occasional but at the same time not occasional enough to keep avg non-zero per row/column high. When vectorization unit is severely under-utilized, depending on the architecture and the length of vectorization unit, computations are carried out much slower compared to regular SISD (single instruction single data) instructions. This is the direct result of vector instructions being slower than regular ones, however they are also forgiving and sometimes rewarding when heavily utilized.

To efficiently vectorize even the most ill-behaved matrices, an hybrid storage format and an algorithm to how to extract it is presented in the following two sections.

4.5.1 Hybrid JDS-CSR storage format

There are various SpM storage formats [6, 7, 10, 15]. One perfect storage format for all types of SpM is still a goal versus reality. Because, they tend to fix certain problems at the expense of performace or some other problems they introduce themselves. This study asks the question, ***"Is it really necessary to put an effort to fit a global SpM into certain data structure?"***.

In current era, parallel programming paradigm is adopted, in which, problems are expressed in terms of smaller sub-problems and solved simultaneously. Moreover, parallelism is expressed in terms of tasks (rather than threads) which helps with scalability and load balancing. As a result, partitioning a global SpM, produces many small sub-matrices with each having different characteristics. Some expressing parents' ill-behaved nature while others having completely different structure.

For the reasons stated above, a partial hybrid JDS-CSR storage format for 'sub-matrices' - 'not for global SpM' is implemented aiming to not only neutralize worst case behaviour, but also attain otherwise lost performance. In Algorithm 4.4, application kernel for this new format is presented. In Table 4.5, peak GFlops achieved by normal CSR, vectorized CSR, JDS, and hybrid JDS-CSR formats are shown. Despite code size growing by a small portion, results show that hybrid sub-matrix format is superior to both CSR and JDS formats. In the following section, how to efficiently extract this hybrid structure is discussed.

Algorithm 4.4 Kernel function for hybird JDS-CSR format.

1: **function** HYBRID_JDS_CSR_KERNEL($subMatrix, x, y$)
2:     **if** subMatrix.hybrid.JDS is not empty **then**
3:         $JDS_KERNEL(subMatrix.hybrid.JDS, x, y)$
4:     **end if**
5:     **if** subMatrix.hybrid.CSR is not empty **then**
6:         $CSR_KERNEL(subMatrix.hybrid.CSR, x, y)$
7:     **end if**
8: **end function**

Table 4.5: Storage format performance (measured in GFLOPs) comparison for single precision SpMxV. For CSR format, the routine that performs best is chosen to include OpenMP Dynamic Loop implementation. For JDS and CSR formats, results belong to DWS routine.

Matrix	CSR-O2 no-vec (best)	CSR-O3 (best)	JDS-O3 (DWS)	Hybrid JDS-CSR (DWS)
3D_51448_3D	9.89	8.29	0.32	13.29
3dtube	13.06	21.84	5.29	20.51
adaptive	5.84	3.01	13.79	14.24
atmosmodd	9.09	5.97	24.78	24.59
av41092	13.02	12.13	1.07	14.49
cage14	10.94	9.65	26.91	26.24
cnr-2000	10.08	8.90	7.76	16.88
F1	13.69	23.24	16.80	22.00
Freescale1	7.15	5.16	16.71	17.48
in-2004	8.83	11.89	3.53	26.20
memchip	8.94	5.31	4.55	15.66
road_central	3.54	1.31	1.42	1.42
torso1	10.40	21.91	0.59	30.30
webbase-1M	5.01	3.55	0.33	6.46
wheel_601	3.33	2.26	2.46	3.40
geo_mean	**8.15**	**7.10**	**3.74**	**13.51**

As shown in Table 4.5, developed hybrid storage format performs more consistently on wider spectrum of matrix structures and has the biggets geometric mean by far. In most cases, it can remove the inefficiency of vectorization and reveal additional performance which is masked in all other routines.

4.5.2 Laid Back Approach

To create a hybrid JDS-CSR sub-matrix that efectively utilizes vector unit, it is crucial to find 'near optimum' point from where a sub-matrix can be splitted into

JDS and CSR formats. Also, finding the best point depends on host architecture's vector unit length and precision of floating point numbers used for SpMxV (to how many elements that can fit into built in vector unit). Considering all these, the issue can be modelled as a bin-packing problem which is NP-hard. Additionally, using such a sollution, even if a decent point is found, it may take a lot of time & resources to find that point.

In Algorithm 4.5 pseudo code for overall hybrid sub-matrix extraction is demonstrated.

Algorithm 4.5 Hybird JDS-CSR sub-matrix extraction.

1: **function** EXTRACT_HYBRID_SUBMATRICES($A, subMtxSize$)
2: $subMatrices \leftarrow ROWWISE_1D_PARTITON(A, targetedSize)$
3: $hybridSubMatrices \leftarrow NULL$
4: $globalOrderingArray \leftarrow NULL$
5: **while** subMatrices is not empty **do**
6: $subMatrix \leftarrow subMatrices.remove()$
7: $nnz \leftarrow extractNonZeros(subMatrix)$
8: $nnz_sorted, ordering_info \leftarrow sortForJDS(nnz)$
9:
10: $cutColInd, cutRowInd \leftarrow LAID_BACK(nnz_sorted)$
11: $hybridSubMatrix \leftarrow CUT(nnz_sorted, cutColInd, cutRowInd)$
12:
13: $hybridSubMatrices.add(hybridSubMatrix)$
14: $globalOrderingArray.add(ordering_info)$
15: **end while**
16: **return** $hybridSubMatrices, globalOrderingArray$
17: **end function**

4.5.2.1 Laid Back Algorithm: Reasoning

Laid back algorithm (LBA) is designed to evaluate whether it is advantageous to execute a sub-matrix as JDS, as CSR, or as both to efficiently use vectorization. It doesn't find the optimal sollution and has a single assumption and a constraint.

Therefore, it can also be thought of as a heuristic function.

The assumption in LBA is that ***"the length of vectorization unit on host architecture is infinite"***.

Considering infinite vector unit length, LBA then finds the fewest number of lines, drawn parallel to X and Y axis, that can traverse all non-zero elements of sub-matrix. There is only one constraint, ***"once changed the direction of line from Y to X (or vise versa) it cannot be changed back"*** (shown in Figure 4.3).

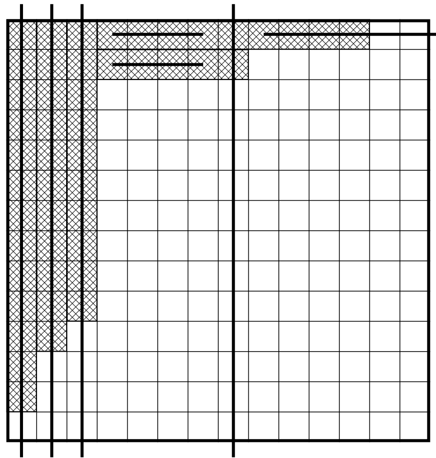

Figure 4.3: Constraint violation in LBA is described.

Since,

- sub-matrix non-zeros are ordered for JDS format,
- and hardware vectorization unit has infinite length,

finding lines by ruleset defined above, is equal to finding an hybrid JDS-CSR format that can be executed by the fewest number of vector instructions, which is also equal to finding a square with the biggest perimeter that can completely fit into 'zero space'. As shown in Figure 4.4, problem turns into seaching for

maximum (x + y). Lines drawn parallel to Y axis will be converted to JDS, and parallel to X axis will be converted to CSR structure.

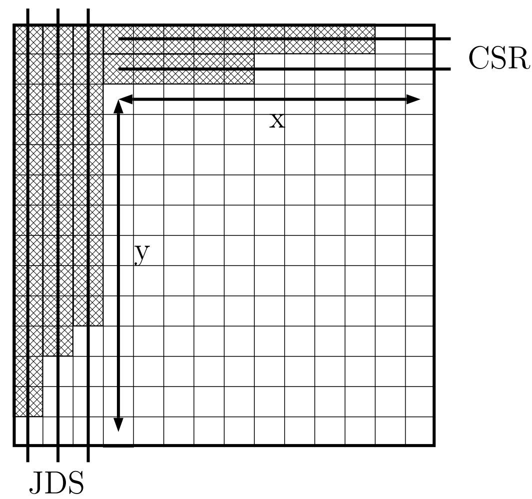

Figure 4.4: LBA in action.

LBA can be implemented using CSR formed much like JDS, JDS, or some structure that keeps non-zero elements ordered in JDS fashion. Therefore, it is not mandatory to previously create storage formats to extract this hybrid structure. However, because of simplicity, JDS and CSR implementation of LAID_BACK algorithm is given in Algorithm 4.6 and 4.7.

Algorithm 4.6 JDS implementation of LBA to find the optimum cut for infinite vector unit length.

```
1: function LAID_BACK_JDS(jds)
2:     cutColumnIndex ← 0
3:     cutRowIndex ← 0
4:     maxPerimeter ← 0
5:     for i = 0; i ≤ jds.idiagLength; + + i do
6:         y ← jds.rowCount − (jds.idiag[i + 1] − jds.idiag[i])
7:         x ← jds.colCount − i
8:         perimeter ← x + y
9:         if maxPerimeter ≤ perimeter then
10:            maxPerimeter ← perimeter
11:            cutColumnIndex ← i
12:            cutRowInd ← jds.rowCount − y
13:        end if
14:    end for
15:
16:    y ← jds.rowCount
17:    x ← jds.colCount − i
18:    perimeter ← x + y
19:    if maxPerimeter ≤ perimeter then
20:        maxPerimeter ← perimeter
21:        cutColumnIndex ← i
22:        cutRowInd ← jds.rowCount − y
23:    end if
24:
25:    return cutColumnInd, cutRowInd
26: end function
```

Algorithm 4.7 CSR implementation of LBA to find the optimum cut for infinite vector unit length.

```
1: function LAID_BACK_CSR(csr)
2:     cutColumnIndex ← 0
3:     cutRowIndex ← 0
4:     maxPerimeter ← 0
5:     for i = 0; i ≤ csr.rowCount; ++i do
6:         x ← csr.colCount − (csr.rowPtr[i + 1] − csr.rowPtr[i])
7:         y ← jds.rowCount − i
8:         perimeter ← x + y
9:         if maxPerimeter ≤ perimeter then
10:            maxPerimeter ← perimeter
11:            cutColumnIndex ← csr.colCount − x
12:            cutRowInd ← i
13:        end if
14:    end for
15:
16:    x ← csr.colCount
17:    y ← csr.rowCount − i
18:    perimeter ← x + y
19:    if maxPerimeter ≤ perimeter then
20:        maxPerimeter ← perimeter
21:        cutColumnIndex ← csr.colCount − x
22:        cutRowInd ← i
23:    end if
24:
25:    return cutColumnInd, cutRowInd
26: end function
```

Despite being identical in number of lines drawn, both JDS and CSR implementation have different results. See Figure 4.5 for the sample output of CSR implementation for same input matrix depicted in Figure 4.4.

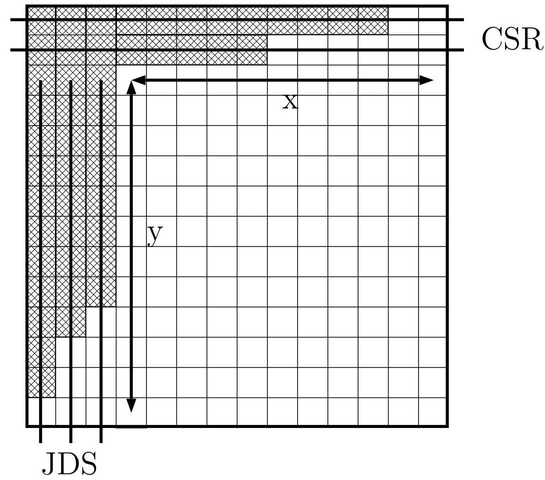

Figure 4.5: Sample output for CSR implementation of LBA.

4.5.2.2 Laid Back Algorithm: Complexity

Worst case complexity of laid back algorithm (for the implementation above) is O(n) where n is the number of columns in a sub-matrix. However, in reality sub-matrices rarely have a row full with non-zeros. For most cases, n is the max number of non-zeros a row has in the sub-matrix. For JDS format, this is equal to the idiag-length and for CSR format it is populated row count.

4.5.3 Possible Improvements & Performance Analysis

There are 2 minor elements that can hinder performance for Hybrid JDS-CSR format, both of which are explained below.

1. Sparse matrix formats are accessed sequentially. So, prefetchers can easily capture their data access patterns. In hybrid format both JDS and CSR are stored seperately. The arrays stored in those formats are allocated in arbitrary memory segments. As a results, application kernel can be intervened by 2 times the stalling prefetch issue for regular storage formats

in order to access those arrays.

2. Application kernel uses both JDS and CSR kernels' code seperately. As a results the code grows bigger with additional for loops.

These are somewhat hazardous side-effects, however they will be the case for every sub-matrix. The former can be addressed by allocating single array structures for both JDS and CSR together and using pointers to access as depicted in Figure 4.6.

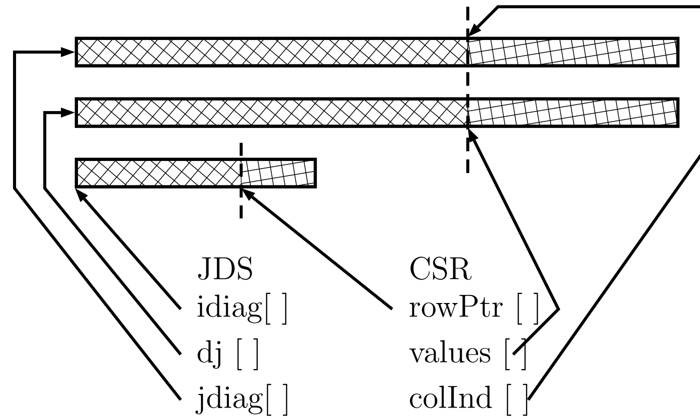

Figure 4.6: Possible improvement for hybrid JDS-CSR format to avoid extra prefetch issues.

However, because of the latter, this sollution will most likely create peel loops as addressed in [3] due to alignement issues. To solve these issues, it is mandatory to create a 'true hybrid storage format' that can process both JDS and CSR alike. These issues are not inspected further in this study.

4.5.4 Choosing optimum partition size and decomposition algorithm

Since each core can run up to 4 threads, initialy 128KB is choosen for each thread to fit in L2 cache (for a total of 512KB). However, for smaller SpMs, partition

count may be less than 240 (which is supported hardware thread count). In this case smaller matrix sizes work better. Additionally, smaller sub-matrix sizes produce better load balance, thus perform faster. But, for matrices which are denser and quiet capable of filling SIMD unit, with rowwise 1-D partitioning algorithm, it is observed that small partition sizes can be restrictive. Most of the time, causing hybrid storage format to act as CSR. More importantly, to ensure SIMD unit slots are filled, the more iterations in inner most loop, meaining bigger sub-matrix sizes are required. Therefore, sub-matrix size is chosen as 64KB per thread (for a total of 256KB).

4.5.5 Effect of Hypergraph Partitioning: Analyzed

As briefly explained in Chapter 2, hypergraph partitioning tries to exploit temporal data-locality for X vector entries by reordering matrix rows & columns. As mentioned in Chapter 2, hypergraph partitioning tool used throughout this study is PATOH [11].

In this section, most of ordering and blocking routines are tested to investigate the effect of data-locality on bidirectional ring based communication subsystem of Xeon Phi. In Table 4.6 results of this test (using hybrid JDS-CSR storage format) is presented.

Choosing the victim or execution order with OpenMP Task routine performs poorly. DWS-Ring routines using chunk distribution method performs consistently better than most other routines. However, for SpMs that benefit from hypergraph partitioning, scatter distribution method or tree work stealing model can perform better (as can be seen in road_central). Also, in one instance (3D_51448_3D), static chunk routine runs faster than DWS chunk routine. This can be caused by either interruptions on warm-up phase, ruined data-locality, or false-sharing. While the third one being more probable, this predicament is worth further investigation.

Since it is the most consistent (load balance), reliable (can use hypergraph

partitioning but doesn't depend on it), and has biggest geometric mean for spontaneous use, DWS routine with chunk distribution model is choosen for further comparisons in Chapter 5.

Table 4.6: Hypergraph partitioning effect is observed on different schedulers. 'sp' stands for spontanenous mode, while 'hp' for hypergraph partitioning mode. 'su' is the speed up of hypergraph partitioning based sequential routine over spontaneous sequential run. Out of 3 tree work stealing routines, only chunk distribution method is tested. Results are presented in GFlops.

Matrix	su	Static				OMP-Task				GWS		DWS							
		Chunk		Scatter		Chunk		Scatter		GWS		Chunk		Scatter		Shared-Queue		Tree W.S.	
		sp	hp	sp	hp	sp	hp	sp	hp	sp	hp	sp	hp	sp	hp	sp	hp	sp	hp
3D_51448_3D	0.97	13.12	14.55	10.12	10.64	10.64	9.81	8.70	11.46	10.73	8.85	13.29	14.09	11.77	12.21	10.47	11.36	10.70	14.66
3dtube	0.96	15.64	18.54	11.30	16.68	10.85	9.30	9.48	8.20	14.27	14.45	20.51	22.03	17.16	16.57	17.83	16.10	20.51	20.37
adaptive	1.09	4.03	6.15	4.07	20.66	5.01	17.12	5.18	5.75	5.68	6.07	14.24	20.91	13.55	20.26	13.67	7.28	6.87	8.60
atmosmodd	0.95	16.78	19.93	15.52	17.88	16.85	13.59	17.28	12.71	23.46	15.94	24.59	19.93	23.39	18.65	23.01	17.25	22.61	20.21
av41092	1.13	14.01	23.34	13.98	22.28	9.15	13.84	10.79	11.46	10.97	11.78	14.49	22.79	12.84	22.21	13.24	18.74	13.48	23.60
cage14	0.64	6.66	5.37	6.24	5.15	22.98	5.66	7.77	5.35	8.16	5.46	26.24	5.83	23.59	5.62	24.09	5.68	9.09	5.83
cnr-2000	1.04	12.40	17.02	11.58	18.01	8.63	10.12	9.01	10.71	13.26	13.53	16.88	16.46	15.80	17.56	16.06	18.11	15.93	16.59
F1	1.71	10.51	21.97	8.83	20.75	19.49	20.36	16.85	18.33	19.04	20.98	22.00	22.55	18.77	21.34	19.10	21.26	16.82	22.38
Freescale1	1.09	5.18	24.67	6.34	23.97	16.15	20.54	15.44	19.91	17.55	23.25	17.48	23.91	17.20	23.37	17.50	23.38	10.33	23.48
in-2004	1.04	13.53	24.51	14.37	24.56	20.54	20.65	18.30	21.04	23.52	25.42	26.20	27.00	25.08	26.08	24.46	26.78	20.22	26.04
memchip	1.04	3.56	5.21	3.79	23.49	4.72	19.01	4.55	18.19	17.49	22.87	15.66	23.43	13.24	23.56	16.63	22.95	10.00	23.86
road_central	2.16	0.67	12.83	0.78	12.68	1.39	5.26	1.41	5.31	1.48	9.97	1.42	5.43	1.69	14.13	1.41	9.21	1.04	10.16
torso1	1.01	21.40	27.32	21.58	25.29	20.03	21.07	20.87	23.01	29.07	27.40	30.30	28.37	28.92	28.50	29.07	26.96	28.98	26.76
webbase-1M	1.05	5.08	17.69	6.38	16.96	6.25	9.31	5.91	8.20	7.22	12.87	6.46	17.49	7.19	17.36	8.04	14.35	6.50	17.23
wheel_601	1.04	2.75	7.86	2.36	5.96	2.54	5.20	2.94	4.92	2.32	5.08	3.40	8.02	3.21	6.05	3.09	7.16	3.28	8.10
MIN		0.67	5.21	0.78	5.15	1.39	5.20	1.41	4.92	1.48	5.08	1.42	5.43	1.69	5.62	1.41	5.68	1.04	5.83
GEO-MEAN		7.27	14.40	7.08	16.10	9.09	11.97	8.31	10.78	10.60	13.05	13.51	16.64	12.72	16.69	12.83	14.76	10.28	16.26
MAX		21.40	27.32	21.58	25.29	22.98	21.07	20.87	23.01	29.07	27.40	30.30	28.37	28.92	28.50	29.07	26.96	28.98	26.76

Chapter 5

Experimental Results and Future Work

In this chapter,

- specifications of hardware used in testing,
- information about data sets,
- details of test environment,
- results of test runs,
- discussions on proposed algorithms' scalability & performance

are provided.

5.1 Test Environment

- Test runs are taken using Xeon Phi native execution model.
- Balanced affinity model (explained in chapter 2) is used for thread to core assignment.

5.2 Hardware Specifications

Tables 5.1, 5.2, and 5.3 show specifications of Xeon and Xeon Phi systems used in tests.

Table 5.1: CPU and memory specifications of Xeon Phi model used in tests.

Xeon Phi CPU and Memory Specifications	
Clock Frequency	1.053 GHz
Number of Cores	60
Memory Size/Type	8GB / GDDR5
Memory Speed	5.5 GT/sec
Peak Memory Bandwidth	352 GB/sec

Table 5.2: Cache specifications of Xeon Phi model used in tests.

Xeon Phi Cache Specifications		
	L1	L2
Size	32 KB + 32 KB	512 KB
Associativity	8-way	8-way
Line Size	64 Bytes	64 Bytes
Banks	8	8
Access Time	1 cycle	11 cycles
Policy	pseudo LRU	pseudo LRU

Table 5.3: CPU and memory specifications of Xeon model used in tests.

Xeon Processor, Memory, and Cache Specifications	
Model	2 x Intel(R) Xeon(R) CPU E5-2643 0 @ 3.30GHz
Clock Frequency	3.30 GHz
Number of Cores	16 (8 x 2 CPUs)
Memory Size/Type	128 GB / DDR3
Memory Speed	1600 MHz
L2 Cache	2 x (4 x 256 KB)
L3 Cache	2 x (10 MB)

5.3 Test Data

Below DWS routines and Math Kernel Library's CSR based cblas functions [24] are compared. Total of 243 SpMs are tested, all of which are taken from [21]. In spontaneous SpMxV for both single and double precision, in 201 instances, default DWS (DWS-RING) algorithm outperformed MKL, and in 42 behind by usually a small margin as shown in. Moreover, it uses blocking information better, thus runs faster utilizing hypergraph partitioning model. Properties of those matrices as well as results, are presented in Tables 5.4 and 5.5 accordingly.

There is only one condition for choosing matrices. They have to have atleast 240 (max Xeon Phi thread capacity) sub-matrices when partitioned. Partition size for a sub-matrix is chosen to be 64 KB as explained in Chapter 4.

Table 5.4: For all the SpMs used to test proposed routines, row & column count, NNZ, min/avg/max NNZ per row/col, and their matrix structures are grouped by their problem types.

Matrix	rows	columns	nnz	minr	avgr	maxr	minc	avgc	maxc	matrix structure
Problem Type: Unknown										
cp2k-h2o-.5e7	279936	279936	3816315	3	13.6	24	3	13.6	27	symmetric
cp2k-h2o-e6	279936	279936	2349567	2	8.4	20	2	8.4	20	symmetric
debr_G_18	1048576	1048576	3145722	1	3	4	0	3	4	symmetric
Problem Type: Undirected Graph										
144	144649	144649	2004137	4	13.9	26	3	13.9	26	symmetric
adaptive	6815744	6815744	20432896	1	3	4	0	3	4	symmetric
as-Skitter	1696415	1696415	20494181	0	12.1	35455	0	12.1	35455	symmetric
belgium_osm	1441295	1441295	1658645	0	1.2	8	0	1.2	9	symmetric
citationCiteseer	268495	268495	2044799	0	7.6	1078	0	7.6	1227	symmetric
coPapersCiteseer	434102	434102	31639338	0	72.9	1188	0	72.9	1188	symmetric
coPapersDBLP	540486	540486	29950972	0	55.4	3299	0	55.4	3299	symmetric
delaunay_n19	524288	524288	2621358	0	5	19	0	5	19	symmetric
delaunay_n21	2097152	2097152	10485664	0	5	23	0	5	23	symmetric
delaunay_n22	4194304	4194304	20971434	0	5	23	0	5	23	symmetric
germany_osm	11548845	11548845	13189517	0	1.1	10	0	1.1	12	symmetric
great-britain_osm	7733822	7733822	8579212	0	1.1	6	0	1.1	7	symmetric
hugetrace-00000	4588484	4588484	9169782	0	2	3	0	2	3	symmetric
hugetric-00000	5824554	5824554	11642492	0	2	3	0	2	3	symmetric
hugetric-00010	6592765	6592765	13178943	0	2	3	0	2	3	symmetric
hugetric-00020	7122792	7122792	14238762	0	2	3	0	2	3	symmetric
human_gene1	22283	22283	24669643	1	1107.1	7939	1	1107.1	7939	symmetric
human_gene2	14340	14340	18068388	1	1260	7229	1	1260	7229	symmetric
italy_osm	6686493	6686493	7341463	0	1.1	8	0	1.1	8	symmetric
m14b	214765	214765	3143271	2	14.6	40	2	14.6	40	symmetric

Continued on next page

Table 5.4 – *Continued from previous page*

Matrix	rows	columns	nnz	minr	avgr	maxr	minc	avgc	maxc	matrix structure
netherlands_osm	2216688	2216688	2665788	0	1.2	6	0	1.2	7	symmetric
packing-500x100x100-b050	2145852	2145852	32830634	0	15.3	18	0	15.3	18	symmetric
pattern1	19242	19242	9323432	9	484.5	6028	9	484.5	6028	symmetric
roadNet-CA	1971281	1971281	3561933	0	1.8	10	0	1.8	10	symmetric
roadNet-PA	1090920	1090920	1992876	0	1.8	8	0	1.8	8	symmetric
roadNet-TX	1393383	1393383	2449937	0	1.8	12	0	1.8	12	symmetric
Problem Type: Undirected Bipartite Graph										
12month1	12471	872622	22624727	1	1814.2	75355	1	25.9	3420	rectangular
Problem Type: Undirected Graph Sequence										
debr	1048576	1048576	3145722	1	3	4	0	3	4	symmetric
Problem Type: Undirected Weighted Graph										
mouse_gene	45101	45101	28967291	1	642.3	8032	1	642.3	8032	symmetric
pdb1HYS	36417	36417	4344765	18	119.3	204	18	119.3	204	symmetric
Problem Type: Undirected Random Graph										
rgg_n_2_18_s0	262144	262144	2832422	0	10.8	31	0	10.8	31	symmetric
rgg_n_2_19_s0	524288	524288	6015244	0	11.5	30	0	11.5	30	symmetric
rgg_n_2_20_s0	1048576	1048576	12734664	0	12.1	36	0	12.1	36	symmetric
rgg_n_2_21_s0	2097152	2097152	26878838	0	12.8	37	0	12.8	37	symmetric
Problem Type: Directed Graph										
amazon-2008	735323	735323	5158388	0	7	10	1	7	1076	unsymmetric
amazon0312	400727	400727	3200440	0	8	10	1	8	2747	unsymmetric
amazon0505	410236	410236	3356824	0	8.2	10	0	8.2	2760	unsymmetric
amazon0601	403394	403394	3387388	0	8.4	10	0	8.4	2751	unsymmetric
auto	448695	448695	6180527	0	13.8	37	1	13.8	37	symmetric
cit-Patents	3774768	3774768	16518948	0	4.4	770	0	4.4	779	unsymmetric
cnr-2000	325557	325557	3216152	0	9.9	2716	1	9.9	18235	unsymmetric
eu-2005	862664	862664	19235140	0	22.3	6985	1	22.3	68922	unsymmetric
flickr	820878	820878	9837214	0	12	10272	1	12	8549	unsymmetric

Continued on next page

Table 5.4 – *Continued from previous page*

Matrix	rows	columns	nnz	minr	avgr	maxr	minc	avgc	maxc	matrix structure
in-2004	1382908	1382908	16917053	0	12.2	7753	0	12.2	21866	unsymmetric
patents	3774768	3774768	14970767	0	4	36	0	4	762	unsymmetric
Stanford	281903	281903	2312497	0	8.2	38606	0	8.2	255	unsymmetric
Stanford_Berkeley	683446	683446	7583376	0	11.1	83448	0	11.1	249	unsymmetric
web-BerkStan	685230	685230	7600595	0	11.1	249	0	11.1	84208	unsymmetric
web-Google	916428	916428	5105039	0	5.6	456	0	5.6	6326	unsymmetric
web-Stanford	281903	281903	2312497	0	8.2	255	0	8.2	38606	unsymmetric
Problem Type: Directed Weighted Graph										
cage12	130228	130228	2032536	5	15.6	33	5	15.6	33	unsymmetric
cage13	445315	445315	7479343	3	16.8	39	3	16.8	39	unsymmetric
cage14	1505785	1505785	27130349	5	18	41	5	18	41	unsymmetric
webbase-1M	1000005	1000005	3105536	1	3.1	4700	1	3.1	28685	unsymmetric
Problem Type: Undirected Multigraph										
kron_g500-logn16	65536	65536	4847260	0	74	17998	0	74	17998	symmetric
kron_g500-logn17	131072	131072	10097678	0	77	28805	0	77	26105	symmetric
kron_g500-logn18	262144	262144	20904300	0	79.7	49163	0	79.7	49163	symmetric
kron_g500-logn19	524288	524288	43038668	0	82.1	80675	0	82.1	80675	symmetric
Problem Type: Bipartite Graph										
IMDB	428440	896308	3782463	0	8.8	1334	0	4.2	1590	rectangular
Problem Type: Semiconductor Device Problem										
matrix_9	103430	103430	2121550	12	20.5	4057	13	20.5	677	unsymmetric
ohne2	181343	181343	11063545	15	61	3441	15	61	3441	unsymmetric
para-9	155924	155924	5416358	8	34.7	6931	8	34.7	6931	unsymmetric
Problem Type: Semiconductor Device Problem Sequence										
barrier2-2	113076	113076	3805068	10	33.7	7031	10	33.7	7031	unsymmetric
barrier2-3	113076	113076	3805068	10	33.7	7031	10	33.7	7031	unsymmetric
para-4	153226	153226	5326228	10	34.8	5776	10	34.8	5776	unsymmetric
Problem Type: Subsequent Semiconductor Device Problem										

Continued on next page

Table 5.4 – *Continued from previous page*

Matrix	rows	columns	nnz	minr	avgr	maxr	minc	avgc	maxc	matrix structure
barrier2-1	113076	113076	3805068	10	33.7	7031	10	33.7	7031	unsymmetric
barrier2-11	115625	115625	3897557	8	33.7	8437	8	33.7	8437	unsymmetric
barrier2-12	115625	115625	3897557	8	33.7	8437	8	33.7	8437	unsymmetric
barrier2-4	113076	113076	3805068	10	33.7	7031	10	33.7	7031	unsymmetric
barrier2-9	115625	115625	3897557	8	33.7	8437	8	33.7	8437	unsymmetric
para-10	155924	155924	5416358	8	34.7	6931	8	34.7	6931	unsymmetric
para-5	155924	155924	5416358	8	34.7	6931	8	34.7	6931	unsymmetric
para-6	155924	155924	5416358	8	34.7	6931	8	34.7	6931	unsymmetric
para-7	155924	155924	5416358	8	34.7	6931	8	34.7	6931	unsymmetric
para-8	155924	155924	5416358	8	34.7	6931	8	34.7	6931	unsymmetric
Problem Type: Optimization Problem										
c-big	345241	345241	2341011	2	6.8	19578	2	6.8	19578	symmetric
exdata_1	6001	6001	2269501	3	378.2	1503	3	378.2	1503	symmetric
gupta1	31802	31802	2164210	3	68.1	8413	3	68.1	8413	symmetric
gupta2	62064	62064	4248286	3	68.5	8413	3	68.5	8413	symmetric
gupta3	16783	16783	9323427	33	555.5	14672	33	555.5	14672	symmetric
ins2	309412	309412	2751484	5	8.9	309412	5	8.9	309412	symmetric
kkt_power	2063494	2063494	14197192	1	6.9	96	0	6.9	96	symmetric
largebasis	440020	440020	5560100	4	12.6	14	4	12.6	14	unsymmetric
mip1	66463	66463	10352819	4	155.8	66395	4	155.8	66395	symmetric
net100	29920	29920	2033200	3	68	181	3	68	181	symmetric
net125	36720	36720	2577200	3	70.2	231	3	70.2	231	symmetric
net150	43520	43520	3121200	3	71.7	281	3	71.7	281	symmetric
net4-1	88343	88343	2441727	2	27.6	4791	2	27.6	4791	symmetric
nlpkkt80	1062400	1062400	28704672	5	27	28	5	27	28	symmetric

Continued on next page

Table 5.4 – *Continued from previous page*

Matrix	rows	columns	nnz	minr	avgr	maxr	minc	avgc	maxc	matrix structure
Problem Type: Thermal Problem										
FEM_3D_thermal2	147900	147900	3489300	12	23.6	27	12	23.6	27	unsymmetric
thermal2	1228045	1228045	8580313	1	7	11	1	7	11	symmetric
Problem Type: Theoretical/Quantum Chemistry Problem										
CO	221119	221119	7666057	15	34.7	313	15	34.7	313	symmetric
Ga10As10H30	113081	113081	6115633	7	54.1	698	7	54.1	698	symmetric
Ga19As19H42	133123	133123	8884839	15	66.7	697	15	66.7	697	symmetric
Ga3As3H12	61349	61349	5970947	15	97.3	1622	15	97.3	1622	symmetric
Ga41As41H72	268096	268096	18488476	18	69	702	18	69	702	symmetric
GaAsH6	61349	61349	3381809	15	55.1	1646	15	55.1	1646	symmetric
Ge87H76	112985	112985	7892195	7	69.9	469	7	69.9	469	symmetric
Ge99H100	112985	112985	8451395	7	74.8	469	7	74.8	469	symmetric
H2O	67024	67024	2216736	14	33.1	37	14	33.1	37	symmetric
Si34H36	97569	97569	5156379	17	52.8	494	17	52.8	494	symmetric
Si41Ge41H72	185639	185639	15011265	13	80.9	662	13	80.9	662	symmetric
Si87H76	240369	240369	10661631	17	44.4	361	17	44.4	361	symmetric
SiO2	155331	155331	11283503	15	72.6	2749	15	72.6	2749	symmetric
Problem Type: 2D/3D Problem										
BenElechi1	245874	245874	13150496	1	53.5	54	1	53.5	54	symmetric
cant	62451	62451	4007383	1	64.2	78	1	64.2	78	symmetric
consph	83334	83334	6010480	1	72.1	81	1	72.1	81	symmetric
cop20k_A	121192	121192	2602982	0	21.5	81	0	21.5	81	symmetric
darcy003	389874	389874	1945496	1	5	7	0	5	7	symmetric
Dubcova3	146689	146689	3636649	9	24.8	49	9	24.8	49	symmetric
ecology1	1000000	1000000	4996000	3	5	5	3	5	5	symmetric
ecology2	999999	999999	4995991	3	5	5	3	5	5	symmetric
helm2d03	392257	392257	2741935	4	7	9	4	7	9	symmetric
mc2depi	525825	525825	2100225	2	4	4	2	4	4	symmetric

Continued on next page

Table 5.4 – *Continued from previous page*

Matrix	rows	columns	nnz	minr	avgr	maxr	minc	avgc	maxc	matrix structure
nd12k	36000	36000	14220946	126	395	519	126	395	519	symmetric
nd24k	72000	72000	28715634	110	398.8	520	110	398.8	520	symmetric
nd3k	9000	9000	3279690	127	364.4	515	127	364.4	515	symmetric1
nd6k	18000	18000	6897316	130	383.2	514	130	383.2	514	symmetric
stomach	213360	213360	3021648	7	14.2	19	6	14.2	22	unsymmetric
torso1	116158	116158	8516500	9	73.3	3263	8	73.3	1224	unsymmetric
Problem Type: Dublicate 2D/3D Problem										
mario002	389874	389874	1945496	1	5	7	0	5	7	symmetric
Problem Type: Circuit Simulation Problem										
ASIC_320k	321821	321821	2635364	1	8.2	203800	1	8.2	203800	unsymmetric
ASIC_320ks	321671	321671	1827807	1	5.7	412	1	5.7	412	unsymmetric
ASIC_680k	682862	682862	3871773	1	5.7	395259	1	5.7	395259	unsymmetric
ASIC_680ks	682712	682712	2329176	1	3.4	210	1	3.4	210	unsymmetric
circuit5M_dc	3523317	3523317	19194193	1	5.4	27	1	5.4	25	unsymmetric
Freescale1	3428755	3428755	18920347	1	5.5	27	1	5.5	25	unsymmetric
G3_circuit	1585478	1585478	7660826	2	4.8	6	2	4.8	6	symmetric
Hamrle3	1447360	1447360	5514242	2	3.8	6	2	3.8	9	unsymmetric
LargeRegFile	2111154	801374	4944201	1	2.3	4	1	6.2	655876	rectangular
memchip	2707524	2707524	14810202	2	5.5	27	1	5.5	27	unsymmetric
rajat21	411676	411676	1893370	1	4.6	118689	1	4.6	100470	unsymmetric
rajat24	358172	358172	1948235	1	5.4	105296	1	5.4	105296	unsymmetric
rajat30	643994	643994	6175377	1	9.6	454746	1	9.6	454746	unsymmetric
rajat31	4690002	4690002	20316253	1	4.3	1252	1	4.3	1252	unsymmetric
Problem Type: Frequency-Domain Circuit Simulation Problem										
pre2	659033	659033	5959282	1	9	628	1	9	745	unsymmetric
Problem Type: Structural Problem										
af_0_k101	503625	503625	17550675	15	34.8	35	15	34.8	35	symmetric
af_1_k101	503625	503625	17550675	15	34.8	35	15	34.8	35	symmetric

Continued on next page

Table 5.4 – *Continued from previous page*

Matrix	rows	columns	nnz	minr	avgr	maxr	minc	avgc	maxc	matrix structure
af_2_k101	503625	503625	17550675	15	34.8	35	15	34.8	35	symmetric
af_3_k101	503625	503625	17550675	15	34.8	35	15	34.8	35	symmetric
af_4_k101	503625	503625	17550675	15	34.8	35	15	34.8	35	symmetric
af_5_k101	503625	503625	17550675	15	34.8	35	15	34.8	35	symmetric
apache2	715176	715176	4817870	4	6.7	8	4	6.7	8	symmetric
bcsstk30	28924	28924	2043492	4	70.7	219	4	70.7	219	symmetric
bcsstk39	46772	46772	2089294	12	44.7	63	12	44.7	63	symmetric
bmw3_2	227362	227362	11288630	2	49.7	336	2	49.7	336	symmetric
bmw7st_1	141347	141347	7339667	1	51.9	435	1	51.9	435	symmetric
bmwcra_1	148770	148770	10644002	24	71.5	351	24	71.5	351	symmetric
Chebyshev4	68121	68121	5377761	9	78.9	68121	9	78.9	81	unsymmetric
crankseg_1	52804	52804	10614210	48	201	2703	48	201	2703	symmetric
crankseg_2	63838	63838	14148858	48	221.6	3423	48	221.6	3423	symmetric
ct20stif	52329	52329	2698463	2	51.6	207	2	51.6	207	symmetric
Emilia_923	923136	923136	41005206	15	44.4	57	15	44.4	57	symmetric
engine	143571	143571	4706073	9	32.8	159	9	32.8	159	symmetric
F1	343791	343791	26837113	24	78.1	435	24	78.1	435	symmetric
F2	71505	71505	5294285	22	74	345	22	74	345	symmetric
Fault_639	638802	638802	28614564	15	44.8	318	15	44.8	318	symmetric
fcondp2	201822	201822	11294316	27	56	96	27	56	96	symmetric
fullb	199187	199187	11708077	18	58.8	144	18	58.8	144	symmetric
gearbox	153746	153746	9080404	6	59.1	99	6	59.1	99	symmetric
halfb	224617	224617	12387821	12	55.2	120	12	55.2	120	symmetric
hood	220542	220542	10768436	21	48.8	77	21	48.8	77	symmetric
inline_1	503712	503712	36816342	18	73.1	843	18	73.1	843	unsymmetric
m_t1	97578	97578	9753570	48	100	237	48	100	237	symmetric
msdoor	415863	415863	20240935	28	48.7	77	28	48.7	77	symmetric
nasasrb	54870	54870	2677324	12	48.8	276	12	48.8	276	symmetric

Continued on next page

Table 5.4 – *Continued from previous page*

Matrix	rows	columns	nnz	minr	avgr	maxr	minc	avgc	maxc	matrix structure
oilpan	73752	73752	3597188	28	48.8	70	28	48.8	70	symmetric
pct20stif	52329	52329	2698463	2	51.6	207	2	51.6	207	symmetric
pkustk10	80676	80676	4308984	30	53.4	90	30	53.4	90	symmetric
pkustk11	87804	87804	5217912	18	59.4	132	18	59.4	132	symmetric
pkustk12	94653	94653	7512317	12	79.4	4146	12	79.4	4146	symmetric
pkustk13	94893	94893	6616827	18	69.7	300	18	69.7	300	symmetric
pkustk14	151926	151926	14836504	12	97.7	333	12	97.7	333	symmetric
pwtk	217918	217918	11634424	2	53.4	180	2	53.4	180	symmetric
s3dkq4m2	90449	90449	4820891	13	53.3	54	13	53.3	54	symmetric
s3dkt3m2	90449	90449	3753461	7	41.5	42	7	41.5	42	symmetric
ship_001	34920	34920	4644230	18	133	438	18	133	438	symmetric
ship_003	121728	121728	8086034	18	66.4	144	18	66.4	144	symmetric
shipsec1	140874	140874	7813404	24	55.5	102	24	55.5	102	symmetric
shipsec5	179860	179860	10113096	12	56.2	126	12	56.2	126	symmetric
shipsec8	114919	114919	6653399	15	57.9	132	15	57.9	132	symmetric
sme3Db	29067	29067	2081063	24	71.6	345	24	71.6	345	unsymmetric
sme3Dc	42930	42930	3148656	24	73.3	405	24	73.3	405	unsymmetric
smt	25710	25710	3753184	52	146	414	52	146	414	symmetric
srb1	54924	54924	2962152	36	53.9	270	36	53.9	270	symmetric
Problem Type: Structural Problem Sequence										
af_shell1	504855	504855	17588875	20	34.8	40	20	34.8	40	symmetric
Problem Type: Subsequent Structural Problem										
af_shell2	504855	504855	17588875	20	34.8	40	20	34.8	40	symmetric
af_shell3	504855	504855	17588875	20	34.8	40	20	34.8	40	symmetric
af_shell4	504855	504855	17588875	20	34.8	40	20	34.8	40	symmetric
af_shell5	504855	504855	17588875	20	34.8	40	20	34.8	40	symmetric
af_shell6	504855	504855	17588875	20	34.8	40	20	34.8	40	symmetric
af_shell7	504855	504855	17588875	20	34.8	40	20	34.8	40	symmetric

Continued on next page

Table 5.4 – *Continued from previous page*

Matrix	rows	columns	nnz	minr	avgr	maxr	minc	avgc	maxc	matrix structure
af_shell8	504855	504855	17588875	20	34.8	40	20	34.8	40	symmetric
af_shell9	504855	504855	17588875	20	34.8	40	20	34.8	40	symmetric
Problem Type: Electromagnetics Problem										
gsm_106857	589446	589446	21758924	12	36.9	106	12	36.9	106	symmetric
offshore	259789	259789	4242673	5	16.3	31	5	16.3	31	symmetric
Problem Type: Computational Fluid Dynamics Problem										
3dtube	45330	45330	3213618	10	70.9	2364	10	70.9	2364	symmetric
atmosmodd	1270432	1270432	8814880	4	6.9	7	4	6.9	7	unsymmetric
atmosmodj	1270432	1270432	8814880	4	6.9	7	4	6.9	7	unsymmetric
atmosmodl	1489752	1489752	10319760	4	6.9	7	4	6.9	7	unsymmetric
atmosmodm	1489752	1489752	10319760	4	6.9	7	4	6.9	7	unsymmetric
cfd2	123440	123440	3087898	8	25	30	8	25	30	symmetric
laminar_duct3D	67173	67173	3833077	1	57.1	89	3	57.1	89	unsymmetric
mixtank_new	29957	29957	1995041	11	66.6	154	11	66.6	154	unsymmetric
parabolic_fem	525825	525825	3674625	3	7	7	3	7	7	symmetric
poisson3Db	85623	85623	2374949	6	27.7	145	6	27.7	145	unsymmetric
PR02R	161070	161070	8185136	1	50.8	92	5	50.8	88	unsymmetric
ramage02	16830	16830	2866352	45	170.3	270	45	170.3	270	symmetric
RM07R	381689	381689	37464962	1	98.2	295	1	98.2	245	unsymmetric
rma10	46835	46835	2374001	4	50.7	145	4	50.7	145	unsymmetric
StocF-1465	1465137	1465137	21005389	1	14.3	189	1	14.3	189	symmetric
Problem Type: Linear Programming Problem										
cont1_l	1918399	1921596	7031999	3	3.7	5	1	3.7	1279998	rectangular
cont11_l	1468599	1961394	5382999	3	3.7	5	1	2.7	7	rectangular
degme	185501	659415	8127528	4	43.8	624079	7	12.3	18	rectangular
neos3	512209	518832	2055024	3	4	6480	1	4	239	rectangular
rail2586	2586	923269	8011362	2	3098	72554	1	8.7	12	rectangular
rail4284	4284	1096894	11284032	2	2634	56182	1	10.3	12	rectangular

Continued on next page

Table 5.4 – *Continued from previous page*

Matrix	rows	columns	nnz	minr	avgr	maxr	minc	avgc	maxc	matrix structure
stat96v2	29089	957432	2852184	97	98.1	3232	1	3	12	rectangular
stat96v3	33841	1113780	3317736	97	98	3760	1	3	12	rectangular
stormG2_1000	528185	1377306	3459881	0	6.6	48	1	2.5	1013	rectangular
Problem Type: Least Squares Problem										
ESOC	327062	37830	6019939	8	18.4	19	0	159.1	12090	symmetric
Rucci1	1977885	109900	7791168	1	3.9	4	30	70.9	108	rectangular
sls	1748122	62729	6804304	1	3.9	4	2	108.5	1685394	rectangular
Problem Type: Model Reduction Problem										
boneS01	127224	127224	6715152	12	52.8	81	12	52.8	81	symmetric
filter3D	106437	106437	2707179	8	25.4	112	8	25.4	112	symmetric
Problem Type: Combinatorial Problem										
ch7-9-b5	423360	317520	2540160	6	6	6	8	8	8	unsymmetric
ch8-8-b5	564480	376320	3386880	6	6	6	9	9	9	rectangular
GL7d15	460261	171375	6080381	0	13.2	38	0	35.5	137	rectangular
GL7d17	1548650	955128	25978098	0	16.8	69	0	27.2	94	rectangular
GL7d18	1955309	1548650	35590540	1	18.2	73	0	23	69	rectangular
GL7d19	1911130	1955309	37322725	3	19.5	121	0	19.1	54	rectangular
GL7d20	1437547	1911130	29893084	0	20.8	395	0	15.6	43	rectangular
GL7d21	822922	1437547	18174775	1	22.1	396	0	12.6	36	rectangular
GL7d22	349443	822922	8251000	2	23.6	403	0	10	26	rectangular
n4c6-b9	186558	198895	1865580	10	10	10	7	9.4	12	rectangular
wheel_601	902103	723605	2170814	1	2.4	602	2	3	3	rectangular

Table 5.5: Performance is measured in GFlops and compared between MKL and default DWS (DWS-RING) routine. 'spc' stands for spontaneous mode partition (/ sub-matrix) count, while 'hppc' for hypergraph-partitioning mode partition count. 'su' is speed up of hypergraph-partitioning use over spontanenous use while running sequential routine. If, for a matrix, in spontaneous mode, DWS routine performs worse than MKL in either single or double precision versions, that matrix's row is highlighted in red, statistics for problem groups are highlighted in blue, and overall statistics in green.

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
Problem Type: Unknown												
cp2k-h2o-.5e7	512	483	1.03	11.58	13.76	16.78	20.68	1.04	10.75	6.25	10.09	10.61
cp2k-h2o-e6	512	304	1.02	7.41	6.73	16.19	18.67	1.00	6.54	5.77	8.45	8.87
debr_G_18	598	448	0.79	3.05	2.65	17.08	10.41	0.77	2.72	2.32	10.55	6.90
Problem Type: Undirected Graph												
144	512	254	1.79	3.76	12.00	5.25	12.06	3.06	2.27	6.53	2.86	6.70
adaptive	4096	2911	1.09	2.66	2.66	14.05	20.51	1.15	1.96	2.27	8.60	11.83
as-Skitter	3959	2606	1.61	3.52	4.57	7.72	11.01	1.63	2.60	3.31	5.30	7.16
belgium_osm	512	291	0.99	1.28	1.23	6.61	15.17	1.21	1.15	1.15	4.50	6.04
citationCiteseer	512	266	2.97	1.47	2.87	1.79	4.42	2.91	1.11	1.90	1.23	2.75
coPapersCiteseer	4114	3889	1.28	18.37	20.11	17.39	26.18	1.27	12.99	14.29	15.17	18.21
coPapersDBLP	4096	3690	1.35	13.48	15.01	14.99	22.21	1.31	9.39	10.72	11.74	13.57
delaunay_n19	512	352	1.10	2.55	5.49	2.95	20.02	1.11	2.19	5.47	5.12	9.45
delaunay_n21	2048	1408	1.11	3.85	3.93	8.66	20.96	1.13	3.17	3.30	7.22	10.77
delaunay_n22	4096	2816	1.16	3.67	3.99	11.18	21.93	1.14	2.79	3.26	8.17	11.19
germany_osm	4096	2315	1.24	1.21	1.33	6.90	13.50	1.31	1.14	1.21	5.05	7.27
great-britain_osm	2048	1520	1.09	1.26	1.24	7.57	12.81	1.27	1.08	1.14	5.36	6.97
hugetrace-00000	2048	1399	1.56	1.65	1.83	5.00	15.23	1.61	1.43	1.59	3.83	8.65
hugetric-00000	4096	1777	1.42	1.65	1.79	6.14	14.95	1.55	1.42	1.58	4.13	8.56

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
hugetric-00010	4096	2011	2.39	1.45	1.69	2.82	14.27	2.30	1.24	1.48	2.18	8.21
hugetric-00020	4096	2172	2.51	1.41	1.69	2.79	14.67	2.36	1.22	1.47	2.17	8.24
human_gene1	4233	3012	1.06	22.17	19.14	28.73	28.96	1.19	12.47	10.78	17.08	17.79
human_gene2	3561	2206	1.04	19.23	15.27	29.95	28.77	1.03	11.17	8.24	20.49	19.78
italy_osm	2048	1305	1.05	1.31	1.30	9.63	12.59	1.09	1.13	1.16	6.16	7.07
m14b	512	397	2.61	2.40	10.53	3.47	14.83	2.54	1.90	7.64	2.99	8.80
netherlands_osm	1024	461	1.02	1.31	1.32	6.78	14.97	1.09	1.16	1.18	4.48	6.89
packing-500x100x100-b050	8192	4139	0.98	10.30	9.33	28.96	25.72	0.95	7.66	6.90	16.27	13.45
pattern1	2046	1140	0.98	16.97	14.73	19.61	20.27	1.01	8.36	7.92	11.40	12.32
roadNet-CA	1024	556	1.05	1.76	1.78	8.84	12.23	1.09	1.54	1.57	5.18	6.33
roadNet-PA	512	310	0.96	1.69	1.72	9.38	13.32	1.05	1.50	1.53	5.12	6.56
roadNet-TX	512	384	0.89	1.69	1.72	9.81	14.01	1.05	1.43	1.50	5.59	6.99
MIN				1.21	1.23	1.79	4.42		1.08	1.14	1.23	2.75
GEO-MEAN				3.13	3.74	8.20	16.06		2.42	2.93	5.77	8.90
MAX				22.17	20.11	29.95	28.96		12.99	14.29	20.49	19.78
Problem Type: Undirected Bipartite Graph												
12month1	2978	2763	1.27	3.02	1.65	8.65	8.95	1.10	1.72	0.97	5.01	5.42
Problem Type: Undirected Graph Sequence												
debr	598	448	0.80	3.04	2.66	17.36	11.21	0.77	2.62	2.30	10.83	6.80
Problem Type: Undirected Weighted Graph												
mouse_gene	4337	3538	1.05	15.77	12.85	20.97	20.88	1.07	9.09	6.73	12.05	10.20
pdb1HYS	1024	533	0.99	24.73	24.92	26.38	26.34	0.99	19.24	19.36	17.07	16.24
MIN				15.77	12.85	20.97	20.88		9.09	6.73	12.05	10.20
GEO-MEAN				19.75	17.90	23.52	23.45		13.22	11.41	14.34	12.87
MAX				24.73	24.92	26.38	26.34		19.24	19.36	17.07	16.24

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
Problem Type: Undirected Random Graph												
rgg_n_2_18_s0	512	362	1.05	8.94	11.81	17.10	19.66	1.01	6.95	6.70	9.73	10.27
rgg_n_2_19_s0	1024	767	1.04	4.97	8.22	19.66	20.94	1.01	3.04	6.46	10.76	11.43
rgg_n_2_20_s0	2048	1619	1.04	4.96	8.70	22.83	24.18	1.01	3.12	6.77	11.29	12.99
rgg_n_2_21_s0	4096	3410	1.02	4.49	8.62	23.23	24.49	1.03	3.07	6.74	11.00	12.76
			MIN	4.49	8.22	17.10	19.66		3.04	6.46	9.73	10.27
			GEO-MEAN	5.61	9.23	20.55	22.22		3.77	6.67	10.68	11.81
			MAX	8.94	11.81	23.23	24.49		6.95	6.77	11.29	12.99
Problem Type: Directed Graph												
amazon-2008	1024	674	1.51	4.84	4.39	7.74	9.11	1.44	3.99	3.53	6.08	5.69
amazon0312	512	416	2.48	2.82	5.93	2.51	9.03	2.36	2.46	4.66	2.77	5.53
amazon0505	512	435	2.59	2.89	6.16	2.60	8.84	2.54	2.58	4.28	2.80	5.52
amazon0601	512	439	2.79	3.09	6.40	2.90	9.43	2.39	2.67	4.82	2.83	5.74
auto	1024	782	3.09	3.37	4.03	3.21	12.33	3.65	2.33	2.95	2.44	8.18
cit-Patents	4096	2247	3.06	1.40	1.78	1.72	4.11	2.79	1.22	1.51	1.48	3.19
cnr-2000	512	412	1.04	8.14	8.49	15.78	16.57	1.03	6.02	6.19	10.24	8.53
eu-2005	4096	2400	1.14	11.58	12.55	21.00	24.63	1.16	8.31	9.09	12.94	14.21
flickr	2045	1251	1.52	3.54	3.72	5.56	6.06	1.59	2.17	2.55	3.23	4.34
in-2004	4095	2150	1.05	10.13	10.52	25.83	27.08	1.03	7.89	8.42	15.78	16.59
patents	4096	2058	2.96	1.38	1.75	1.71	4.07	2.76	1.20	1.48	1.47	3.17
Stanford	478	300	4.39	1.39	1.63	2.09	5.31	4.31	0.87	0.75	1.42	2.63
Stanford_Berkeley	1112	967	1.00	5.54	5.84	13.75	19.83	0.99	3.78	3.97	8.94	12.27
web-BerkStan	1025	969	1.03	8.15	8.52	17.61	22.17	1.02	6.62	7.04	12.84	13.34
web-Google	1024	679	6.93	1.64	5.65	1.60	12.29	6.28	1.47	4.46	1.63	8.86
web-Stanford	512	299	5.50	2.30	6.16	2.50	16.96	5.08	2.18	4.27	2.15	8.69
			MIN	1.38	1.63	1.60	4.07		0.87	0.75	1.42	2.63

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
			GEO-MEAN	3.55	5.00	4.96	11.00		2.80	3.67	3.89	6.86
			MAX	11.58	12.55	25.83	27.08		8.31	9.09	15.78	16.59
Problem Type: Directed Weighted Graph												
cage12	512	257	0.85	10.03	9.63	19.25	12.88	0.83	8.23	6.55	9.27	6.32
cage13	1024	941	0.75	7.51	5.42	22.04	11.60	0.73	5.59	3.96	13.26	7.38
cage14	4096	3403	0.63	8.14	4.81	27.05	11.16	0.64	5.78	3.60	13.70	6.68
webbase-1M	520	441	1.01	2.23	3.03	6.76	17.62	1.05	1.78	2.40	5.51	9.29
			MIN	2.23	3.03	6.76	11.16		1.78	2.40	5.51	6.32
			GEO-MEAN	6.08	5.25	16.69	13.09		4.66	3.87	9.82	7.34
			MAX	10.03	9.63	27.05	17.62		8.23	6.55	13.70	9.29
Problem Type: Undirected Multigraph												
kron_g500-logn16	1002	596	2.18	6.28	0.68	6.75	11.76	1.71	4.23	0.42	4.72	7.24
kron_g500-logn17	1999	1241	1.31	4.84	3.62	5.80	7.21	1.28	2.38	1.89	3.09	3.90
kron_g500-logn18	3928	2567	1.38	2.75	2.95	4.12	5.41	1.09	1.78	1.45	2.88	3.20
kron_g500-logn19	7680	5286	1.17	2.01	1.85	2.98	3.45	1.09	1.59	1.23	2.44	2.63
			MIN	2.01	0.68	2.98	3.45		1.59	0.42	2.44	2.63
			GEO-MEAN	3.60	1.92	4.68	6.31		2.31	1.09	3.18	3.93
			MAX	6.28	3.62	6.75	11.76		4.23	1.89	4.72	7.24
Problem Type: Bipartite Graph												
IMDB	1009	488	3.40	1.38	1.36	1.62	3.78	2.77	1.20	0.98	1.39	2.62
Problem Type: Semiconductor Device Problem												
matrix_9	512	266	0.93	11.86	14.50	12.64	19.10	0.94	7.83	10.17	7.86	11.33
ohne2	2048	1362	1.03	14.07	17.31	17.85	22.74	1.05	9.25	11.88	12.70	13.08
para-9	1024	671	1.21	10.73	18.66	10.67	21.31	1.35	5.33	11.31	6.42	10.68
			MIN	10.73	14.50	10.67	19.10		5.33	10.17	6.42	10.68
			GEO-MEAN	12.14	16.73	13.40	21.00		7.28	11.10	8.62	11.65

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
			MAX	14.07	18.66	17.85	22.74		9.25	11.88	12.70	13.08
Problem Type: Semiconductor Device Problem Sequence												
barrier2-2	512	472	1.22	10.16	15.16	9.31	14.15	1.46	5.12	8.48	5.89	7.71
barrier2-3	512	472	1.29	9.45	16.02	9.20	22.07	1.54	4.84	8.93	5.74	11.70
para-4	1023	660	1.25	10.09	19.11	10.81	18.97	1.36	5.23	10.99	6.60	10.52
			MIN	9.45	15.16	9.20	14.15		4.84	8.48	5.74	7.71
			GEO-MEAN	9.89	16.68	9.75	18.09		5.06	9.41	6.07	9.83
			MAX	10.16	19.11	10.81	22.07		5.23	10.99	6.60	11.70
Problem Type: Subsequent Semiconductor Device Problem												
barrier2-1	512	472	1.29	9.58	16.35	9.04	15.23	1.35	5.57	8.56	6.64	8.01
barrier2-11	515	483	1.27	9.50	14.84	8.67	17.85	1.42	5.07	8.53	6.21	9.21
barrier2-12	515	483	1.22	10.38	14.31	8.55	16.07	1.38	5.30	8.04	5.90	9.40
barrier2-4	512	472	1.25	10.28	18.63	9.17	16.17	1.40	5.49	10.39	5.83	8.41
barrier2-9	515	483	1.34	8.27	14.44	8.22	18.48	1.40	5.07	8.67	6.11	10.79
para-10	1024	671	1.23	10.55	17.95	10.73	20.33	1.42	5.49	10.92	6.78	12.01
para-5	1024	671	1.22	10.19	16.87	11.00	19.20	1.36	5.35	11.22	6.40	10.74
para-6	1024	671	1.21	10.20	18.65	11.12	17.77	1.31	5.57	10.88	6.92	10.59
para-7	1024	671	1.21	10.64	17.81	10.77	18.98	1.33	5.32	10.13	6.49	10.84
para-8	1024	671	1.39	6.80	19.59	9.45	21.37	1.35	5.44	10.91	6.70	12.07
			MIN	6.80	14.31	8.22	15.23		5.07	8.04	5.83	8.01
			GEO-MEAN	9.56	16.85	9.61	18.05		5.36	9.75	6.39	10.12
			MAX	10.64	19.59	11.12	21.37		5.57	11.22	6.92	12.07
Problem Type: Optimization Problem												
c-big	510	307	1.05	3.81	6.13	8.52	16.97	1.15	2.35	4.51	4.99	8.74
exdata_1	489	278	0.98	16.93	13.47	25.31	41.68	0.96	10.75	8.13	17.55	22.10
gupta1	460	267	0.90	3.28	3.16	14.11	16.87	0.85	1.70	1.74	10.79	9.88

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
gupta2	889	523	0.83	3.78	3.62	21.04	18.44	0.81	2.02	1.98	11.99	10.89
gupta3	1762	1140	0.98	10.79	11.21	30.17	28.09	0.96	6.07	6.38	19.63	19.87
ins2	414	355	0.96	0.45	1.22	4.11	4.35	0.94	0.26	0.74	2.32	2.63
kkt_power	2048	1859	0.71	3.19	3.30	13.90	10.08	0.77	2.43	2.68	5.70	6.69
largebasis	1024	706	0.99	13.76	15.48	23.74	26.45	0.96	10.15	9.76	14.28	17.37
mip1	2034	1268	0.95	14.04	17.10	27.66	26.56	0.94	8.55	10.24	17.41	16.87
net100	257	251	0.91	17.01	11.76	11.17	15.58	0.81	9.88	7.30	8.78	9.51
net125	512	317	0.88	15.37	12.31	14.99	14.13	0.75	9.24	7.73	9.78	8.19
net150	512	384	0.79	13.63	12.37	18.43	13.87	0.71	8.09	7.53	10.39	8.77
net4-1	512	304	1.02	1.85	7.87	12.84	14.91	1.01	0.97	4.71	6.99	6.90
nlpkkt80	4096	3569	0.91	15.41	11.90	32.53	25.37	0.92	11.41	8.92	16.97	14.04
			MIN	0.45	1.22	4.11	4.35		0.26	0.74	2.32	2.63
			GEO-MEAN	6.51	7.53	16.30	17.33		3.96	4.74	9.82	10.25
			MAX	17.01	17.10	32.53	41.68		11.41	10.24	19.63	22.10
Problem Type: Thermal Problem												
FEM_3D_thermal2	512	435	0.99	20.48	16.91	22.36	27.86	0.96	14.77	12.95	12.52	14.09
thermal2	2048	1122	1.21	3.97	4.58	8.50	22.58	1.31	3.28	3.76	5.29	11.62
			MIN	3.97	4.58	8.50	22.58		3.28	3.76	5.29	11.62
			GEO-MEAN	9.02	8.80	13.79	25.08		6.96	6.98	8.14	12.79
			MAX	20.48	16.91	22.36	27.86		14.77	12.95	12.52	14.09
Problem Type: Theoretical/Quantum Chemistry Problem												
CO	1024	950	0.94	11.80	10.48	22.13	21.76	0.92	8.46	7.32	12.48	12.01
Ga10As10H30	1024	754	0.96	16.00	14.39	21.14	21.38	0.93	10.20	9.00	12.41	12.46
Ga19As19H42	2048	1093	0.96	17.38	15.48	23.48	22.17	0.91	11.15	9.75	14.20	13.29
Ga3As3H12	1024	733	0.96	14.77	8.79	23.61	22.64	0.95	8.41	4.65	14.56	13.83
Ga41As41H72	4096	2274	0.94	18.08	15.69	26.36	25.23	0.91	11.69	10.14	15.84	14.77

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
GaAsH6	512	417	0.97	11.50	6.13	19.34	22.50	0.94	6.37	3.38	11.96	11.63
Ge87H76	1024	971	0.98	18.15	17.21	22.19	23.21	0.95	11.91	10.75	13.48	13.97
Ge99H100	2025	1039	0.98	19.17	15.98	22.41	21.02	0.98	10.77	9.71	13.30	13.05
H2O	512	275	0.98	14.73	16.38	16.66	17.21	0.95	11.06	12.36	8.62	8.62
Si34H36	1024	636	0.98	16.27	11.56	18.66	21.43	0.94	9.89	6.43	12.06	11.90
Si41Ge41H72	2048	1844	0.94	18.46	17.25	27.78	25.99	0.91	11.48	10.45	15.59	14.64
Si87H76	2048	1317	0.94	14.15	11.92	22.06	20.06	0.91	9.53	8.00	13.44	11.42
SiO2	2048	1387	0.94	13.85	11.48	27.78	25.22	0.93	8.82	7.23	15.84	14.19
			MIN	11.50	6.13	16.66	17.21		6.37	3.38	8.62	8.62
			GEO-MEAN	15.52	12.78	22.35	22.18		9.84	7.96	13.22	12.64
			MAX	19.17	17.25	27.78	25.99		11.91	12.36	15.84	14.77
Problem Type: 2D/3D Problem												
BenElechi1	2048	1621	0.96	22.33	21.92	27.65	28.89	0.96	16.74	16.42	12.78	17.76
cant	512	492	0.96	28.00	27.41	17.28	20.72	0.95	19.48	16.38	13.41	11.32
consph	1024	739	0.97	21.80	22.98	20.44	23.09	0.97	14.86	14.41	14.92	12.88
cop20k_A	512	325	1.74	7.11	12.99	6.85	17.99	1.98	3.73	7.82	5.06	9.84
darcy003	512	262	1.55	3.42	4.88	5.16	21.02	1.64	2.57	5.73	3.40	9.78
Dubcova3	512	453	1.08	5.85	15.79	7.12	23.93	1.10	3.31	11.89	7.46	13.21
ecology1	1024	671	1.00	6.01	4.75	21.88	20.32	1.00	6.11	3.67	13.58	12.61
ecology2	1024	671	0.99	3.68	4.64	22.14	22.22	0.99	4.38	2.95	13.60	13.10
helm2d03	512	359	1.10	2.56	6.11	2.92	24.98	1.12	2.09	7.33	4.75	11.45
mc2depi	512	289	0.99	4.17	3.86	23.29	22.34	0.99	3.81	2.92	11.80	11.17
nd12k	2048	1739	0.97	26.55	26.69	33.51	32.92	0.94	17.90	17.46	22.97	22.22
nd24k	4096	3509	0.96	27.62	27.33	35.58	35.98	0.92	19.13	18.44	23.86	23.57
nd3k	512	401	1.00	23.56	23.07	31.15	41.16	0.99	20.59	21.14	18.73	20.20
nd6k	1024	844	1.00	26.29	25.40	29.26	31.08	0.96	16.65	16.87	19.87	21.03

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
stomach	512	382	1.03	9.64	13.06	20.63	25.11	1.03	9.78	9.30	10.93	12.82
torso1	1694	1046	1.02	14.76	16.14	30.09	28.28	1.02	8.80	10.13	20.22	17.69
			MIN	2.56	3.86	2.92	17.99		2.09	2.92	3.40	9.78
			GEO-MEAN	10.70	12.98	17.12	25.59		8.08	9.60	11.86	14.42
			MAX	28.00	27.41	35.58	41.16		20.59	21.14	23.86	23.57
Problem Type: Circuit Simulation Problem												
mario002	512	262	1.58	3.14	5.11	4.80	17.30	1.66	2.67	4.61	3.41	9.99
ASIC_320k	450	342	1.23	2.24	1.50	5.56	5.65	1.28	1.39	0.89	3.21	3.24
ASIC_320ks	512	243	0.94	4.70	7.29	10.16	16.27	1.03	2.88	5.14	4.86	8.06
ASIC_680k	822	515	0.92	1.74	1.96	4.67	4.58	1.00	1.10	1.22	2.64	2.59
ASIC_680ks	512	325	0.95	2.82	3.61	4.63	17.08	1.05	2.49	2.98	3.79	7.88
circuit5M_dc	4096	2559	1.08	4.32	4.64	16.89	24.72	1.08	3.62	3.83	11.13	13.82
Freescale1	4096	2519	1.10	3.79	4.45	16.34	24.33	1.12	3.36	3.78	9.59	14.08
G3_circuit	2048	1031	1.04	3.90	4.07	14.50	20.55	1.03	3.22	3.45	8.93	12.00
Hamrle3	1024	762	0.97	3.67	3.45	15.92	19.14	0.98	3.00	2.82	10.49	10.81
LargeRegFile	1024	733	0.90	2.71	2.65	17.85	19.28	0.98	2.45	2.45	11.98	10.93
memchip	4096	1974	1.05	4.31	4.61	13.76	22.97	1.08	3.73	3.86	8.31	13.56
rajat21	468	257	0.99	2.35	2.77	4.92	6.66	0.99	1.55	1.73	2.72	3.60
rajat24	484	260	1.01	2.57	3.05	5.55	6.76	1.03	1.63	1.96	2.93	3.89
rajat30	892	794	0.99	2.29	2.29	5.49	5.53	0.97	1.41	1.44	3.22	3.15
rajat31	4096	2767	0.99	3.48	3.54	24.22	24.29	1.02	2.75	2.85	14.42	13.26
			MIN	1.74	1.50	4.63	4.58		1.10	0.89	2.64	2.59
			GEO-MEAN	3.08	3.31	9.74	13.13		2.30	2.47	5.87	7.28
			MAX	4.70	7.29	24.22	24.72		3.73	5.14	14.42	14.08

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
Problem Type: Frequency-Domain Circuit Simulation Problem												
pre2	1024	768	1.01	10.53	5.62	18.89	22.10	0.98	7.50	6.18	11.54	12.98
Problem Type: Structural Problem												
af_0_k101	4096	2174	0.96	19.20	18.16	29.14	30.93	0.97	14.43	13.97	16.04	19.25
af_1_k101	4096	2174	0.96	18.94	18.03	28.98	30.39	0.97	14.45	13.80	16.29	19.11
af_2_k101	4096	2174	0.96	18.95	18.10	28.06	31.03	0.97	14.33	13.77	16.11	18.76
af_3_k101	4096	2174	0.96	19.08	18.18	29.33	31.08	0.97	14.45	13.75	16.36	18.98
af_4_k101	4096	2174	0.96	18.93	18.12	28.93	31.05	0.97	14.44	13.88	16.20	18.86
af_5_k101	4096	2174	0.96	18.94	18.08	29.32	31.29	0.97	14.40	13.77	15.95	19.21
apache2	1024	632	0.97	7.09	5.82	22.68	20.15	0.94	7.26	6.75	13.44	11.38
bcsstk30	256	252	0.99	21.53	23.42	13.25	17.12	0.99	19.29	18.39	11.54	10.14
bcsstk39	512	258	0.98	23.86	26.01	18.29	16.33	0.98	18.50	15.29	10.29	12.03
bmw3_2	2048	1392	1.03	20.09	20.27	23.24	25.99	1.04	14.81	14.83	14.54	15.78
bmw7st_1	1024	905	1.03	20.33	20.45	21.58	25.65	1.01	14.19	14.52	13.02	15.70
bmwcra_1	2048	1309	0.98	21.63	21.48	25.21	22.82	0.98	15.58	15.35	16.31	13.83
Chebyshev4	821	660	0.96	1.35	1.17	17.51	21.02	1.04	0.64	0.60	11.84	11.41
crankseg_1	2048	1299	1.03	25.02	25.02	28.96	29.22	1.05	15.59	16.04	18.45	20.43
crankseg_2	2048	1732	1.04	25.17	26.03	30.72	32.22	1.04	16.39	16.91	20.55	20.75
ct20stif	512	333	1.01	23.18	22.35	16.36	21.55	1.01	17.74	16.97	9.30	11.58
Emilia_923	8192	5062	0.95	18.89	17.17	29.09	26.31	0.95	13.39	12.73	16.20	16.19
engine	1024	584	1.25	8.32	21.97	15.20	22.65	1.28	5.45	14.27	10.02	12.67
F1	4096	3297	1.70	18.18	18.75	22.07	22.23	1.67	12.86	13.93	14.88	16.35
F2	1024	651	1.43	15.04	26.21	15.04	22.99	1.50	7.86	17.04	11.44	13.19
Fault_639	4096	3532	0.98	19.36	18.12	28.93	26.87	1.00	14.00	13.44	14.17	15.80
fcondp2	2048	1392	0.98	22.03	21.56	24.97	25.61	0.98	16.35	15.95	16.41	15.95
fullb	2048	1442	1.07	21.21	21.63	22.70	26.65	1.10	15.55	15.84	16.49	15.89

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
gearbox	2048	1118	0.98	21.02	21.32	20.36	22.67	0.98	15.09	15.19	13.28	14.15
halfb	2048	1526	1.03	21.21	21.43	25.36	28.09	1.03	15.81	15.99	15.29	16.35
hood	2048	1327	1.14	18.99	18.85	19.51	25.08	1.14	14.08	13.92	12.65	15.39
inline_1	8192	1425	1.17	21.48	22.37	26.32	29.33	1.21	14.75	15.29	16.95	17.85
m_t1	2048	1197	0.99	23.97	24.21	28.36	28.69	0.97	16.39	16.18	18.84	18.21
msdoor	4096	2496	1.14	20.23	19.74	23.38	27.61	1.15	14.95	14.84	14.06	17.44
nasasrb	512	331	0.98	22.72	25.20	17.10	21.26	0.98	18.46	17.55	9.45	12.11
oilpan	512	444	0.97	23.74	23.24	18.13	24.99	0.98	18.93	19.72	10.42	14.97
pct20stif	512	333	1.00	23.22	18.71	15.85	20.19	1.02	17.42	18.66	9.43	11.85
pkustk10	1024	531	1.01	23.79	27.04	17.36	23.36	1.01	15.03	15.24	11.33	14.76
pkustk11	1024	643	1.00	22.97	27.84	20.75	23.08	1.01	14.77	16.56	13.59	14.45
pkustk12	1036	923	1.00	15.92	14.43	23.31	25.56	0.99	10.24	8.80	15.56	15.59
pkustk13	1024	814	1.01	20.54	21.09	20.37	22.56	1.01	13.53	13.72	15.04	13.48
pkustk14	2048	1821	0.99	22.51	23.43	27.18	27.77	1.00	16.21	16.28	18.45	17.29
pwtk	2048	1433	0.98	21.80	21.43	26.00	28.31	0.96	16.33	16.04	15.85	16.65
s3dkq4m2	1024	595	0.94	26.04	19.63	19.79	25.55	0.96	19.05	17.11	12.00	15.62
s3dkt3m2	512	464	0.98	21.40	25.54	21.76	29.17	0.95	15.38	16.19	12.15	15.43
ship_001	1024	570	1.06	25.14	26.47	24.93	28.22	1.07	19.86	19.36	16.72	18.11
ship_003	1024	995	0.98	21.95	21.87	21.78	23.76	1.00	14.82	14.82	15.74	14.44
shipsec1	1024	963	1.01	20.88	20.75	22.22	25.16	0.98	14.60	14.87	13.43	15.02
shipsec5	2048	1246	0.98	20.60	20.76	22.85	25.02	0.99	15.14	15.18	14.23	15.28
shipsec8	1024	820	0.97	21.09	21.63	20.09	23.69	0.99	14.16	14.54	14.24	14.88
sme3Db	497	256	1.53	6.52	14.79	9.36	13.75	1.95	4.04	11.06	5.50	8.51
sme3Dc	512	387	3.08	3.89	15.75	6.25	15.56	3.05	3.10	11.62	5.19	9.51
smt	512	460	0.99	23.61	24.44	24.41	29.73	1.00	18.01	19.44	16.21	17.63
srb1	512	365	0.99	24.78	25.40	19.08	22.39	0.99	19.22	20.64	10.57	14.50

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
			MIN	1.35	1.17	6.25	13.75		0.64	0.60	5.19	8.51
			GEO-MEAN	18.13	19.43	21.44	24.73		13.02	14.01	13.48	15.08
			MAX	26.04	27.84	30.72	32.22		19.86	20.64	20.55	20.75
Problem Type: Structural Problem Sequence												
af_shell1	4096	2178	0.96	18.99	18.02	29.12	30.80	0.97	14.49	13.75	16.21	18.75
Problem Type: Subsequent Structural Problem												
af_shell2	4096	2178	0.96	19.00	18.01	27.49	30.93	0.97	14.36	13.73	16.00	18.89
af_shell3	4096	2178	0.96	18.97	17.98	28.83	30.49	0.97	14.57	13.69	16.28	18.91
af_shell4	4096	2178	0.96	19.13	18.11	28.96	30.96	0.97	14.48	13.83	16.16	19.14
af_shell5	4096	2178	0.96	19.06	18.05	28.99	31.11	0.97	14.61	13.76	16.12	18.77
af_shell6	4096	2178	0.96	19.01	18.06	29.33	30.34	0.97	14.48	13.79	16.05	18.84
af_shell7	4096	2178	0.97	19.05	18.13	29.23	31.04	0.97	14.48	13.69	15.96	18.08
af_shell8	4096	2178	0.96	19.04	18.03	28.91	28.10	0.97	14.50	13.74	15.96	18.96
af_shell9	4096	2178	0.96	19.13	18.00	28.68	30.46	0.97	14.42	13.71	15.99	18.85
			MIN	18.97	17.98	27.49	28.10		14.36	13.69	15.96	18.08
			GEO-MEAN	19.04	18.04	28.83	30.46		14.49	13.74	16.08	18.80
			MAX	19.13	18.13	29.33	31.11		14.61	13.83	16.28	19.14
Problem Type: Electromagnetics Problem												
gsm_106857	4096	2693	2.93	5.44	9.29	9.82	22.47	3.02	3.89	6.13	6.17	12.21
offshore	1024	534	1.31	8.23	14.03	9.15	15.32	1.37	5.26	9.33	5.25	8.70
			MIN	5.44	9.29	9.15	15.32		3.89	6.13	5.25	8.70
			GEO-MEAN	6.69	11.42	9.48	18.56		4.52	7.57	5.69	10.31
			MAX	8.23	14.03	9.82	22.47		5.26	9.33	6.17	12.21
Problem Type: Computational Fluid Dynamics Problem												
3dtube	512	396	0.95	23.62	21.26	20.36	21.21	0.93	16.50	14.49	12.58	13.07
atmosmodd	2048	1154	0.97	5.65	4.82	24.67	19.96	0.89	4.53	4.10	15.11	11.90

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
atmosmodj	2048	1154	0.97	5.64	4.78	24.62	20.39	0.88	4.53	4.07	15.10	11.95
atmosmodl	2048	1351	0.91	5.19	4.78	25.18	19.60	0.90	4.25	4.10	15.34	11.84
atmosmodm	2048	1351	0.92	5.31	4.69	25.18	20.28	0.89	4.23	4.09	15.37	11.90
cfd2	512	385	0.98	15.92	14.64	20.50	24.37	0.98	12.31	11.51	11.46	12.41
laminar_duct3D	512	473	0.92	18.69	25.63	20.01	20.84	0.94	15.33	16.44	12.90	13.13
mixtank_new	256	246	1.09	18.88	22.48	9.54	16.86	1.16	10.88	14.21	8.85	8.93
parabolic_fem	1024	481	1.02	4.86	6.71	14.06	22.41	1.04	4.73	7.65	8.60	12.24
poisson3Db	512	296	3.67	1.63	13.18	3.02	12.93	3.80	1.27	9.35	2.38	7.20
PR02R	1024	1009	1.00	19.57	19.59	23.32	25.84	0.97	14.39	13.89	10.98	14.95
ramage02	512	351	0.95	24.47	23.58	26.11	30.61	0.93	21.80	20.41	18.36	18.72
RM07R	8192	4597	1.03	23.86	23.57	30.22	31.22	1.02	16.83	16.69	20.26	21.64
rma10	512	293	1.00	19.45	21.55	17.85	22.25	0.99	16.40	15.00	10.70	11.08
StocF-1465	4096	2653	0.98	8.16	7.98	25.15	22.81	1.02	6.26	6.12	13.01	12.17
MIN				1.63	4.69	3.02	12.93		1.27	4.07	2.38	7.20
GEO-MEAN				10.35	12.00	18.57	21.64		8.08	9.30	11.68	12.47
MAX				24.47	25.63	30.22	31.22		21.80	20.41	20.26	21.64
Problem Type: Linear Programming Problem												
cont1_l	2048	976	0.93	3.58	3.53	21.40	20.37	1.00	3.08	3.09	13.67	12.68
cont11_l	1024	747	0.95	3.47	3.34	18.79	19.22	1.00	2.88	2.91	12.39	11.25
degme	961	1004	1.21	2.88	2.92	5.83	5.90	1.16	1.69	1.71	3.32	3.38
neos3	512	283	0.91	4.41	4.58	22.86	20.96	0.87	4.89	4.29	11.24	9.92
rail2586	807	978	1.11	4.02	2.85	8.05	9.30	1.06	2.51	1.62	5.48	6.56
rail4284	1539	1378	1.08	1.82	1.61	6.67	6.57	1.02	1.21	1.02	4.14	4.35
stat96v2	512	350	0.94	23.18	24.68	20.85	24.92	0.98	18.58	17.77	12.87	13.19
stat96v3	512	408	0.94	21.04	24.74	20.18	26.46	0.98	16.38	17.78	12.75	13.78
stormG2_1000	512	455	1.00	7.48	7.47	13.71	17.97	1.01	6.77	6.33	8.04	9.01

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
MIN				1.82	1.61	5.83	5.90		1.21	1.02	3.32	3.38
GEO-MEAN				5.44	5.30	13.66	14.85		4.25	3.92	8.31	8.47
MAX				23.18	24.74	22.86	26.46		18.58	17.78	13.67	13.78
Problem Type: Least Squares Problem												
ESOC	1024	755	1.01	12.22	12.37	22.49	26.39	0.99	9.20	9.79	12.60	15.86
Rucci1	2048	1072	1.00	4.01	3.82	15.08	22.66	1.00	3.42	3.23	8.85	13.29
sls	2048	937	1.05	3.86	3.95	16.52	18.86	1.30	3.00	3.09	7.13	11.22
MIN				3.86	3.82	15.08	18.86		3.00	3.09	7.13	11.22
GEO-MEAN				5.74	5.72	17.76	22.43		4.55	4.61	9.27	13.32
MAX				12.22	12.37	22.49	26.39		9.20	9.79	12.60	15.86
Problem Type: Model Reduction Problem												
boneS01	1024	828	0.95	20.27	20.09	18.76	23.54	0.95	13.93	13.76	14.05	14.07
filter3D	512	337	1.01	13.83	16.69	12.40	15.61	1.03	9.54	11.77	7.23	8.71
MIN				13.83	16.69	12.40	15.61		9.54	11.77	7.23	8.71
GEO-MEAN				16.74	18.31	15.25	19.17		11.53	12.72	10.08	11.07
MAX				20.27	20.09	18.76	23.54		13.93	13.76	14.05	14.07
Problem Type: Combinatorial Problem												
ch7-9-b5	512	336	0.68	6.47	3.99	18.16	4.42	0.57	5.54	3.01	10.57	2.81
ch8-8-b5	512	448	0.67	6.69	4.30	17.55	6.57	0.57	6.15	3.52	11.37	3.99
GL7d15	1024	771	1.16	3.41	3.23	3.52	4.34	1.56	1.90	2.03	1.98	2.59
GL7d17	4096	3266	1.26	1.96	1.87	2.10	2.35	1.31	1.58	1.55	1.75	2.00
GL7d18	8192	4464	1.27	1.81	1.65	1.94	2.14	1.35	1.51	1.48	1.70	1.94
GL7d19	8192	4673	1.36	1.75	1.64	1.89	2.16	1.39	1.47	1.41	1.66	1.87
GL7d20	4096	3737	1.40	1.76	1.67	1.89	2.21	1.46	1.48	1.42	1.64	1.91
GL7d21	4096	2269	1.41	1.84	1.70	1.91	2.27	1.46	1.52	1.44	1.65	1.90
GL7d22	2048	1029	1.38	1.91	1.76	1.88	2.25	1.50	1.49	1.48	1.55	1.88

Continued on next page

Table 5.5 – *Continued from previous page*

Matrix	spc	hppc	Single Precision					Double Precision				
			su	MKL		DWS		su	MKL		DWS	
				un	ord	un	ord		un	ord	un	ord
n4c6-b9	256	240	0.65	8.93	4.86	16.50	7.09	0.54	6.64	3.38	11.19	3.83
wheel_601	512	321	1.15	2.42	2.27	3.59	7.99	1.06	2.08	2.49	3.08	4.41
MIN				1.75	1.64	1.88	2.14		1.47	1.41	1.55	1.87
GEO-MEAN				2.90	2.40	3.93	3.47		2.32	1.97	2.99	2.50
MAX				8.93	4.86	18.16	7.99		6.64	3.52	11.37	4.41
OVERALL GEO-MEAN				9.45	10.43	14.65	19.3		6.48	7.14	9.32	11.24
OVERALL MAX				28	27.84	35.58	41.68		20.59	21.14	23.86	23.57

5.4 Future Work

5.4.1 Experiments with hybrid JDS-CSR format

There are 3 variants of hybrid JDS-CSR sub-matrix extraction as listed below.

1. **JDS-heavy:** Default version of the algorithm. In certain situations where estimated JDS and CSR vectorization costs are same, creates JDS structure.

2. **JDS-heavy CSR no-vec:** Everything is same with default version except CSR kenel is not vectorized. This scheme is designed to make use not only the vector pipeline but also the regular 2-issue pipeline contained in Xeon Phi Cores at the same time.

3. **CSR-heavy:** Both kernels are vectorized. In situations where estimated JDS and CSR vectorization costs are same, favors CSR structure. CSR heavy LBA is implemented by changing '$\leq$' in lines 9, 19 in Algorithm 4.6 into '$<$'. Below, in Figures 5.1 and 5.2, results of different senarios are given for both JDS and CSR heavy versions.

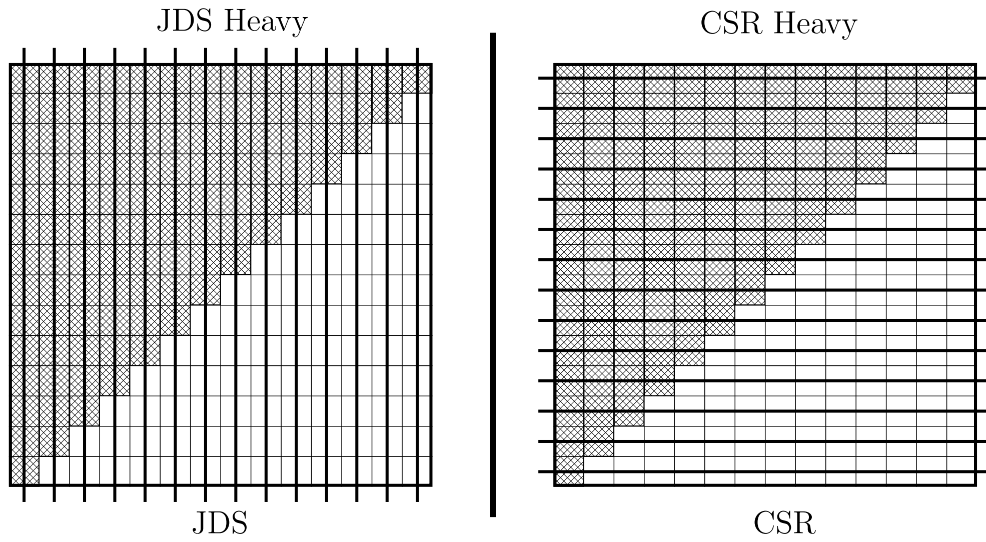

Figure 5.1: Comparison of JDS-heavy CSR and CSR-heavy JDS in certain senarios. Hybrid format doesn't necessarily contain both JDS and CSR everytime. In this case, cost of JDS and CSR are same. Therefore, one of them is chosen depending on the implementaion.

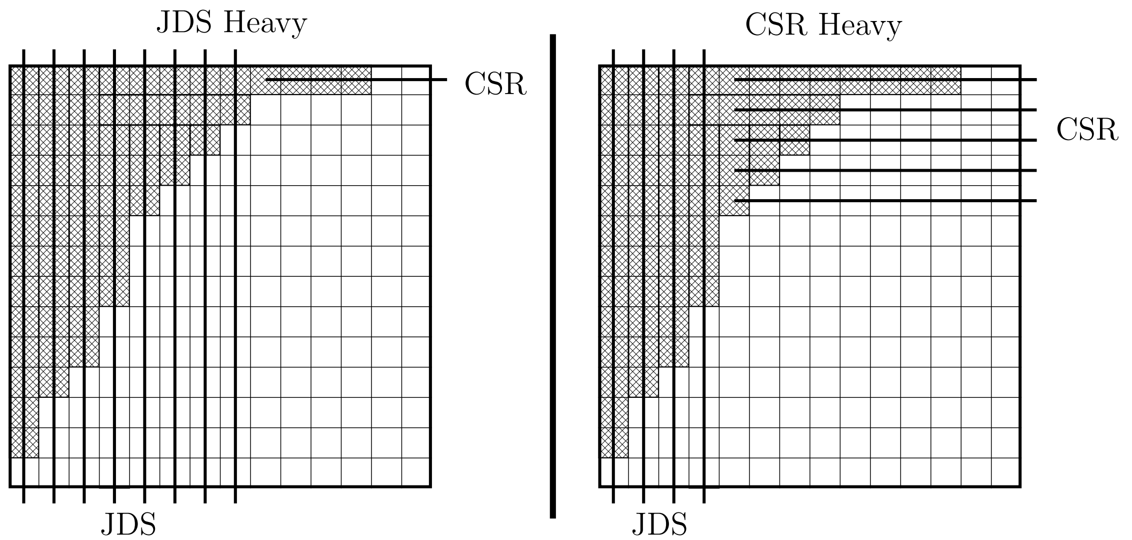

Figure 5.2: Comparison of JDS-heavy CSR and CSR-heavy JDS in certain senarios. Here, at certain points costs are the same. Depending on version of LBA, one of them will be chosen.

Out of 3 versions JDS heavy version performs the best by a very close margin (most of the time). So, only the results for that specific version is provided in Chapter 5. However, performance comparison of all 3 structures on test data set are given in Table 5.6.

Table 5.6: Different variants of LBA are compared. Results are taken from single precision Static routine. Performance and measured in GFlops. Partition size is 32KB.

Matrix	jds heavy csr no-vec	jds heavy	csr heavy
3D_51448_3D	8.42	13.12	13.4
3dtube	12.15	15.64	15.09
adaptive	4.15	4.03	7.52
atmosmodd	16.84	16.78	16.74
av41092	11.09	14.01	15.62
cage14	6.6	6.66	14.65
cnr-2000	11.2	12.4	12.51
F1	9.59	10.51	10.63
Freescale1	5.14	5.18	5.2
in-2004	13.36	13.53	13.68
memchip	7.71	3.56	7.75
road_central	0.72	0.67	1.2
torso1	14.8	21.4	21.18
webbase-1M	4.81	5.08	5.04
wheel_601	2.48	2.75	3.45
MIN	0.72	0.67	1.20
GEO-MEAN	6.89	7.27	8.95
MAX	16.84	21.40	21.18

Although, primarily used hybrid sub-matrix extraction version in this work is JDS heavy, as can be seen from Table 5.6, the version favoring CSR is more consistent and performs better. There are 2 possible reasons behind this;

1. Since CSR traverses single row upon completion of inner loop, only 1 write (after y entry is read) will commence. Instead of multiple times as in JDS. X vector entries are accessed randomly in both, but for matrices whose data

access patterns can be effectively regulated, JDS might perform better. The reason behind this is the possiblity of repeatedly using same X vector entry for different rows is higher than CSR.

2. The other reason is vector reduction instructions can be used in CSR based scheme. Not only to perform faster, but also to decrease cache usage by getting rid of extra writes on Y vector entries.

Judging by these results, CSR implementation of LBA described in Algorithm 4.7, should perform better than CSR heavy version of JDS implementation. It also doesn't incur conflicting writes when JDS and CSR packages in hybrid structure are tried to be executed simultaneously.

5.4.2 Choosing optimum partition size and decomposition algorithm

For matrices which are denser and quiet capable of filling SIMD unit, with rowwise 1-D partitioning algorithm, it is observed that L2 cache size of Xeon Phi can be restrictively small. As a result, some sub-matrices become mere single rows. To prevent this, other partitioning schemes such as 2D checkerboard are worth implementing.

5.4.3 GPU implementation

Scheduling & Load Balancing algorithms and LBA version described in Algorithm 4.7 described in this paper can be adopted by GPUs. In fact, job queues implemented for DWS routine warm-up stage is similar to OpenCL command queues and event system. Therefore, same environment can be simulated for GPU. Judging from single & double precision DWS routines performance difference, LBA does effectively use vectorization. Therefore, a GPU implementation is inevitable.

Chapter 6

Conclusion

In this thesis,

- locality aware & architecture aware task decomposition strategies,
- adaptive runtime scheduling and load balancing algorithms,
- a hybrid storage scheme for sub-matrices and
- a heuristic based algorithm to extract it

are developed to effciently execute SpMxV process for shared memory architectures.

Developed tools can be used by both spontaneous and regulated SpMxV operations. In this work, hypergraph partitioning models are used in an effort to regulate memory access patterns of X vector entries.

In Chapter 1, Generic concepts related to Parallel Programming and High Performance Computing are briefly introduced.

In Chapter 2, background information about SpMxV, task decomposition for SpMxV, hypergraph model, and Xeon Phi Microarchitecture are explored.

In Chapter 3, scheduling algorithms used in this work and what they are aiming for are explained in detail.

In Chapter 4, SpMxV routines are optimized to make better use of hardware resources, mostly SIMD unit and tested on a sample of SpMs having different structures. To uncover performance otherwise left hidden, a hybrid storage scheme and Laid Back Algorithm (LBA) is presented.

In Chapter 5, Distributed Work Stealing (DWS) routine is compared to Math Kernel Library's (MKL) cblas routines. Also additional tests to improve hybrid format produced by LBA are made.

In the future, As shown in Chapter 5, improvements for faster runtime and consistent behaviour are still possible. We plan to try those variations techniques on both Xeon Phi Coprocessors and implement a GPU version using OpenCL.

Bibliography

[1] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, "Introduction to Parallel Computing," edition.2, 1994.

[2] Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa, "Heterogeneous Compting with OpenCL," edition.1, 2012.

[3] Jim Jeffers and James Reinders, Intel Xeon Phi Coprocessor High Performance Programming, edition.1, 2013.

[4] Mark Sabahi, "A Guide to Auto-Vectorization with Intel C++ Compilers," 2012.

[5] Albert Jan Yzelman, "Generalised Vectorization for Sparse Matrix Vector Multiplication", 2014.

[6] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey, "Efficient Sparse Matrix Vector Multiplication on x86-Based Many-Core Processors," *27th International Conference on Supercomputing (ICS)*, 2013.

[7] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, Achim Basermann, and Alan R. Bishop, "Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation," 2012.

[8] Erik Saule, Kamer Kaya, and Ümit V. Çataly¨rek, "Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi," *Parallel Processing and Applied Mathematics*, 2013.

[9] Ronald W Green, "OpenMP Thread Affinity Control," 2014.

[10] F. V azquez, G. Ortega, J.J. Fern andez, E.M. Garz on, “Improving the performance of the sparse matrix vector product with GPUs,” *Computer and Information Technology (CIT)*, no.10, 2010.

[11] Umit V. Catalyurek and Cevdet Aykanat, “Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication,” *IEEE Transactions on Parallel and Distributed Systems*, vol.10, no.7, pp. 673 – 693, 1999.

[12] OpenMP Application Programming Interface, pp. 53 – 67, 2013.

[13] Rezaur Rahman, “Intel Xeon Phi Coprocessor Vector Micro-achitecture,”, 2013.

[14] Rezaur Rahman, “Intel Xeon Phi Micro-achitecture”, 2013.

[15] Yousef Saad, “Iterative methods for sparse linear systems,”, 2003.

[16] Andrew Witkin, Michael Kaas, “Spacetime Constraints,”, *Proceeding SIGGRAPH '88 Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pp. 159 – 168, 1988.

[17] Natarajan Viswanathan and Chris Chong-Nuen Chu, “FastPlace: Efficient Analytical Placement using Cell Shifting, Iterative Local Refinement and a Hybrid Net Model,” *Proceeding ISPD '04 Proceedings of the 2004 international symposium on Physical design*, pp. 26 – 33, 2004.

[18] Linux Kernel Archives, *https://www.kernel.org/*.

[19] Michael J. Flynn, “Some Computer Organizations and Their Effectiveness,” *Computers, IEEE Transactions* on vol.C-21, Issue.9, pp. 948 – 960, 1972.

[20] R. Duncan,“A survey of parallel computer architectures,” 1990.

[21] The University of Florida Sparse Matrix Collection, *http://www.cise.ufl.edu/research/sparse/matrices/*.

[22] GCC, the GNU Compiler Collection, *https://gcc.gnu.org/*.

[23] Intel C and C++ Compilers, *https://software.intel.com/en-us/c-compilers*.

[24] Intel Math Kernel Library Documentation, *https://software.intel.com/sites/products/documentation/hpc/mkl/mklman/.*

[25] Intel 64 and IA-32 Architectures Software Developers Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C.

[26] Intel Xeon Phi TM Coprocessor Instruction Set Architecture Reference Manual, 2012.

[27] Intel Xeon Phi Coprocessor - the Architecture, George Chrysos, 2012.

[28] Performance Obstacles for Threading: How do they affect OpenMP code?, Paul Lindberg, 2009.

[29] Valgrind official website, *http://valgrind.org/.*

[30] Nicholas Nethercote and Julian Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[31] Nicholas Nethercote, Robert Walsh and Jeremy Fitzhardinge, "Building Workload Characterization Tools with Valgrind," *Invited tutorial, IEEE International Symposium on Workload Characterization (IISWC)*, 2006.

[32] Nicholas Nethercote, "Dynamic Binary Analysis and Instrumentation,", 2004.

[33] Nicholas Nethercote and Julian Seward, "How to Shadow Every Byte of Memory Used by a Program," *Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, 2007.