Professur für Höchstleistungsrechnen
Department Informatik
Friedrich-Alexander-Universität
Erlangen-Nürnberg

MASTER THESIS

Implementation and Performance Engineering of the Kaczmarz Method for Parallel Systems

Christie Louis Alappat

Erlangen, 30th November, 2016

Examiner:
Prof. Dr. Gerhard Wellein

Advisors:
Moritz Kreutzer
Dr. habil. Georg Hager
Dr. rer.nat. Jonas Thies

Eidesstattliche Erklärung / Statutory Declaration

Erlangen, 30th November, 2016

Christie Louis Alappat

Abstract

The Kaczmarz method is a simple and robust iterative solver for linear systems of equations. It is used in different fields of science and engineering ranging from medical imaging to solving convection dominated flows, Helmholtz equations and eigenvalue problems. In this thesis we investigate hardware-efficiency and scalable shared memory parallelization strategies for the Kaczmarz method when used as a solver for sparse linear systems. The inherent data dependencies of this method hinder fine-grained parallelism like SIMD or multi-threading to be used efficiently. However, there exist techniques like multicoloring which can enable this level of parallelism. A critical analysis of the multicoloring approach both in terms of performance and qualitative behavior reveals its deficiencies on modern compute platforms. Starting with existing ideas, this thesis proposes a novel "block multicoloring" method, which leverages structural features of (partly) band- or hull-structured matrices. A thorough node-level performance analysis demonstrates that this approach outperforms traditional multicoloring significantly (up to $3\times$ on a single compute node) for a selection of relevant application matrices and never falls behind it even for malicious cases. Finally, our Kaczmarz implementation combined with block multicoloring is used as a linear solver in the FEAST method, to compute inner eigenvalues of large sparse matrices. These first results demonstrate the applicability of the presented approach and indicate its superiority for large scale computations as compared to direct solvers which are state-of-the art for FEAST method.

CONTENTS

1 Introduction **1**
1.1 Outline . . . 2
1.2 Acknowledgments . . . 3

2 Sparse Matrices **5**
2.1 Storage Format . . . 5
2.2 Permutations . . . 7

3 Shared Memory Parallelism **9**
3.1 Shared Memory Parallelism . . . 9
3.2 OpenMP Pitfalls and Best Practices . . . 10

4 Kaczmarz and related Methods **13**
4.1 Kaczmarz Method . . . 13
4.2 CARP Method . . . 15
4.3 CGMN and CARP-CG . . . 16

5 Multicoloring algorithms **17**
5.1 Graph (Vertex) Coloring . . . 17
5.2 SMP and Multicoloring . . . 19
5.3 Implementation . . . 22
5.4 Test Bed . . . 22
5.5 Results and Discussions . . . 23

6 Block Multicoloring **29**
6.1 Basic Idea . . . 29
6.2 Related Work . . . 31
6.3 General variants . . . 32
6.4 Bandwidth Reduction algorithms . . . 37
6.5 SMP and Block Multicoloring . . . 39
6.6 Implementation . . . 41
6.7 Results and Discussions . . . 46
6.8 Multicoloring vs Block multicoloring . . . 48
6.9 Validation . . . 52

7 Model Driven Optimisation **55**
7.1 Performance Model . . . 55
7.2 Block Vectors and Layer Condition . . . 61

7.3 Case study of complex shifts 65

8 Application Run - FEAST **67**

8.1 Eigenvalue Problem 67

8.2 FEAST algorithm 68

8.3 Test and Results 70

9 Conclusion and Future Work **73**

9.1 Conclusion 73

9.2 Future Work 74

10 Appendix **75**

10.1 Matrices 75

10.1.1 BENCH3D-x-A0 75

10.1.2 Graphene-x-x 75

10.1.3 PWTK 75

10.1.4 Spin-24-24-24 76

10.1.5 TAU 76

10.1.6 Topi-36-36-36 76

10.2 Complex Shift Algorithm 77

Bibliography **79**

1 INTRODUCTION

Linear systems of equations are an inseparable part of science and technology. Every linear systems can be expressed with a matrix $A \in \mathbb{C}^{n \times n}$, and two vectors b and $x \in \mathbb{C}^n$ as follows:

$$Ax = b$$

The unknown vector x has to be determined and is typically computed numerically using a linear solver. Development, optimization and parallelization of such solvers for modern parallel computer architectures are active fields of research in computer science and numerical mathematics. Together with the steady increase of available computational power this will allow to continuously increase n and to provide new insights in the application field. The problem of solving linear systems of equations is split up into two classes which differ in terms of application areas, numerical methods and hardware efficiency: Dense problems where A is a fully populated matrix and sparse systems, where most of the entries of A are zeros. In this thesis we focus on sparse systems, which can arise from many applications like computational fluid dynamics (CFD), finite element methods (FEM), quantum physics and chemistry. To solve these sparse linear systems there are two classes of solvers based on direct or iterative methods.

This thesis addresses one such solver called Kaczmarz method, proposed by the Polish mathematician Stefan Kaczmarz in the year 1937 [16, 40], and published in a German paper with the title "Angenäherte Auflösung von Systemen linearer Gleichungen". This method falls under the category of iterative methods, and most specifically it is a projection based method. The main motivation for analyzing such a solver is the current trend of scientific community to move from direct methods to robust indirect methods based on projections [67]. However, in order to exploit modern compute resources to the full extent, we need to efficiently handle the data and make use of the maximum possible parallelism. The naive Kaczmarz kernel does not allow for this due to data dependencies. A major part of the thesis is dedicated to the discussion on avoiding such dependencies, performance optimization and analysis of the method for sparse systems.

The Kaczmarz method is rarely used as a stand-alone solver due to its slow convergence for many systems that appear in the scientific community. However, recent developments have shown that derivatives of this method perform well and are extremely robust. It has been shown in [27, 28, 34] that one such method called CGMN (or its parallel variant CARP-CG) produces good convergence results for convection-dominated partial differential equations (PDEs) and high frequency Helmholtz equations, which are quite difficult to solve due to small diagonal elements.

The goal of this thesis is to provide a shared-memory parallel and hardware efficient Kaczmarz solver for the GHOST (General, Hybrid, and Optimized Sparse Toolkit) library [44] which is being developed in the ESSEX (Equipping Sparse Solvers for Exascale) [5] project under the German Research Foundation (DFG) priority program SPPEXA [13]. A potential application of the CGMN (CG accelerated Kaczmarz) algorithm in the ESSEX project is for solving the inner linear systems that appear in the FEAST [54] algorithm. FEAST is an eigenvalue solver used to compute some interior eigenvalues (λ_i) and eigenvectors (v_i) of a system as follows:

$$\begin{aligned} Av_i &= \lambda_i v_i \qquad (1.1)\\ \text{where, } A &\in \mathbb{R}^{n\times n}\\ \lambda_i &= \text{i-th eigenvalue}\\ v_i &= \text{eigenvector corresponding to } \lambda_i \end{aligned}$$

FEAST requires to solve multiple linear systems in each of its iterations. Since the arising linear systems are ill-conditioned and highly indefinite, current state of the art implementations of FEAST use direct solvers [42]. However, due to the nature of our application we commonly encounter large problem sizes which prohibits the use of direct methods. It has been shown that CGMN (or CARP_CG), due to its robustness is one good alternative for solving the systems encountered in FEAST [32]. Towards the end of the thesis a comparison between FEAST using CGMN and a standard implementation of FEAST is done to see the increasing need of such iterative methods as we move towards the exascale era.

Implementing a robust, scalable and hardware-efficient iterative solver for FEAST will allow to substantially extend the capability of eigenvalue solvers for the computation of interior eigenvalues for a large range of problem classes. Through the ESSEX project this development can be used to study recent discoveries in the field of quantum physics, e.g. properties of graphene or topological insulators [25, 37], as well as quantum chemistry.

1.1 Outline

The thesis is structured into 8 chapters. First 4 chapters are brief introductory chapters where various concepts relevant to the thesis like sparse matrices, shared memory parallelisation and Kaczmarz method are introduced. A reader who is familiar with these concepts could skip these chapters. The chapter 5 explains the difficulty involoved in parallelisation of Kaczmarz method and describes an initial approach to parallelise using the multicoloring method. In chapter 6 we then describe an alternative approach using block multicoloring which is the prime focus of this thesis. Implementation strategies and exhaustive performance analysis for both of these methods are conducted in the respective chapters. Later in chapter 7, we deal with performance modeling of the Kaczmarz kernel and shows some relevant case studies. Finally in chapter 8 we present a relevant application of the developed Kaczmarz kernel, where we study the qualitative behavior of the implemented method.

1.2 Acknowledgments

This thesis was conducted in the context of the joint DFG project ESSEX-II[1], with participation by Erlangen Regional Computing Center (RRZE), the department of computer science of the University of Erlangen-Nuremberg (FAU), German Aerospace Center (DLR) in Cologne, and the University of Wuppertal.

Foremost, I would like to express my heartfelt gratitude to my advisor Moritz Kreutzer for his constant and continuous support, patience, encouragement, sage advices and his meticulous attention to the detail. I am also grateful to Jonas Thies for fruitful discussions, ideas and his guidance throughout the thesis.

This thesis would not have been possible without the support and the wise knowledge from Prof. Gerhard Wellein and Georg Hager. Furthermore, I would like to thank Gerhard especially for his suggestions and critical evaluation for the betterment of the thesis and Georg for clearing my doubts and for his effort put into straightening out the thesis.

I am thankful to Thomas Röhl for his knowledgeable insights and clearing my pointless doubts. I would also like to thank all the staff and members of the RRZE HPC group who supported me during the thesis.

Finally, I would like to express my deepest gratitude to my parents and my sister for the love, encouragement and moral support.

[1] `http://blogs.fau.de/essex`

2 Sparse Matrices

Sparse matrices are a common part of scientific modeling and applications. In the ESSEX project large sparse matrices arise commonly from the discretization of Partial Differential Equations (PDE) in quantum physics and chemistry [62] . The way these matrices are stored and algorithms used to treat them plays a significant role.

In this chapter we introduce some of the basic concepts of sparse matrices, namely storage format and permutations.

2.1 Storage Format

A sparse matrix is a matrix with lots of zero elements. The ratio of zero elements to the total number of elements is called the sparsity of the matrix. In order to take advantage of the sparsity, one needs to store only the non-zero elements (nnz) of the matrix. The sequence or data format in which these elements are stored has a large impact on the performance of an algorithm implemented on it. Each data format has its own advantages and disadvantages and the optimal choice of a format is strongly related to the hardware in use. Although GHOST supports various storage formats through its novel unified data format approach [43], here in this thesis we mainly restrict ourselves to the widely used Compressed Row Storage (CRS) or Compressed Sparse Rows (CSR) format.

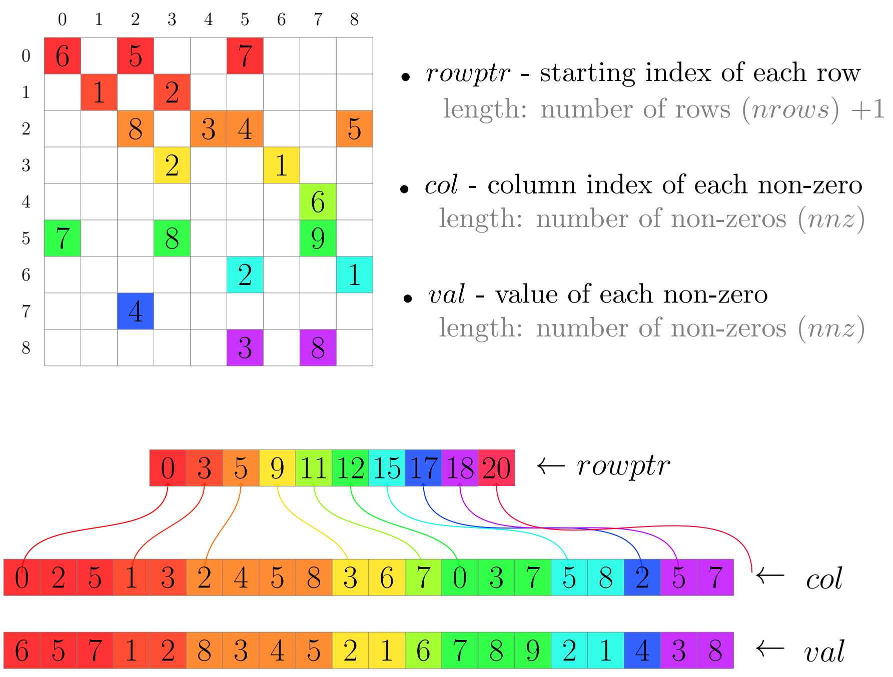
CRS

CRS is a standard sparse matrix storage format which has been in use from the mid-1960's [3]. This conventional data format consists of 3 arrays to store the entire sparse matrix:

- val - Stores non-zero values consecutively
- col -The column indices of the non-zeros

- $rowptr$ - starting index of each row in the compressed form. It is calculated as follows:

$$
\begin{aligned}
rowptr[0] =& 0 \\
rowptr[i] =& rowptr[i-1] + \text{number of non-zeros in } (i-1)^{th}\text{row}, \text{ if } i > 0
\end{aligned}
$$



rowptr - starting index of each row
length: number of rows (nrows) +1
col - column index of each non-zero
length: number of non-zeros (nnz)
val - value of each non-zero
length: number of non-zeros (nnz)
0 3 5 9 11 12 15 17 18 20 ← rowptr
0 2 5 1 3 2 4 5 8 3 6 7 0 3 7 5 8 2 5 7 ← col
6 5 7 1 2 8 3 4 5 2 1 6 7 8 9 2 1 4 3 8 ← val

Figure 2.1: Construction of the CRS storage format

Fig. 2.1 illustrates the basic concept of CRS. Here we see the matrix is compressed such that only non-zeros are stored. The name 'Row' in 'Compressed Row Storage' is due to the fact that non-zeros are stored consecutively in Row-major order (see the val array). Column indices of each non-zeros are stored in the col array and the indices to the beginning of each row are stored in the $rowptr$ array. By this way the position of any non-zero in the matrix could be determined by the col index and $rowptr$. Algorithm 1 shows the basic sparse matrix vector multiplication (SPMV) kernel using the CRS format.

Algorithm 1 SPMV (b=A x) in CRS format

```
for $row = 0 : nrows$ do
  temp = 0
  for $idx = rowptr[row] : rowptr[row+1]$ do
    $temp += val[idx] * x[col[idx]]$
  end for
  $b[row] = temp$
end for
```

CRS is in general a good and convenient format but in some cases it lacks the ability to exploit the vectorization capabilities of modern processors efficiently. We see from Fig. 2.1 that the only possibility for vectorization in this case is along each row, since the elements of a row are stored consecutively. However, the expected speed up of vectorization along rows might not be great due to 2 reasons:

1. The length of each row is usually very small and not constant, thus the overheads required for vectorization (like remainder loop, loop peeling etc.) are not negligible.
2. Most of the algorithms (such as SPMV, see for example Algorithm 1) have a reduction along the row length which causes further overhead.

Its also important to note that this potential problem of CRS depends on the algorithm and the machine. For example for the SPMV seen in Alg. 1, this effect might not be prominent on modern multi-core processors [43].

2.2 Permutations

Permutation (also known as reordering) is one of the most common techniques used in parallel implementations of direct and indirect solvers to resolve loop-carried dependencies, minimize communication and improve performance. Both columns and rows of a matrix can be reordered in arbitrary fashion, without disturbing the linear system. A formal definition for permutations is as follows [67].

Definition 1. *Let A be a $\mathbb{R}^{n\times n}$ matrix and π_r = {$i_1, i_2, ..., i_n$}, π_c = {$j_1, j_2, ..., j_n$} permutations of the set $1, 2, ..., n$. Then the matrices*

$$A_{\pi_r,j} = \{a_{\pi_r(i),j}\}_{i=1,...,n;j=1,...,n} \tag{2.1}$$

$$A_{i,\pi_c} = \{a_{i,\pi_c(j)}\}_{i=1,...,n;j=1,...,n} \tag{2.2}$$

$$\textit{where, } a_{i,j} - \textit{elements of A}$$

are called the row and column permutation of A respectively.

Mathematically the permutation operator can be seen as, a multiplication of the original matrix A with permutation matrices P_{π_r} and Q_{π_c}, which corresponds to row and column permutation matrix respectively. The permutation matrices P_{π_r} and Q_{π_c} are formed from the identity matrix by permuting its row and column indices as shown in eqn. 2.1 and 2.2 respectively. Then it holds:

$$A_{\pi_r(i),\pi_c(j)} = P_{\pi_r} A Q_{\pi_c} \tag{2.3}$$

If both rows and columns are permuted in similar way (i.e. $\pi_r = \pi_c$), the permutation is called a symmetric permutation. In this case $Q_{\pi_c} = P^{\mathsf{T}}{}_{\pi_r} = P^{-1}{}_{\pi_r}$ (since permutation matrices are orthogonal).

In practice the permutation matrices are not stored explicitly, but only the vector π_r and π_c. The reordering is directly applied to original matrix A as shown in Definition 1. The validity of permutations could be easily understood by relating the matrix to its corresponding linear system.

Let $\pi_r = \{1, 3, 2\}$ *and* $\pi_c = \{3, 1, 2\}$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$A_{\pi_r,\pi_c} = \begin{bmatrix} a_{13} & a_{11} & a_{12} \\ a_{33} & a_{31} & a_{32} \\ a_{23} & a_{21} & a_{22} \end{bmatrix}, x_{\pi_c} = \begin{bmatrix} x_3 \\ x_1 \\ x_2 \end{bmatrix}, b_{\pi_r} = \begin{bmatrix} b_1 \\ b_3 \\ b_2 \end{bmatrix}$$

From the above example, row permutations can be seen as reordering the order of equations and column permutations as reordering (renumbering) of unknowns. Both of these do not change the linear system as such, i.e., the systems $Ax = b$ and $A_{\pi_r,\pi_c}\, x_{\pi_c} = b_{\pi_r}$ are equivalent, if the column space vector [1] x is permuted according to the column permutation and the row space vector b is permuted according to the row permutation.

A major part of this thesis is concerned with finding an efficient permutation for an iterative solver which has dependencies.

[1] row and column space vector denotes the position of vector w.r.t matrix A, i.e, row denotes vector in domain-space of A and column denotes vector in range-space of A

3 Shared Memory Parallelism

Exploiting parallelism is the key to achieve high performance. Modern trends in hardware development to cope with Moore's Law [48] reveal ever increasing necessity to extract parallelism on the node-level. It is important to extract maximum performance on this level before going to inter-nodal parallelism, since the performance at this level gets amplified at large node counts. This trend will continue at least to the near future, as it can be seen from the emergence of many-core architectures, which push the need for concurrency even harder.

3.1 Shared Memory Parallelism

Shared Memory Parallelism (SMP[1]) is the parallelism extracted from parallel processors (multi-core) that share a global address space. In computing this form of parallelism enjoys many advantages over the other forms of parallel programming model like message passing. Major benefit comes due to the fact that all the processors have a consistent view of memory (shared memory) thereby avoiding storage of redundant data and eliminating the need to communicate data from one processor to the other. But programming a SMP system might be challenging sometimes and it requires a lot of practice to write a correct and efficient SMP program. SMP programming is usually based on 2 fundamental concepts: threads and the fork-join model.

A thread is the smallest unit of processing that can be scheduled by an operating system [22]. Threads can be executed concurrently within a multicore processor. This allows the program to utilize multiple processing units on modern processors.

Fig. 3.1 shows one of the most commonly used models in SMP programming called the fork-join model. According to this model, a program can be comprised of serial regions where the program is executed sequentially and parallel regions where a number of threads work simultaneously on a given task. The ultimate objective of a programmer is to enhance the percentage of the parallel region within the code, since according to Amdhal's law (Eqn. 3.1) the maximum parallel speedup is directly related to the parallel

[1]Note that in this thesis SMP refers to Shared Memory Parallelism and not Symmetric multiprocessing

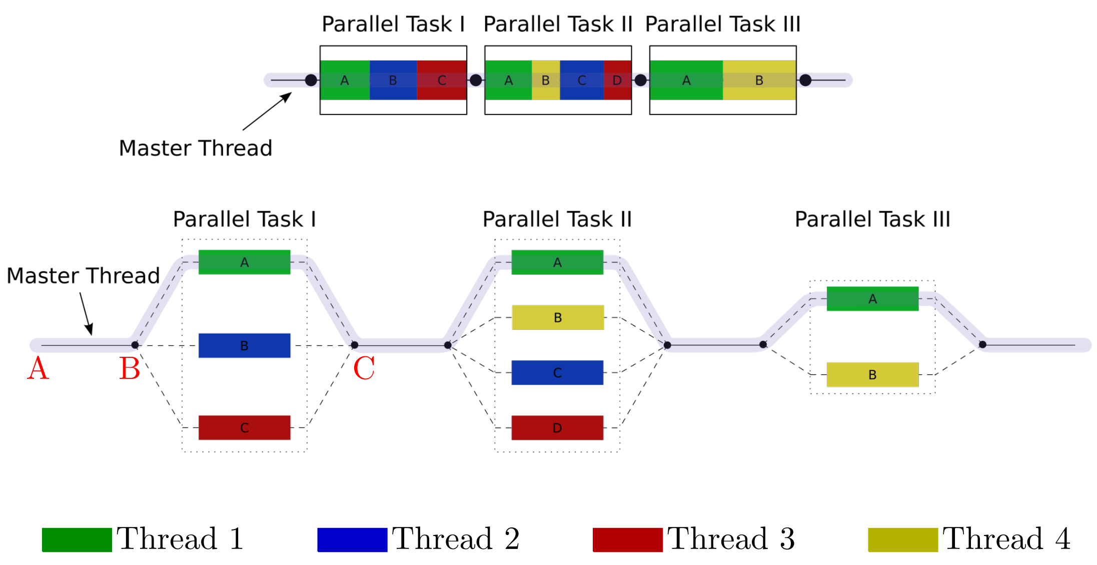


Figure 3.1: Fork-join model. (Figure taken from [17])

fraction of the code.

$$S_p(n) = \frac{1}{(1-p) + \frac{p}{n}} \tag{3.1}$$

where, $S_p(n)$- the speedup at, n parallel units

n- min (number of threads,number of parallel resources)

p- percentage of parallel region

In Fig. 3.1 initially (towards left) we see only one master thread working on the sequential part of the program. Then at point B a parallel region is constructed using the *fork* construct, where multiple threads work together. Further towards right at point C, threads are synchronized using the *join* construct. This cycle is then continued along the course of the program.

In this thesis we use the OpenMP (Open Multi-Processing) application programming interface (API), which works on the above concept, to employ shared-memory parallelism. OpenMP uses simple compiler directives to parallelize parts of the program. In this way the programmer is relieved from the low-level details like forking, joining, thread management and to some extent load balancing. The API also consists of several user functions to enable fine-tuned parallelism like locks and to support purposes like setting and querying the number of threads and to identify threads.

3.2 OpenMP Pitfalls and Best Practices

With multiple threads having the same view of the memory, there are several things that a programmer has to keep in mind while using the OpenMP programming paradigm. Debugging such a program might be notoriously difficult, and if not taken care the performance might not be as expected. Here we present some of the common aspects one has to bear in mind while using OpenMP. These aspects would serve as the basic platform for chapters 5 and 6. Most of the ideas in this chapter have been derived from the book [36].

Race Conditions

A race condition is a problem of correctness, it happens when a thread tries to access (write) data from (to) a memory location which another thread has modified. Since the threads are executed concurrently the result of such an operation is not deterministic and varies with runs. Due to this non-deterministic nature it might be sometimes difficult to pin-point this kind of bug.

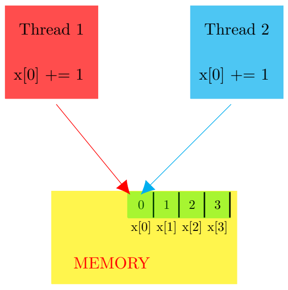


Figure 3.2: Race condition

Fig. 3.2 shows a potential case of a race condition, here each thread tries to update the same memory address x[0] with value 0, using the operation `x[0] = x[0] + 1`. In a parallel environment as shown in the figure the result for this operation is undefined. It can be either 1 or 2 depending on the sequence in which the data from x[0] is fetched. Some of the alternatives to solve such problems are: privatization (clauses like `private`, `firstprivate`) and serialization (clauses like `critical`, `atomic`) . But the programmer has to take care to use them wisely and avoid potential serialization of the code. Sometimes it might be even useful to change the algorithm slightly to support parallelization.

False Sharing

False sharing is a problem which affects the performance of the code rather than its correctness. The situation is similar to race conditions, here the thread tries to access (or write) a variable in a cache line which has been modified by another thread. Note that threads do not access (or write to) the same variable but different variables belonging to the same cache line. This operation causes a lot of extra (background) data transfers due to cache coherence protocols, causing a great deal of performance and scalability.

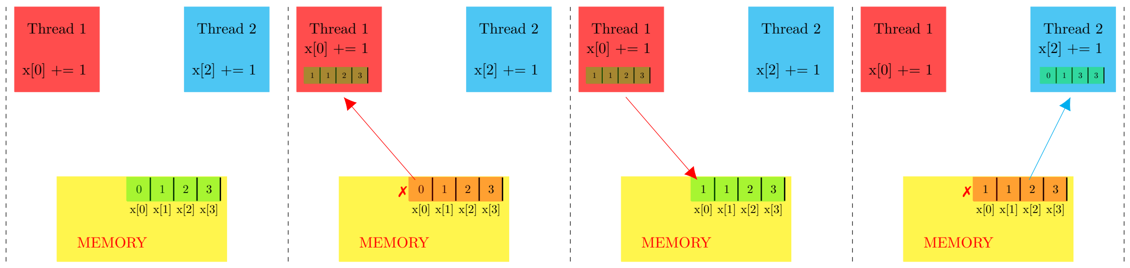


Figure 3.3: False sharing

Fig. 3.3 shows a typical false sharing scenario, assuming the cache line size to be 4, and that x[0] is aligned to the beginning of a cache line. Now thread 1 wants to modify x[0] and the other thread x[2]. Since data in the memory hierarchy is transferred only in blocks (cache lines) of fixed size (cache line size), the entire cache line from memory is transferred to the processor whose thread requests this cache line first (thread 1 in figure). Next thread 1 modifies x[0], causing cache coherence protocol to invalidate the copy of cache line in memory which further causes thread 2 to wait (stall) until the cache line is written back by thread 1 to memory (or a cache hierarchy shared by both threads). The thread 2 can start its work once the updated copy of cache line is present. From this simple example one could imagine the great deal of extra data traffic and stalls that occur in this process.

Two major indicators of false sharing are the lack of scaling in performance with increasing number of threads and an increase in data transfers within the shared memory hierarchy due to the ping-pong effect, which typically increases with the number of threads. Proper scheduling and clever data structures are the key to avoid such effects.

Barrier overhead

Commonly at some point of the code there arises the necessity to synchronize between threads. In OpenMP synchronizations are commonly achieved either by barriers or locking. The most efficient method to synchronize all threads at once is the barrier method. But one has to use them wisely, since each barrier costs cycles and if the ratio of barrier time to computation time is high it might lead to serious performance drops. Also load balancing plays an important role at the point of each synchronization, since the time taken by threads to reach the barrier is governed by the slowest thread.

Sometimes this type of performance degradation might go unnoticed due to implicit barriers in OpenMP. To make the situation worse the time taken by a barrier depends on the compiler used and performance might vary from compiler to compiler if the code has a significant number of barriers. To avoid such situations it might be a good idea to analyze the number of barriers (both implicit and explicit) in the code and remove unneeded barriers and ensure the load of each thread is properly balanced. In some cases methods like locking and relaxed synchronization could also be tried as an alternative.

Data placement

In modern heterogeneous clusters each shared memory node comprises of various individual machines (or sockets) connected to each other. The global address space of the entire node is the same, maintaining a consistent view to memory for each processing unit, irrespective of the socket in which the data is placed. This type of architecture is called the Non-Unified memory access (NUMA).

Due to the transparent view of memory SMP can be employed quite easily in these architectures. But in order to achieve scalability one has to keep an eye on the physical distribution (placement) of the data, since remote data accesses (i.e. accessing data from a neighboring socket) can be quite expensive causing a lack of scalability across the sockets.

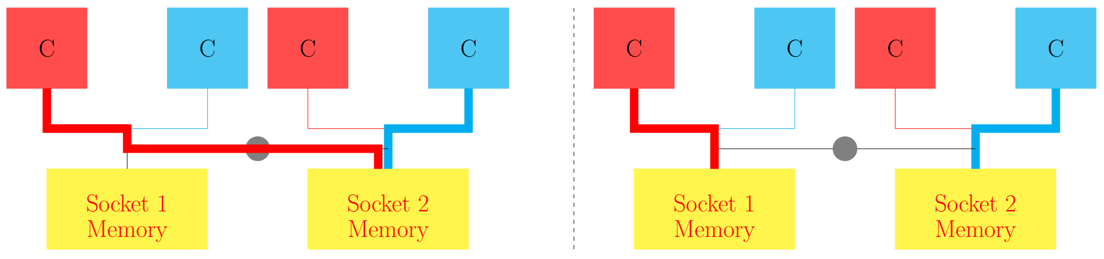


Figure 3.4: Data placement - Left: Incorrect placement of all the data in Socket 2, Right: Correct placement (figure taken from [65])

Fig. 3.4 shows incorrect and correct data placement. On the left we see that the processor in socket 1 has to fetch data from remote memory ,i.e., socket 2 (here) causing the data to be transferred through a potentially slower link that connects the two sockets (called QPI [11], or HyperTransport [7]).

The placement of data in ccNUMA architectures works on the "First Touch Policy" which states that a memory page is mapped into the locality domain of the processor that first writes to it [65]. This implies the data has to be placed properly right from the initialization phase. In Fig. 3.4 data used by socket 1 should be initialized by a thread pinned to socket 1 and correspondingly with socket 2.

4 KACZMARZ AND RELATED METHODS

4.1 Kaczmarz Method

The Kaczmarz method (KACZ) is a linear iterative solver developed by Stefan Kaczmarz in the year 1937. It is based on a row projection technique in which iterates are projected onto successive rows (or hyperplanes) represented by the system.

Consider a system of linear equation

$$\begin{aligned} Ax &= b \qquad (4.1) \\ A &= \mathbb{R}^{m\times m} \text{ matrix} \\ b &= \mathbb{R}^{m} \text{ RHS vector} \\ x &= \mathbb{R}^{m} \text{ unknown vector} \end{aligned}$$

The basic Kaczmarz iteration scheme to solve the above system is shown in Eqn. 4.2

$$x^{k+1} = x^k + \omega * \frac{(b_i - < A_i, x^k >)}{\|A_i\|^2} * A_i^T \qquad (4.2)$$

$$\begin{aligned} \text{where: } k &= 0,1,2,\ldots \text{ is the iterate number,} \\ i &= (\text{k mod m}) + 1 \text{ is the row index} \\ \omega &= \text{relaxation parameter} \\ A_i &= \text{i-th row of } A \\ A_i^T &= \text{complex conjugate of } A_i \end{aligned}$$

The scheme takes the current iterate x_k and projects it orthogonally along the normal direction $\frac{A_i^T}{\|A_i\|}$ of the i-th row of the matrix. The scaling factor $\frac{(b_i - <A_i, x^k>)}{\|A_i\|}$ is the normal distance of the point x_k from the hyperplane represented by the i-th equation of the system. Fig. 4.1 illustrates this basic concept. The

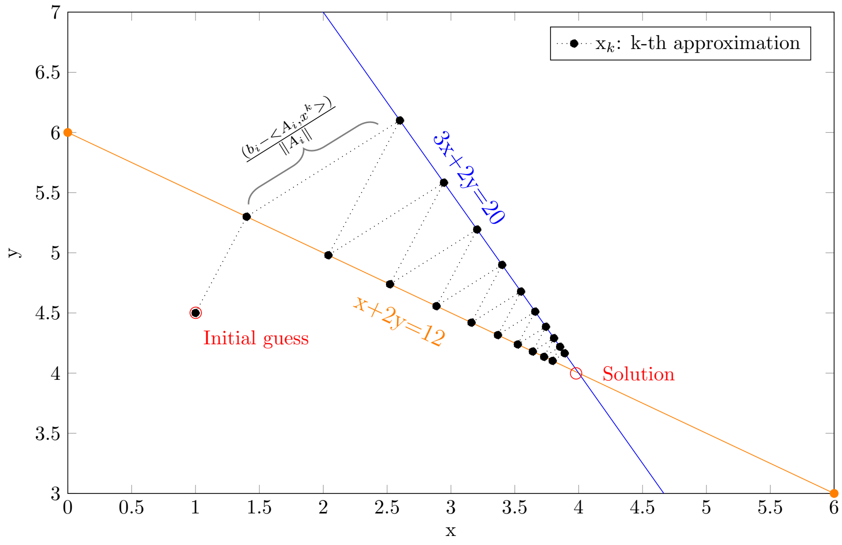


Figure 4.1: Kaczmarz Method Illustration for m=2 : orthogonal projection to each row from the k^{th} iterate to obtain the $k+1^{st}$ iterate is shown.

scheme being an orthogonal projection has the advantage that we need not store extra directions as compared to oblique projection schemes. A proof of the scheme's convergence can be found in [52].

It could be seen that the Kaczmarz scheme with fixed parameter ω is equivalent to a Successive Over Relaxation (SOR) scheme on the normal equation [52]:

$$AA^T y = b \tag{4.3}$$
$$x = A^T y \tag{4.4}$$

Being a solution of the normal equation, one could expect the condition number κ of such a method to be worse than that of matrix A, since

$$\kappa(AA^T) = \kappa(A)^2 \tag{4.5}$$

and hence if the condition number of A is poor (< 1), the squared condition number would be worse decreasing the convergence rate. But on the other hand the matrix AA^T is symmetric positive definite (SPD) even if A is not, which makes it a suitable candidate to be accelerated by simple methods like Conjugate Gradient(CG) as we will see in Section 4.3. Also it makes the diagonal elements relatively large, enhancing the robustness of the system.

In recent years a wide variety of modifications of the basic scheme have been developed, some of which are Randomized Kaczmarz [60] [49], Block Kaczmarz [31], Asynchronous Kaczmarz [45] and various other acceleration schemes. In this thesis we concentrate only on the Direct (basic) Kaczmarz method due to the kind of applications which we encounter (see Section: 4.3, and Chapter 8).

Algorithm 2 shows the basic KACZ algorithm in the CRS matrix format.

Algorithm 2 KACZ: solve for x: $Ax = b$

```
for $row = 0 : nrows$ do
    $scale = b[row]$
    $norm = 0$
    for $idx = rowptr[row] : rowptr[row + 1]$ do
        $scale- = val[idx] * x[col[idx]]$
        $norm+ = val[idx] * val[idx]$
    end for
    $scale* = omega/norm$
    for $idx = rowptr[row] : rowptr[row + 1]$ do
        $x[col[idx]]+ = scale * val[idx]$
    end for
end for
```

4.2 CARP Method

CARP or Component Average Row Projection [33] is a parallel variant of Kaczmarz method, envisaged by Dan Gordon and Rachel Gordon to overcome the parallelization difficulty of Direct Kaczmarz method, explained in Chapter 5. The method works on the concept of domain decomposition and falls under the category of CADD (Component Average Domain Decomposition) methods. In this method the equations are divided into blocks and on each block a local KACZ sweep is performed independently. The unknowns (clones) that appear in 2 or more blocks are then replaced by the average of its value in each of the sharing blocks.

The proof of convergence follows from transforming the equation 4.1 in $\mathbb{R}^m$ to a superspace $\mathbb{R}^s$ (s $\geq$ m), where the equations in each block are independent to each other. Thus independent (parallel) KACZ sweeps on each block are equivalent to a KACZ sweep on the entire superspace $\mathbb{R}^s$. Further it can be shown that the averaging operation is equivalent to row projections that make projection on $\mathbb{R}^s$ equivalent to $\mathbb{R}^m$, hence making "KACZ on $\mathbb{R}^s$ + averaging" equivalent to normal "KACZ on $\mathbb{R}^m$". A detailed proof could be found in [33].

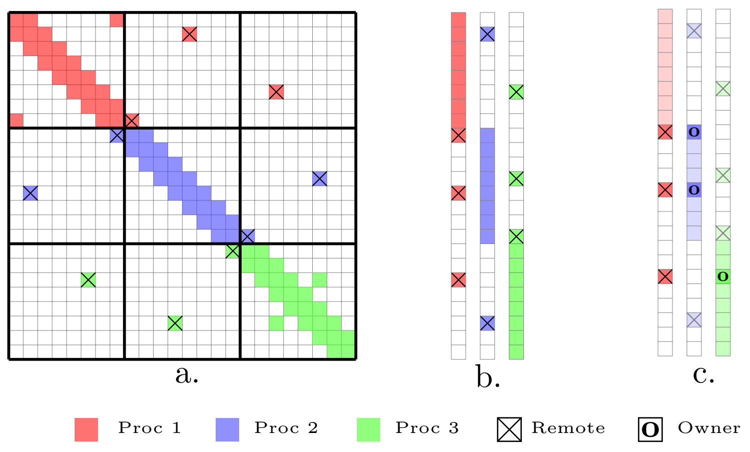


Figure 4.2: CARP: Illustration of the CARP method on a sparse matrix (shown in Fig. 4.2.a). The KACZ iteration can be run in parallel on different processes (shown with different colors). In the second step the clones from each process are sent to their respective owners where they are averaged, Fig. 4.2.c shows clones from process 1 being sent to its respective owners.

The CARP method is an ideal method for parallelization on distributed systems, since all blocks can be assigned to different processes which can do independent KACZ sweeps. Only in the averaging step they need to communicate and synchronize. But in a shared memory system this approach is questionable since one could do better due to the shared view of memory by all processes. Storing the clones (shared unknowns) of the elements might not be the optimal way to go. The key point of this thesis is to introduce alternative methods for shred memory systems (Chapter 5 and 6). Although we use CARP for distributed memory parallelization using MPI we will not discuss about it in this thesis.

4.3 CGMN and CARP-CG

CGMN is the Conjugate Gradient (CG) acceleration of Kaczmarz method and first appeared in the work of Björck and Elfving [23]. It makes use of the fact that KACZ is similar to SOR on the normal equation which is symmetric irrespective of A. Thus, if KACZ is run in two sweeps,i.e., a forward sweep (projection on row 1 to $nrows$) followed by a backward sweep (projection on row $nrows$ to 1) the resulting iteration matrix is symmetric (similar to SSOR). This enables the method to be accelerated using the CG method (see [23] and [34] for more details).

Later on with the emergence of the CARP method (parallel version of KACZ) by Gordon and Gordon they showed that the CGMN method could also be applied to accelerate CARP instead of KACZ [28]. The resulting algorithm was named as CARP-CG. Both of these methods have been proven to be very robust and efficient for many applications [34] [28] [23] [35] .

The algorithm as shown in Alg. 3 is basically similar to a normal CG algorithm but instead of carrying out matrix vector multiplication, 1 iteration of KACZ or CARP is performed. The $KERNEL$ in the algorithm is KACZ for CGMN or CARP in case of CARP-CG.

Algorithm 3 CGMN/CARP-CG : solve for x: $Ax = b$

```
Set $x^0 \in R^n$ to arbitrary value
Set $p^0 = r^0 = KERNEL(A, b, x^0, omega) - x^0$
Set TOL = desired tolerance
$k = 0$
**while** ($\|r^{k+1}\|$ / $\|r^k\|$) < TOL **do**
    $q^k = p^k - KERNEL(A, 0, p^k, omega)$
    $\alpha_k = \|r^k\|^2 / \langle p^k, q^k \rangle$
    $x^{k+1} = x^k + \alpha_k p^k$
    $r^{k+1} = r^k - \alpha_k q^k$
    $\beta_k = \|r^{k+1}\|^2 / \|r^k\|^2$
    $p^{k+1} = r^{k+1} + \beta_k p^k$
    $k = k + 1$
**end while**
```

5 MULTICOLORING ALGORITHMS

5.1 Graph (Vertex) Coloring

Multioloring or graph coloring techniques is one of the oldest techniques used to resolve data dependencies of the kind seen in chapter 3. One of the well-known and earliest form of this method is the Red-Black Gauss Seidel, in which the nodes (or unknowns) are colored into two colors red and black, allowing for complete parallel sweeps within a color. There have been recent advances in the field of graph theory, and a number of efficient graph coloring algorithms and implementations emerged (e.g., COLPACK [2]), which could work on general graphs (or sparse matrices). These techniques enable better use of modern computer architectures by exploiting their parallelism. Please note that throughout this thesis we use the terms "graph coloring" and "vertex coloring" intercahngeably, although it is not limited to vertices but also other components of the graph (e.g., edges).

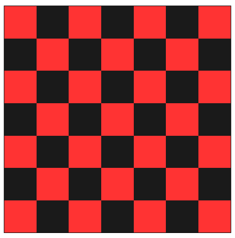


Figure 5.1: Red-Black coloring on a 2D-5pt Stencil

Distance k coloring

In contrast to the 2D Cartesian finite grid case, coloring a general sparse matrix might not be too obvious and in general a matrix can have more than two colors. However, starting from the insight into how 2D

Cartesian grid coloring works, one can generalize the concept to arbitrary sparse matrices. Figure 5.1 illustrates the common checkerboard pattern used for the Red-Black Gauss Seidel. Here we see that each of the neighbors (2D-5pt Stencil) of a point (say red) in the grid has a color different (say black) from itself, but note that two red grids (or black) can share the same neighbors. In graph theory this is known as the distance 1 coloring, since it only has the constrain that no two grid points (or vertices) sharing an edge in common should have the same color. But if we look back into the Kaczmarz algorithm (see Algorithm2), and restrain it to the 2D-5pt Stencil (see Fig 5.2) we see that this is not sufficient, since in the Kaczmarz method we have write to all the neighbors in addition to the current grid point (the center), causing write conflicts. This brings us to the concept of distance-k neighbor and coloring.

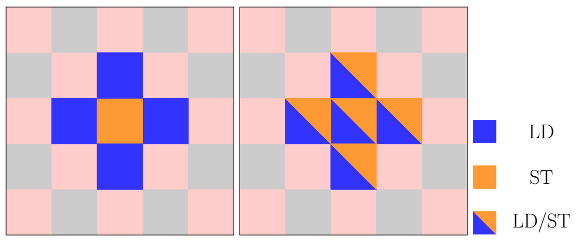


Figure 5.2: Loads and Stores in Gauss Seidel (Left) vs Kaczmarz (Right) using a 2D-5pt Stencil pattern

Two vertices or nodes are said to be **distance-k neighbors** if the shortest path connecting them consists of at most k edges[29]. A **distance-k coloring** is a technique of labeling (called colors) the vertices of a graph, such that no two distance-k (k>0) neighbors of the graph share the same color. For the KACZ algorithm we see if we color the graph with a distance k ≥ 2 , we could avoid the write conflicts seen in Fig. 5.2, since distance 2 coloring (abbreviated as D2-coloring) would ensure that while iterating (sweeping) through a color they even do not have common neighbors.

Analogous to the stencil case seen above the same concepts could be used to D2-color a sparse matrix (or graph), which is equivalent to saying none of the rows in a color of a sparse matrix share the same column entries. Mathematically this is referred to as structural orthogonality.

$$|x(i)||y(i)| = 0; \quad \forall i \in [1, ncols] \tag{5.1}$$

where, x and $y-$different rows of the matrix

$i-$column index

The Figure 5.3 adapted from [29] clearly illustrates a distance-2 coloring. We see that row 1 and row 2 do not have any common column entries, since they belong to same color (here depicted as green). The figure also shows the equivalence of the distance-2 coloring of graph of A (G$_a$) and the distance-1 coloring of the 2nd power of the graph (G_a^2) (i.e distance-2 graph);

$$\begin{bmatrix} a_{11} & 0 & 0 & 0 & a_{15} \\ 0 & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ a_{51} & 0 & a_{53} & a_{54} & a_{55} \end{bmatrix}$$

A

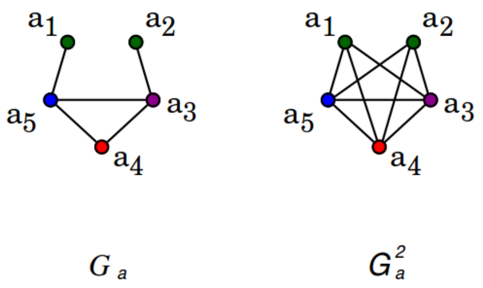


Figure 5.3: An illustration of D2-graph coloring (adapted from[29])

Satisfying this constrain will avoid any race conditions seen in the chapter 3 for KACZ algorithm due to its indirect access in the second loop, made clear in Alg. 4, since within a color one could ensure the col[idx] wouldn't yield the same value for different threads.

Algorithm 4 KACZ: race condition

```
for $row = 0 : nrows$ do
  ...
  $scale* = omega/norm$
  for $idx = rowptr[row] : rowptr[row + 1]$ do
    $x[col[idx]] += scale * val[idx]$
  end for
end for
```

In general one will aim at minimizing the number of colors in a graph; of course, assigning a distinct color to each row of the matrix would satisfy any k-coloring constraint. But this would lead to serialization of the algorithm destroying the entire purpose of multicoloring. The least number of colors that would be needed to color a matrix is known as the **chromatic number**. The efficiency of the KACZ algorithm or any other algorithm that depends on coloring is strongly dependent on this number as we will see in the next section.

The graph coloring problem is NP-Complete[9], i.e., even though any given solution can be verified quickly, there is no (proven) efficient way to find a solution. Most of these algorithms work on heuristics and one of the most commonly used one is the Greedy Algorithm. For our application we used the open source library called COLPACK [2] to D2-color our matrix. This library has a lot of variants of coloring based on the Greedy algorithm. A general overview of the algorithm and coloring techniques implemented can be found in [20].

5.2 SMP and Multicoloring

Once we have a D2 colored matrix, we have seen that rows within a color are structurally orthogonal, which makes it possible for different threads to work on different rows without any write conflicts. Fig. 5.4 illustrates how the coloring and permutations are done for a FDM[1] like matrix. The matrix has to be permuted into blocks of each color during the pre-processing stage as seen in Fig. 5.4.4, to avoid indirect accesses of the matrix rows during the operation on KACZ kernel. One thing interesting to note here is that since we parallelise within a color, the access pattern of this permuted matrix, i.e. the order in which entries are used, is different from the normal access pattern used in other algorithms that do not use this coloring (like SPMV). This data access patterns could create some performance issues which are discussed below.

Increase in α factor

To exploit the modern architectures one should make proper use of fast but small memories called caches. A large gain in performance could be achieved if one could reuse data from the lower cache hierarchies[2]. A look into KACZ algorithm (see Alg. 2) or any algorithm of a similar kind (SPMV), we can clearly see that we access the x (unknown) vector many times. So a good implementation must try to keep the expensive data brought from the memory in the smaller but faster caches as much as possible. The factor α quantifies this x vector traffic[43]. For a given sparse matrix A and a particular memory hierarchy M, it is defined as:

$$\alpha(A, M) = \frac{\textit{Total effective Data traffic from M caused by x-vector}}{\textit{sizeof(element type of x) * number of non-zeros of A(nnz)}} \tag{5.2}$$

possible cases:

[1]FDM - Finite Difference Matrix - Matrix arising from discretization using Finite Difference method
[2]Lower memory hierarchy refers to memory closer to the core (CPU)

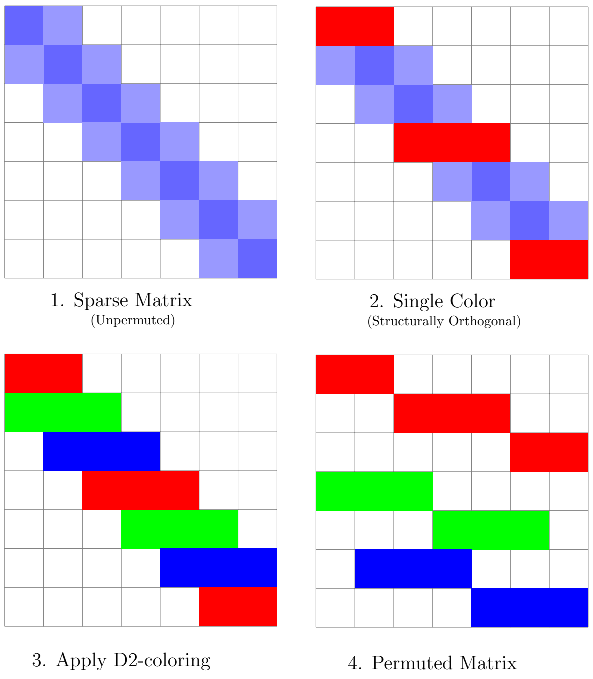


Figure 5.4: Basic idea of multicoloring: Illustration of permutation due to multicoloring for matrix in Fig .5.4.1

1. α(A,M) = 0; x vector fits into a memory hierarchy lower than M
2. α(A,M) = $\frac{nrows}{nnz}$; x vector loaded once from M
3. α(A,M) = 1; no proper cache reuse or no cache available
4. α(A,M) > 1; no proper use of cache lines and/or prefetched data. Data traffic in excess of theoretical maximum without caches

Since in most of the architectures caches use a LRU (Least Recently Used) replacement policy or its variants, the ideal situation (assuming all the data do not fit in M) would be to finish all the calculation on the x element brought into M, before it gets evicted to higher cache levels. This would then correspond to the case 2 shown above. The factor α depends strongly on the matrix. For example, if we consider the original unpermuted matrix in Fig.5.4.1, we can observe that due to its non-zeros being clustered at the diagonal, α would be close to the ideal case. But once the matrix is permuted with multicoloring (Fig.5.4.4) the non-zeros are more scattered and we see there is no reuse of the data within a color. Even worse, after each color the entire x vector is fully re-loaded, possibly evicting the data loaded in the previous color. This is illustrated in Fig.5.5.

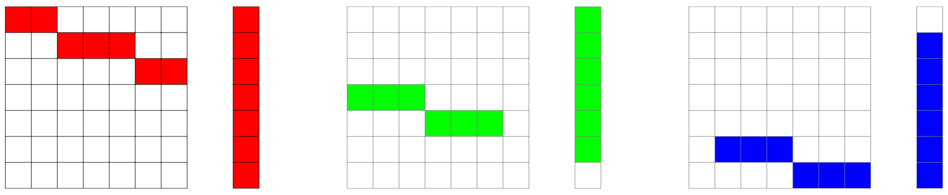


Figure 5.5: Illustration of x vector (unknown) access in a multicolored matrix. Compared to the unpermuted matrix we see here no reuse of loaded elements within a color, causing extra data traffic

This behavior is not limited to this special case or example shown above but also to a general sparse matrix permuted with multicoloring. This is not surprising, since most of our sparse matrices are derived from physical models and hence have strong neighboring relationships, but the ultimate aim of multicoloring is to group rows (here) that are totally unrelated (independent) and that do not have any common column entries (see Section 5.1), thereby disrupting any neighboring relations. This fact implies that there is no reuse of the elements loaded until the next color is reached and hence increase in α.

For a more general matrix(as opposed to stencil-like matrices) one could also expect an increase in α factor especially within inner hierarchies as multicoloring such matrices would lead to more scattered column indices, thereby causing irregular accesses with stride>1 and associated penalties. The situation becomes worse in multi-threaded environments since one cannot ensure the same thread will update the same vector data (probably cached) when it advances to the next color.

Increase in Global Barriers

It is common knowledge that in multi-threaded environments global barriers can be expensive (see Chapter 3). With multicoloring we need a global barrier after each color, since we need to ensure that no 2 threads work in different colors, causing race conditions. Thus the barrier cost is directly proportional to the chromatic number of the matrix. Even though an efficient MC algorithm tries to minimise the chromatic number, this might not be sufficient to make the barrier cost negligible for the matrix and algorithm at hand.

False Sharing

We have seen in chapter 3 that false sharing may be a major performance bottleneck in shared memory systems. In general for an algorithm with ordered writing pattern (like SPMV) false sharing takes place only at boundaries. However, in the KACZ algorithm we have scattered writes depending on the column indices of the current row, and there is no guarantee that no two threads write to the same cache line. A way to minimize this effect is to keep the matrix close to the diagonal pattern. But as we have seen, this is exactly opposite to what the MC algorithm does by scattering the data, thus increasing the probability for false sharing.

Compared to a stencil like structure as shown in Fig. 5.4 the tendency towards false sharing increases for a more general sparse matrix. In case of stencil-like matrix there occurs false sharing mainly at boundaries between threads in each color, but for a general matrix with more scattered column indices the situation could get worse.

One possible way to reduce the impact of false sharing is to color the matrix with a distance-k coloring such that k > cache-line size. This would avoid false sharing as a whole but at a possibly very high cost, since it could lead to a drastic increase in the α factor (α > 1 could be expected), and it would also increase the chromatic number.

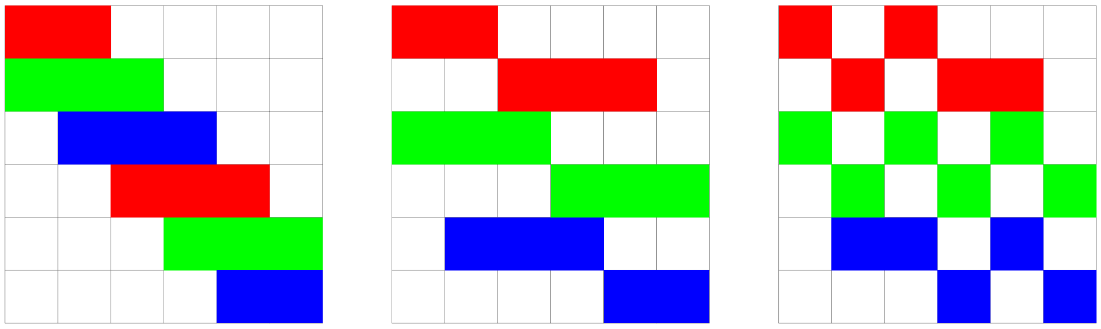
Improper data Placement

Multicoloring may cause non-optimal data placement in ccNUMA architectures (see Chapter 3). The data placement required by algorithms that depend on multicoloring is substantially different from other algorithms that do not have such data dependencies (like SPMV). For example in Fig. 5.4, if we have 2 threads then a normal SPMV algorithm with default static scheduling would distribute the first 4(or 3) rows to thread 1 and the next 3(or 4) rows to thread 2 for computation. However, the same distribution cannot be used for KACZ and here the rows within each color is distributed between the threads as shown in Fig 5.5.

This would potentially affect the scalability of the code across the sockets. One possibility to alleviate this problem is to initially place the data as required by MC dependent kernels (KACZ) and then modify other kernels (like SPMV), to use the same access pattern. Although this is possible this would require major changes in other kernels, and possible degradation of performance of these kernels due to problems of multicoloring that we discussed above. It has to be kept in mind that it is common in algorithms that we consider multiple kernels may be used alongside KACZ, so optimizing for a selected kernel alone is not sufficient. Another possibility is to avoid using OpenMP across sockets but instead use different MPI processes on each socket.

Remark: Symmetric vs Unsymmetric D2 coloring

Even though only row permutations are important for KACZ algorithm, one can also permute the columns in any desired way without affecting structural orthogonality, and hence the results. A permutation that is favourable in many respects is one that is symmetric, i.e., that permutes rows and columns in the same way. A symmetric permutation causes less confusion when dealing with permuted vectors (see Chapter 6).



(a) Unpermuted Matrix **(b)** Unsymmetric Permutation **(c)** Symmetric Permutation

Figure 5.6: Equivalence of Unsymmetric and Symmetric permutation with respect to D2 coloring

Figure. 5.6 illustrates two different kinds of column permutation being applied to the matrix in Fig. 5.6a. In Fig. 5.6b the rows are only permuted according to D2 coloring, while the column permutation remains the same. This kind of permutation is referred to as unsymmetric. On the other hand in Fig. 5.6c the columns are permuted in the same way as the rows, which is referred to as symmetric permutation. Note that in both cases the rows within a color are structurally orthogonal.

One should keep in mind, however, that a symmetrically permuted multicolored matrix bears the danger of more irregular x vector accesses and increased false sharing, due to increased scattering of column index compared to unsymmetrically permuted matrix (compare Fig.5.6b and 5.6c).

5.3 Implementation

Multicoloring in the GHOST[44] library is carried out using an external library called COLPACK [2]. The preprocessing steps for multicoloring are as follows.

1. Row pointers and column entries are created with the help of callback functions.
2. Call COLPACK routines to do a Distance 2 coloring from the CRS format.
3. The appropriate color pointer (pointer to start of each color), number of colors, along with permutations and inverse permutations are calculated and stored.
4. The matrix is assembled in the desired format with the current permutations.

In our implementation one can easily switch between symmetric and unsymmetric permutation as desired. Once we have the color pointers (color_ptr), number of colors(ncolors) and the permuted matrix, we can implement our KACZ_MC (KACZ with multicoloring) sweep. The only difference from a normal KACZ sweep is an extra loop over the colors as shown in algorithm 5

5.4 Test Bed

Most of the tests presented in this thesis are conducted on the RRZE's Emmy Cluster. The cluster features 560 compute nodes, each of which are equipped with two Intel Xeon E5-2660V2 chips and 25 MB shared

Algorithm 5 KACZ_MC algorithm

```
{dir - refers to the direction of sweep, Forward or Backward}
if $dir == FORWARD$ then
    $start = 0, end = ncolors$
    $stride = 1, offset = 0$
else
    $start = ncolors, end = 0$
    $stride = -1, offset = -1$
end if
for $clr = start : stride : end$ do
    $start\_row = color\_ptr[clr] + offset$
    $end\_row = color\_ptr[clr + stride] + offset$
    #pragma omp parallel for
    $KACZ\_SWEEP(start\_row, end\_row, stride)$
end for
```

cache per chip and 64 GB of RAM. Each chip consists of 10 SMT enabled physical cores. The nodes are connected by a QDR Infiniband network with 40 GBit/s bandwith per link and direction. For the purpose of reproducibility of the results the clock frequency was fixed at 2.2 GHz for all the tests. Stream bandwidth [47] per socket is measured to be 41 GB/s (triad). Intel Compiler version 15.0.5.223 is used and the following performance flags are set: `-fno-alias -xHost -O3`.

5.5 Results and Discussions

Baseline

Simply presenting results without any proper comparisons or baseline would be meaningless, so to compare and to get an idea of the overall quality of the results we compare the KACZ algorithm with Sparse Matrix Vector Multiplication (SPMV), and Sparse Matrix Transpose Vector (SPMTV). The choice of SPMV and SPMTV as baseline is due to the similarity in the algorithms. Algorithm 6 provides a side-by-side comparison of these three algorithms in CRS matrix format.

From the comparisons in Alg. 6 we can easily see that for each non-zero in the matrix both SPMV and SPMTV requires 2 FLOPs (1 ADD and 1 MUL). On the other hand for a normal KACZ we need 6 FLOPs per non-zero, and 1 DIV and 1 extra MUL operation per row. However, for the KACZ algorithm the calculation of $norm$ (or $rownorm$) can be avoided (see the lines colored in blue) if the system is normalized at start, reducing the number of FLOP to 4 FLOPs per non-zero. So in general the performance metric is chosen as follows:

$$P_{\mathrm{SPM(T)V}} = \frac{2 * nnz}{time}\text{Flops/sec} \qquad P_{\mathrm{KACZ}} = \frac{4 * nnz}{time}\text{Flops/sec}$$

If one assumes that the memory data traffic is mainly composed of non-zero's and the matrix is well-behaved with respect to the α factor, one could expect the same amount of data traffic per iteration for KACZ and SPM(T)V. This implies that if nothing hinders the performance and the code is memory bound, which is the case in general for SPMV like algorithms, one would expect a factor of 2 for KACZ algorithm compared to SPM(T)V. Note that even in case with rownorm computations we use the same performance metric for easy interpretation of the results, so in ideal situation one can only expect a factor of 2 (not 3) between SPMV and KACZ.

Although this factor of 2 might slightly drop if we have also calculate $rownorm$, but we cannot expect a large difference since all the data needed for the norm calculation is readily available in cache.

It is worth noting that SPMTV like KACZ has data dependencies, and requires multicoloring or some other technique to parallelise it.

Algorithm 6: Comparison SPMV, SPMTV, and KACZ

SPMV: $Ax = b$
```
for $row = 0 : nrows$ do
    temp = 0
    for $idx = rowptr[row] : rowptr[row + 1]$ do
        $temp+ = val[idx] * x[col[idx]]$
    end for
    $b[row] = temp$
end for
```

SPMTV $A^{\intercal}x = b$
```
Ensure: $b = 0$
    for $row = 0 : nrows$ do
        $temp = x[row]$
        for $idx = rowptr[row] : rowptr[row + 1]$ do
            $b[col[idx]]+ = val[idx] * temp$
        end for
    end for
```

KACZ: solve for x: $Ax = b$
```
for $row = 0 : nrows$ do
    $scale = b[row]$
    $norm = 0$
    for $idx = rowptr[row] : rowptr[row + 1]$ do
        $scale- = val[idx] * x[col[idx]]$
        $norm+ = val[idx] * val[idx]$
    end for
    $scale* = omega/norm$
    for $idx = rowptr[row] : rowptr[row + 1]$ do
        $x[col[idx]]+ = scale * val[idx]$
    end for
end for
```

Choice of Matrix and Permutations

In order to study the behavior of multicoloring on a general basis we choose 2 representative matrices: PWTK (see Appendix 10.1.3) and Graphene-4096-4096 (see Appendix 10.1.2). These were selected because they reflect two different class of matrix that appear in our project: Graphene being stencil-like and PWTK being more general without a simple structure.

As mentioned at the end of Section 5.2, the choice of symmetric or unsymmetric multicoloring also plays a vital role. In order to understand this influence we include results for both the symmetrically and unsymmetrically permuted PWTK matrix. Graphene being stencil like does not show a significant performance degradation for symmetric permutations, thus we include results only for the unsymmetrically permuted Graphene matrix.

To distinguish between the two permutations we denote symmetric multicoloring by "MC_symm" and unsymmetric multicoloring by "MC".

Node-level performance

The KACZ algorithm is run in symmetric fashion for all the tests, i.e. forward sweep (rows are projected from 1^{st} row to last row) followed by a backward seep (rows are projected from last row to 1^{st} row), since this is the most relevant scenario in our applications. The runs are conducted on one node (2 sockets) of Emmy with thread 1 to 10 pinned to socket 0 and thread 11 to 20 pinned to socket 1, ignoring the Hyper-Threading feature.

Figure 5.7 might be surprising initially, since this is not what we would expect from our prediction, of KACZ being 2 fold higher than SPMV (see Section 5.5). One could also observe that the difference between

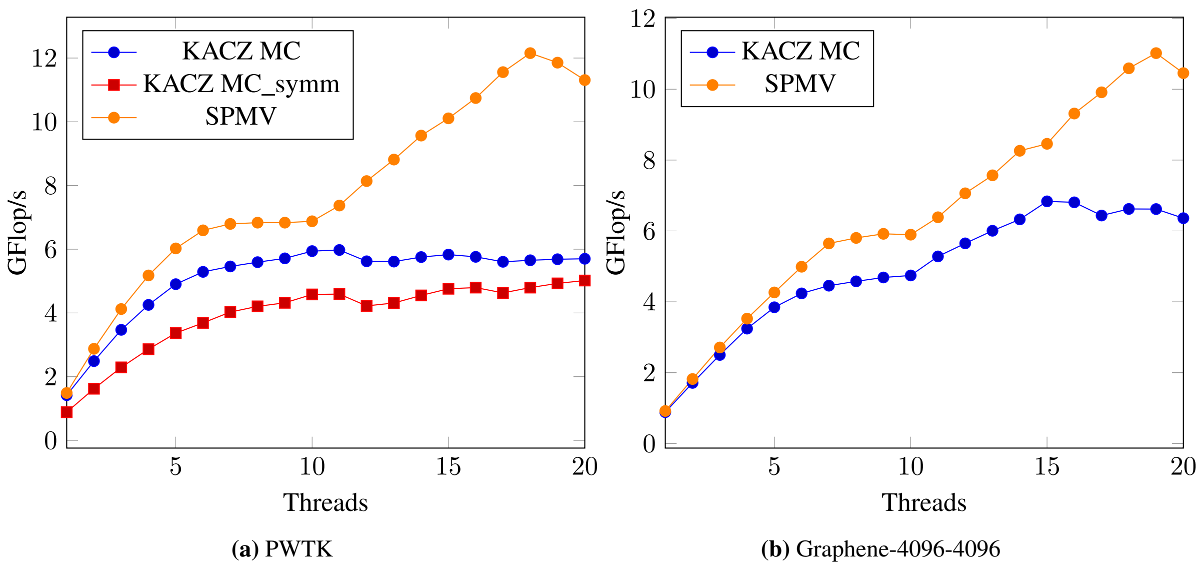


Figure 5.7: Scaling Test on 1 node of Emmy at 2.2 GHz for PWTK matrix (Fig. 5.7a) and Graphene-4096-4096 matrix (Fig. 5.7b).

KACZ_MC and SPMV grows as the number of threads increase and lack of (or no) scaling across sockets. To understand the disparity between our naive prediction and measurement we need to look back at the performance issues described in Section 5.2 and do some extra tests and measurements.

α Effect & Irregular access

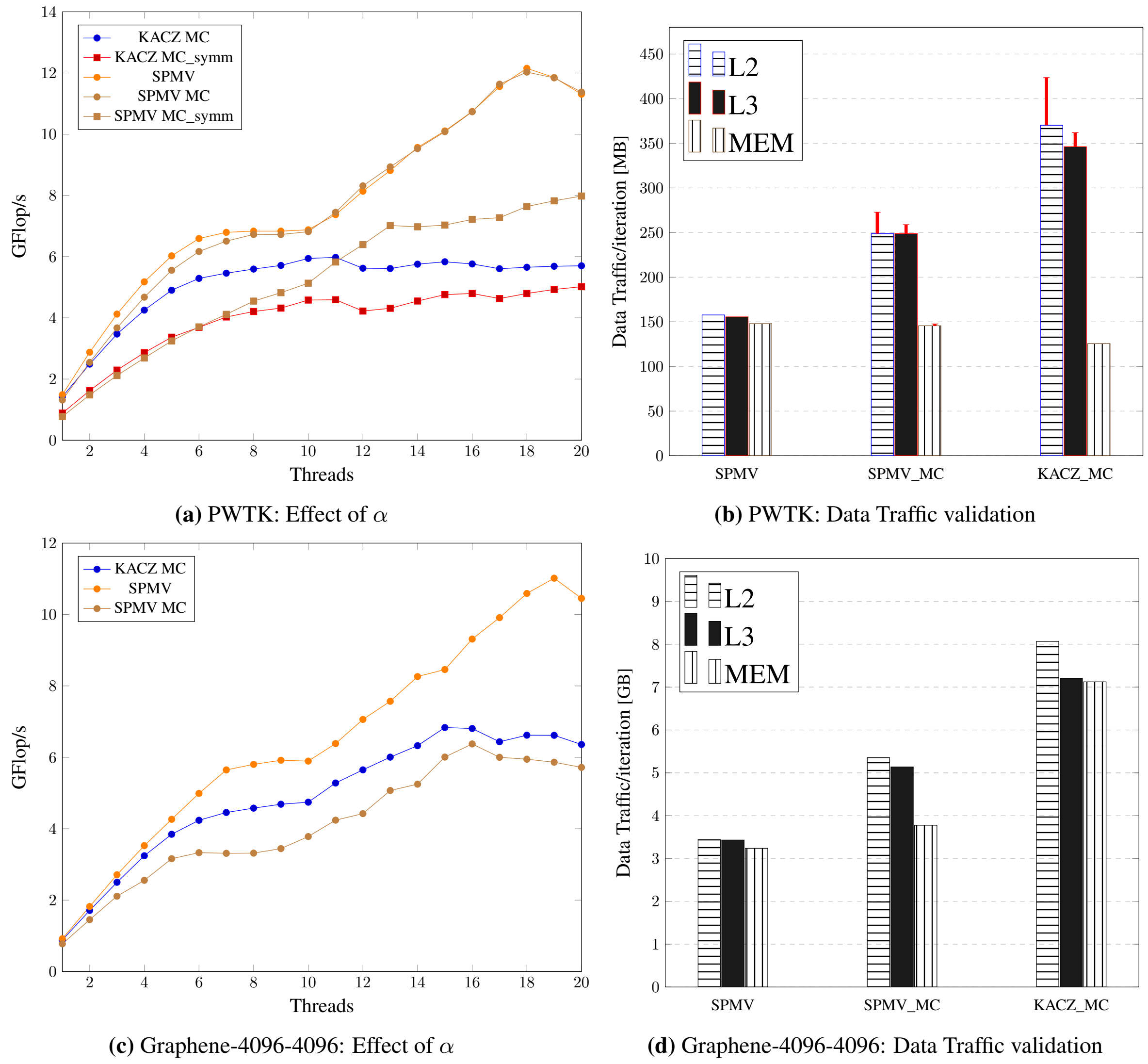
To quantify extra data traffic and random access, due to multicoloring we conduct a SPMV scaling test on a multicolored matrix as opposed to the unpermuted matrix in Fig.5.7.SPMV is not prone to false sharing, barrier effects and other disadvantages due to indirect stores, the difference in performance between permuted and unpermuted matrix gives us a general idea of the effect of these two factors. In Fig.5.8a and 5.8c the difference between orange line and brown line shows this effect. We see that even the normal SPMV operation is affected by the matrix permutation.

One can note that unsymmetrically permuted PWTK matrix is not affected much as compared to the Graphene-4096-4096 matrix by this α factor. In order to understand the reason we need to look at the likwid-perfctr hardware performance counter[63] measurements shown in the Fig.5.8b and 5.8d. Here we see for the PWTK matrix data traffic from L3 and L2 increases by $1.6\times$ in the case of SPMV on MC permuted matrix, while that from the main memory remains the same. Memory data volume remains unaffected due to the fact that number of rows (217,918) in the matrix is sufficiently small to fit in the L3 cache. According to ECM (Execution Cache Memory) model [58], the algorithm can scale linearly until it hits a non-scalable bottleneck. Since for Ivy Bridge (Emmy) all the caches are scalable this increase in L3 and L2 shouldn't make a big difference, it would only delay the saturation. Consequently, the saturation performance is the same, as seen in Fig. 5.8a.

For symmetrically permuted PWTK matrix the situation is different, here the SPMV on permuted matrix does not even saturate the memory, even though there is not a major increase in data traffic compared to symmetrically permuted case (see the red and blue markings in fig. 5.8b, representing increase in data traffic for symmetric permutations). We think the large drop in this case is due to irregular x-vector accesses and associated latency.

On the other hand for Graphene matrix we see almost a $1.2\times$ increase in data traffic from Memory and $1.5\times$ from L3 and L2 for SPMV on multicolored matrix. Since main memory($\alpha(A, MEM)$) is affected in this case we can expect a drop in performance even in the saturation regime. This drop along with the penalties due to irregular accesses contribute to overall performance drop seen in Fig. 5.8c.

On the other hand for KACZ_MC algorithm the data traffic is again considerably higher than that of SPMV on multicolored matrix, which arises as a result of KACZ algorithm having extra stores to x (scatter operations), in addition to the loads(gather), making the α factor all the more worse.



(a) PWTK: Effect of α

(b) PWTK: Data Traffic validation

(c) Graphene-4096-4096: Effect of α

(d) Graphene-4096-4096: Data Traffic validation

Figure 5.8: α effect on PWTK and Graphene-4096-4096 matrices: Difference between SPMV on unpermuted matrix (orange line) and SPMV on MC permuted matrix (brown line) quantifies increase in data traffic (Fig. 5.8a and 5.8c). LIKWID measurements validate the observation (Fig.5.8b and 5.8d).Extra data traffic due to symmetric permutation is marked in fig.5.8b with red markings.

False sharing and Global Barriers

Although increase in α factor plays a role in the performance drop, it is not sufficient to describe the kind of drop seen in Fig. 5.7, especially for unsymmetrically permuted PWTK matrix where we observed that the α factor did not play a great role. If we focus on a single socket (i.e. up to 10 threads on Emmy), we see substantial differences in the saturation pattern between SPMV and KACZ(or SPMTV). The KACZ algorithm does not scale at all after 5 or 6 threads, which is partly due to false sharing. Figure 5.9 validates this fact using LIKWID measurements (cyan colored dotted line), where we see an increase in the false sharing as the number of threads increase. It has to be noted that LIKWID measurements for counters like FALSE_SHARING, might not be 100% accurate, but they provide a good qualitative estimate for comparison. More on counter validation can be found in [55].

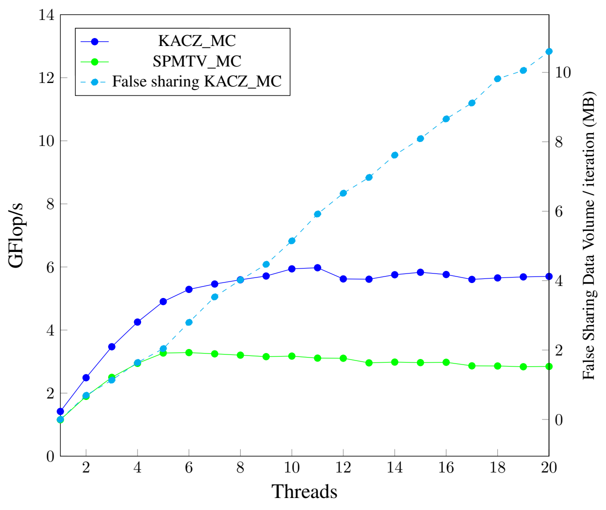
It is interesting to note that the amount of false sharing for symmetric case (compare Fig. 5.9a and 5.9c) almost increases by a factor of 3 compared to unsymmetric permutation due to the effect mentioned at the end of Section 5.2. Yet another interesting fact is the comparatively smaller amount of false sharing for stencil-like matrix, here Graphene-4096-4096 compared to PWTK matrix (compare Fig. 5.9e and 5.9a) strengthening our observation in Section 5.2.

Another important influence in performance is due to global barriers at the end of each color. Even though avoiding global barriers would lead to obviously wrong answers, for the sake of quantifying barrier pain we use the OpenMP `.nowait.` clause and observe the difference to the original version. However, here we cannot completely decouple from the effect of false sharing since removing barriers (or synchronizations) would allow threads to order themselves in a way as to reduce the effect of false sharing. This effect for example is seen prominently for unsymmetrically permuted PWTK (see Fig. 5.9b) matrix where the likwid-perctr measurements of false sharing dropped by a factor of 10 (not shown), therefore making it unable to isolate the effect. But for symmetrically permuted PWTK matrix the amount of false sharing dropped only slightly after taking away the barrier, enabling us to make a rough estimate on the effect of global barriers (see Fig. 5.9d). As stated in Section 5.2 the influence of chromatic number on global barrier can be seen from a comparison of Graphene-4096-4096 containing 22 colors and 16,777,216 rows (Fig.5.9f) with PWTK matrix containing 180 colors and 217,918 rows (Fig.5.9d).

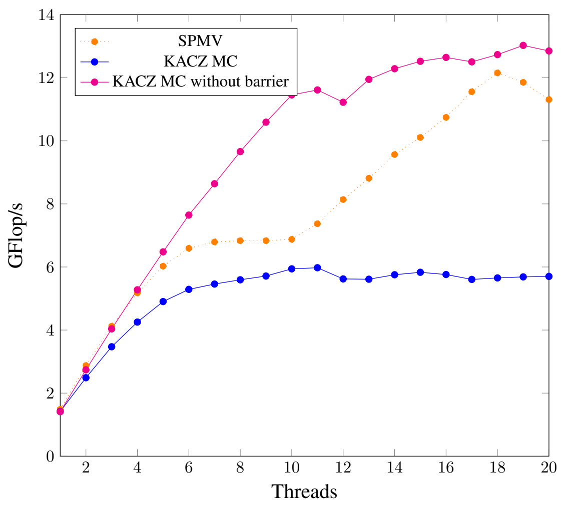
Improper Data Placement and Scaling across Sockets

In all the above tests the data in the NUMA domains was placed in a way suitable for SPMV since most of the algorithms rely on this placement. But as we have seen in Section 5.2 this is not ideal for KACZ or SPMTV. An important characteristic of poor data placement, is that the KACZ or SPMTV performance does not scale across the socket even though there is a 2 fold increase in memory bandwidth due to the availability of a new memory interface (see Fig. 5.8a and 5.8c).

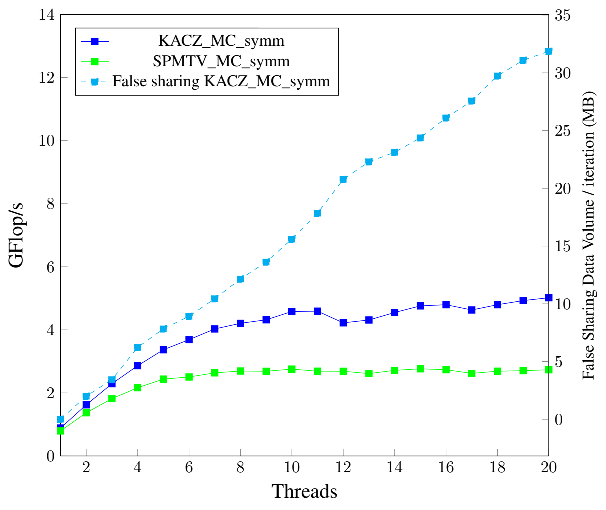
Moreover the problem of false sharing and global barriers aggravates because of remote data. And in general it is much expensive than the local effects we saw in the previous subsections. Taking into account these problems across the NUMA domains, it is advisable to use CARP with MPI for multicoloring algorithms as mentioned in Sec. 5.2.



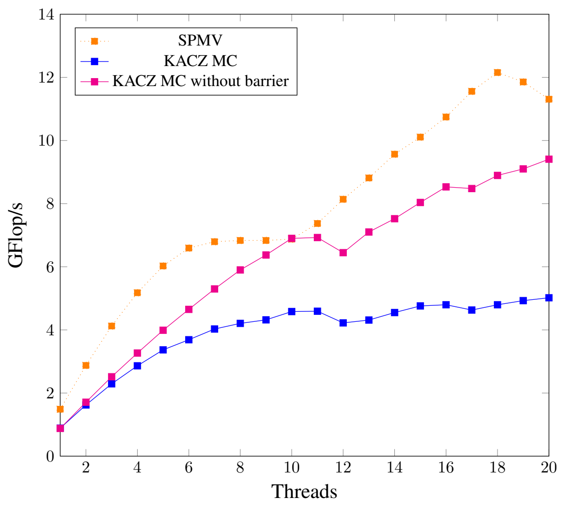
(a) PWTK: KACZ_MC False Sharing



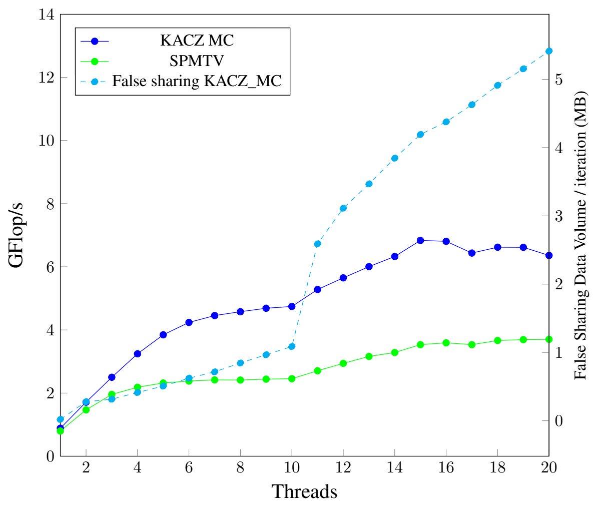
(b) PWTK: KACZ_MC Effect of Global Barrier



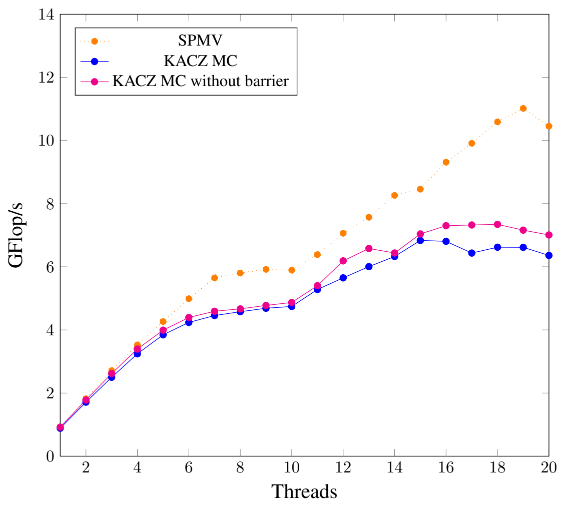
(c) PWTK: KACZ_MC_symm False Sharing



(d) PWTK: KACZ_MC_symm Effect of Global Barrier



(e) Graphene-4096-4096:KACZ_MC False Sharing



(f) Graphene-4096-4096:Effect of Global Barrier

Figure 5.9: False Sharing and Barrier Effect: Fig. 5.9a, 5.9c and 5.9e shows LIKWID measurements of False Sharing Data Volume of KACZ_MC algorithm for unsymmetric and symmetric Multicolored PWTK matrix and Graphene-4096-4096 respectively. Fig. 5.9b, 5.9d and 5.9f shows the effect when barrier at the end of each color is removed.

6 Block Multicoloring

In the previous Chapter 5 we have seen a popular approach currently employed in the scientific community to deal with the problem of data dependency. But the analysis of this method has shown that even though this method preserves the correctness of the KACZ algorithm, performance is far from optimal. Having in mind the lessons we learned from the analysis of the MC methods, in this chapter we try to formulate a method which could avoid these performance impacts.

6.1 Basic Idea

One of the major problems of multicoloring algorithms is their tendency to scatter the data, but in the approach described here we try to do exactly the opposite, i.e., we try to preserve the banded form of the matrix. Moreover the proposed algorithm in this section works only if a certain requirement on its banded structure is enforced. But later in the upcoming sections we will try to relax this constraint. Also we have seen that the performance of the MC algorithm depends on the chromatic number of the matrix, the lower the number the better should be the performance. Another important fact is that we need just enough parallelism to match the maximum available number of threads on the hardware.

Taking into consideration all these facts, we propose that the best idea would be to have $nthreads$ independent blocks, on which each thread can operate without any write conflicts. This can be achieved with two block colors if the following condition is satisfied:

$$\frac{nrows(A)}{2 * bw(A)} \geq nthreads; \tag{6.1}$$

$$\begin{aligned} nrows(A) &- \text{number of rows of matrix A} \\ bw(A) &- \text{lower + upper bandwidth of A} \\ nthreads &- \text{max. threads available} \end{aligned}$$

To understand the basic principle we consider the following definitions and Lemma.

Lower bandwidth: The lower bandwidth of a matrix A is defined as the smallest positive integer k such that $A_{i,j} = 0$ whenever $j < i - k$.

Upper bandwidth: The upper bandwidth of a matrix A is defined as the smallest positive integer l such that $A_{i,j} = 0$ whenever $j < i + l$.

Bandwidth[1]: The bandwidth bw of a matrix A is defined as :$bw = k + l$

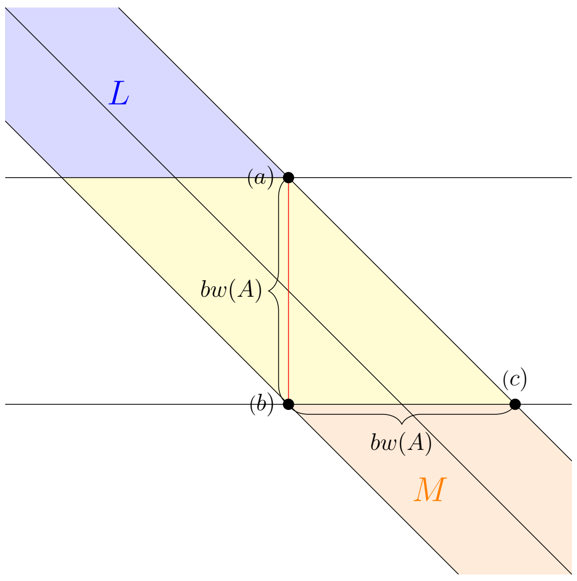
Block of a matrix[2]: A block of a matrix A is a submatrix of A containing a subset of rows in A.

Structurally Orthogonal Blocks: Two blocks L and M of a matrix A are said to be structurally orthogonal if:

$$|L(i,c)| * |M(j,c)| = 0; \quad \forall i \in [1; nrwos(L)],\ \ j \in [1 : nrows(M)]\ \ and\ \ c \in [1 : ncols(A)]^3 \tag{6.2}$$

Lemma 6.1 : Any two blocks L and M of a square matrix A are structurally orthogonal if they are separated by a distance of bandwidth $(bw(A))$.

Proof. Consider a matrix $\hat{A}$ with all its blocks having the same bandwidth $bw(A)$, i.e., matrix $\hat{A}$ is an envelope matrix of A. Now it holds that L is a structurally orthogonal block to M, if the maximum of all non-zeros' column indices in L (say at point a) is less than the minimum of the non-zeros' column indices in M (say point b) or vice versa. Figure 6.1 illustrates this.



a- maximum of non-zeros' column indices in L
b- minimum of non-zeros' column indices in M

Figure 6.1: Illustration of Lemma 6.2

It is straight forward to see that for a square matrix $\hat{A}$ the distance between the row containing a and the row containing b is equal to the row width of the matrix $\hat{A}$ (distance between points b and c) which is nothing but $bw(\hat{A}) = bw(A)$. Thus for the matrix $\hat{A}$, two blocks are mutually structurally orthogonal if they are separated by a distance $bw(A)$.

Since the Lemma holds for the envelope matrix $\hat{A}$, it must also hold for any matrix A with bandwidth, $bw(A)$. □

[1]Note that the **bandwidth** referred throughout this chapter is the **matrix bandwidth** which is the sum of lower and upper bandwidth. Any reference to **memory bandwidth** would be mentioned as such.

[2]Note: This definition is slightly different from the conventional definition of Block Matrix, since here $ncols(A)$ = $ncols$(block of A)

[3]$nrows(A)$ - number of rows of the (sub)matrix A, $ncols(A)$ - number of columns of the (sub)matrix A

Eqn. 6.1 ensures that the given matrix A can be split to $2 * nthreads$ equal blocks, such that any two blocks separated by at least one other block are a distance of $bw(A)$ apart, thus being structurally orthogonal.

Figure 6.2, illustrates how this works for a matrix with 24 rows and a bandwidth of 3.

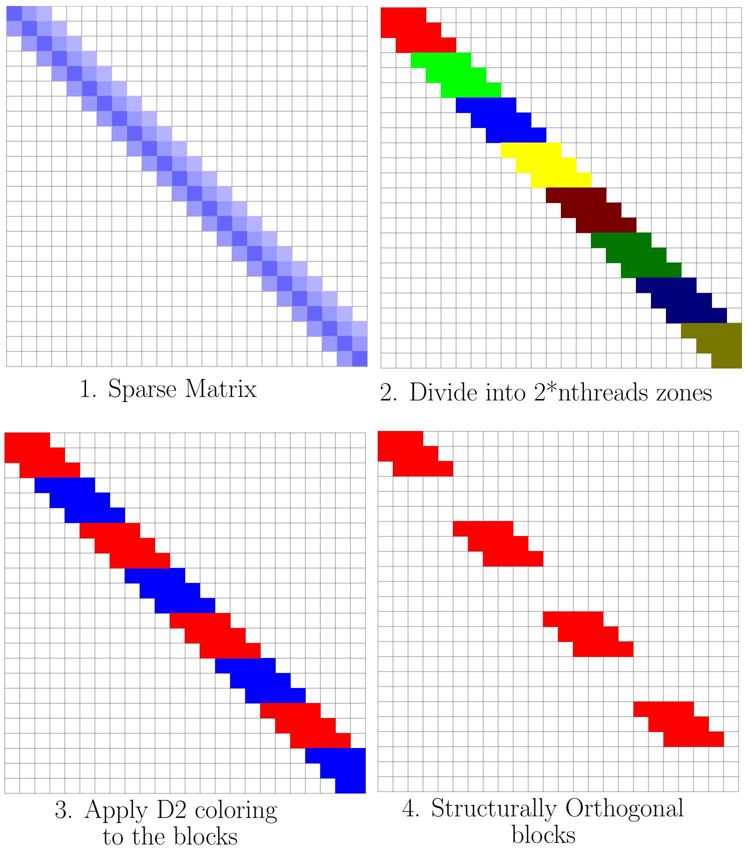

1. Sparse Matrix
2. Divide into 2*nthreads zones
3. Apply D2 coloring to the blocks
4. Structurally Orthogonal blocks

Figure 6.2: Basic idea of block multicoloring(BMC)

In the matrix shown in Fig. 6.2, the maximum number of threads that can be used is $\frac{nrows(A)}{2*bw(A)} = \frac{24}{2*3} = 4$. If this condition is satisfied we see that all the blocks within a color (say red) are separated by a distance of bandwidth. Application of Lemma 6.1 further yields that the bocks within a color are completely independent (structurally orthogonal blocks), thus enabling parallel update within a color. The proposed algorithm on this BMC pre-processed matrix assigns each block in a color to one of the four threads available and performs a KACZ sweep alternately on all red blocks and blue blocks.

The name block multicoloring derives from the following observation: if we consider a single block shown in Fig.6.2.2 as a node in the graph (i.e., here with total of 8 nodes), and apply distance-2 coloring to this graph[4] we obtain 2 colors as shown in Fig. 6.2.3. However, one should note that even though the blocks are totally independent, there is no further parallelism within a block, since the rows within the blocks are strongly connected. This disables SIMD or any other kind of parallelism within a single core.

6.2 Related Work

The roots of this kind of partitioning scheme can be traced back to early 1980's. A general study on the convergence of block methods (not necessarily orthogonal blocks) was done by Elfving in 1980 [31]. Later in 1988 Kamath and Sameh introduced a two-block partitioning scheme (similar to the approach above)

[4]Note that the graph is now directed and coloring will have to be applied to the bipartite graph

for Kaczmarz method on tridiagonal structures [41]. A similar block partition approach was also used for the Cimmino method in 1990's [18, 19] and some recent works in which they study about augmenting the system for Cimmino method to enhance the condition number of the matrix can be found in [30].

In this thesis we have extended the method for general matrices and established a matrix dependent bound (constraint) on the parallelism that could be extracted for such kind of partitioning schemes. In the upcoming sections we try to relax this bound, enabling us to extract further parallelism. Later, we will see the trade-off between performance and parallelism that could be achieved with such kind of approach.

6.3 General variants

Most of the sparse matrices that emerges from the applications in the ESSEX project, satisfy the condition (naturally or by some pre-processing, see Section 6.4) given by Eqn. 6.1 for the maximum threads available within a socket or node. Table 6.3 shows the maximum number of threads that can be used for different sizes of the Graphene matrix. Here we see that starting from size 256 (very small) one could use all the available physical cores on the Ivy Bridge CPUs in the Emmy cluster, and starting at 512 almost all the modern standard multi-core CPUs would be covered.

Graphene size	$nrows = bw$	bw_{RCM}	Max. Threads
16	510	-	1
40	3198	238	3
64	8190	382	5
128	32766	766	10
256	131070	1534	21
512	524286	3070	42

Table 6.1: Graphene (PBC) size vs max. possible threads for 2 colored BMC, with RCM. bw_{RCM} is the bandwidth of matrix after applying RCM (see Section 6.4)

But there are matrices and situations like periodicity, where this might be difficult to attain. Thus in this section we try to relax the condition and generalize the scheme for general sparse matrices, with arbitrary bandwidth. A basic approach for this is based on the divide and conquer rule, where we divide our matrix in a way to find more structurally orthogonal blocks.

As shown in Section 6.1, one could continue to divide the matrix into more blocks by keeping at least blocks of a single color (say red in Fig. 6.2) structurally orthogonal if the following condition is satisfied:

$$\frac{nrows(A)}{bw(A)} \geq nthreads \tag{6.3}$$

The targeted blocks (say red in Fig. 6.2) are ensured to be structurally orthogonal due to the above condition since blocks assigned to each thread can be separated by a distance of bandwidth, and hence by Lemma 6.1 we see they are structurally orthogonal. We call this zone (or blocks) the **pure zone** (blocks), denoted as P_k, where subscript k represents the k^{th} block. But the above procedure would create blocks in between (analogous to blue blocks in Fig. 6.2) which need not be structurally orthogonal to each other. This zone (blocks) is called the **transition zone** (blocks), denoted as T_k. Note that as the number of threads grows the block size (number of rows in a block) in the transition region grows at the expense of the shrinking pure zone.

Even though structural orthogonality with adjacent (neighbor) blocks in the transition zone cannot be ensured, one could ensure structural orthogonality of the block with its alternate neighbors in the transition zone (i.e. even or odd numbered transition zone are structurally orthogonal). The following Lemma shows this.

Lemma 6.2 : A transition block is structurally orthogonal to its alternate neighbors.

Proof. we know, $P_k < T_k < P_{k+1}$
$\implies P_{k+2} < T_{k+2} < P_{k+3}$
but, $P_k \perp^{s}$[5] $P_{k+1} \perp^s P_{k+2} \perp^s P_{k+3}$
$\implies T_k \perp^s T_{k+2}$ □

This would then enable half the available threads (all the even-numbered threads) to work simultaneously and the next half (odd-numbered threads) after this. Fig. 6.3 shows various zones that appear for Graphene-20-20 matrix in two scenarios: Fig. 6.3 a,b,d corresponding to three threads and Fig. 6.3 a,c,e corresponding to four threads . Choice of this small matrix ($nrows = 400$) makes it convenient to illustrate all the emerging zones for smaller $nthreads$. Here the orange colored regions are separated from each other by a distance of bandwidth (bw) which is why we call them the pure zone P_k, while the blue colored zone that separates this pure zone is the transition zone T_k. It is interesting to observe the growth of the transition zone and corresponding shrinking in the pure zone as the number of threads increase (compare Fig. 6.3.b and 6.3.c), also if one observes carefully one could see that the blocks in transition zone overlap with the nearest neighboring blocks (in transition zone) but blocks in pure zone are structurally orthogonal to each other.

Using half the threads means under-utilization of the available resources, and we would like to avoid this situation as much as possible. A careful look into the transition zone reveals that this could be avoided to a certain extent if we further divide the transition zone, thereby enabling the use of all the available threads until a certain limit below the condition stated in Eqn.6.3. In order to achieve this the transition zone is further classified into 3 zones: **red-trans zone**, **black-trans zone** and **trans-trans zone** after the initial coarse classification. The red and black-trans zone is like the pure zone, each block being structurally orthogonal to each other. However, trans-trans zone is necessarily not. The overall idea of classification is to separate out regions in the transition zone that can be done in parallel. For example in Fig. 6.3.b and 6.3.c one can observe some of the initial rows in T_1 and T_2 can be done in parallel (since they are structurally orthogonal) before hitting some conflicting rows. To determine this classification we just need to determine 2 points within the transition zone, namely the end of the red-trans zone (re) and the start of the black-trans zone (bs) as explained in the following.

Red and Black-Trans zone

Our aim is to construct the red (black)-trans zone such that it is structurally orthogonal to all other red (black)-trans blocks. In order to do this we first fix the start (rs_k) of red-trans zone (RT_k) as the start of trans zone (T_k) and the end (be_k) of black-trans zone (BT_k) to the end of trans zone (T_k), see Fig. 6.3.d and 6.3.e. To achieve structural orthogonality ($\perp^s$) between all the red (black)-trans zone we just need to ensure that the current block (for e.g. RT_k) is structurally orthogonal to one of its immediate neighboring block in the same zone (RT_{k+1} or RT_{k-1}). In our approach we try to make red-trans zone $\perp^s$ to the neighboring block following it and black-trans zone $\perp^s$ to neighboring block preceding it. Now by using Lemma 6.1 re and bs are derived as follow:

$$\begin{aligned} re_k &= rs_{k+1} - bw \\ bs_k &= be_{k-1} + bw \\ where, rs_{k+1} &= \text{row index corresponding to start of the block } T_{k+1} \\ be_{k-1} &= \text{row index corresponding to end of the block } T_{k-1} \end{aligned} \tag{6.4}$$

If $re_k > bs_k$ (as seen in Fig. 6.3.d), we can move re upward and bs downward without disturbing any structural orthogonality and we recompute re_k and bs_k as their average $\frac{re_k+bs_k}{2}$. Once re and bs are determined the zones are determined as: red-trans block (RT_k) comprising rows that span from rs_k to re_k and correspondingly black-trans block (BT_k) zone bs_k to be_k. Fig 6.3.d shows this clearly.

[5] $\perp^s$ - represents structurally orthogonal

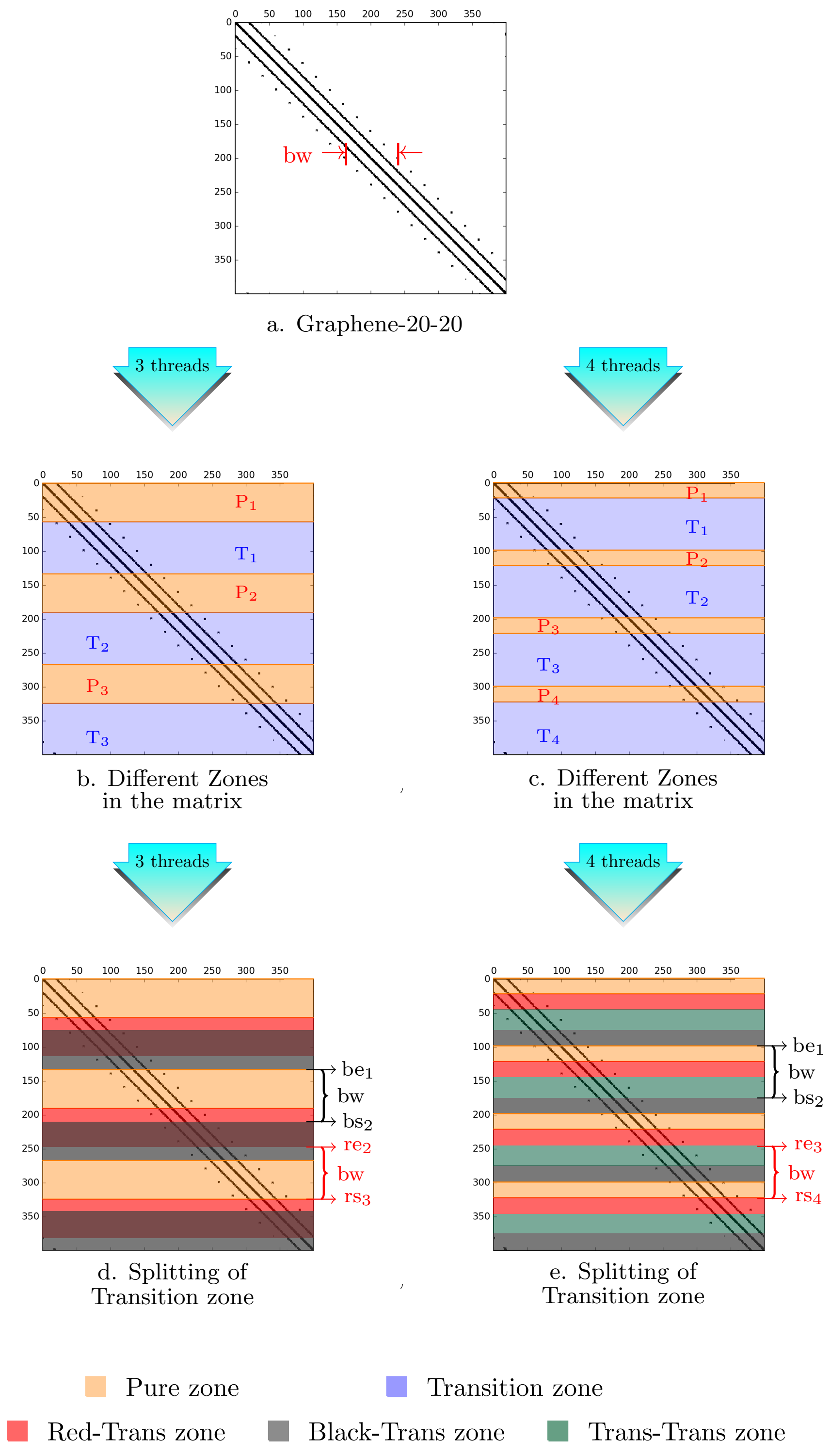


Figure 6.3: Illustration of various zones appearing in block multicoloring, Left side figures Fig. 6.3b,.d depicts the splitting for 3 thread case and figures to right Fig. 6.3.c,.e for 4 thread case

Trans-Trans zone

Trans-trans zone (TT) is the zone that appears between RT and BT if $re < bs$. The rows between re_k and bs_k then consist of the TT_k block. For our example on Graphene-20-20, we do not see any TT zone for $nthreads = 3$ (see Fig.6.3.d), since $re_k > bs_k \; \forall \; k \in [1, nthreads - 1]$. But once this condition is violated as we see for $nthreads = 4$ in Fig.6.3.e, we get a TT zone. The regions in this zone are not necessarily block structural orthogonal to each other, and a simple check for orthogonality of TT zone will be given in Section 6.6. However, since trans-trans (TT) zone is a sub-block of trans zone Lemma 6.2 holds, thereby definitely enabling at least half the available threads to work in parallel.

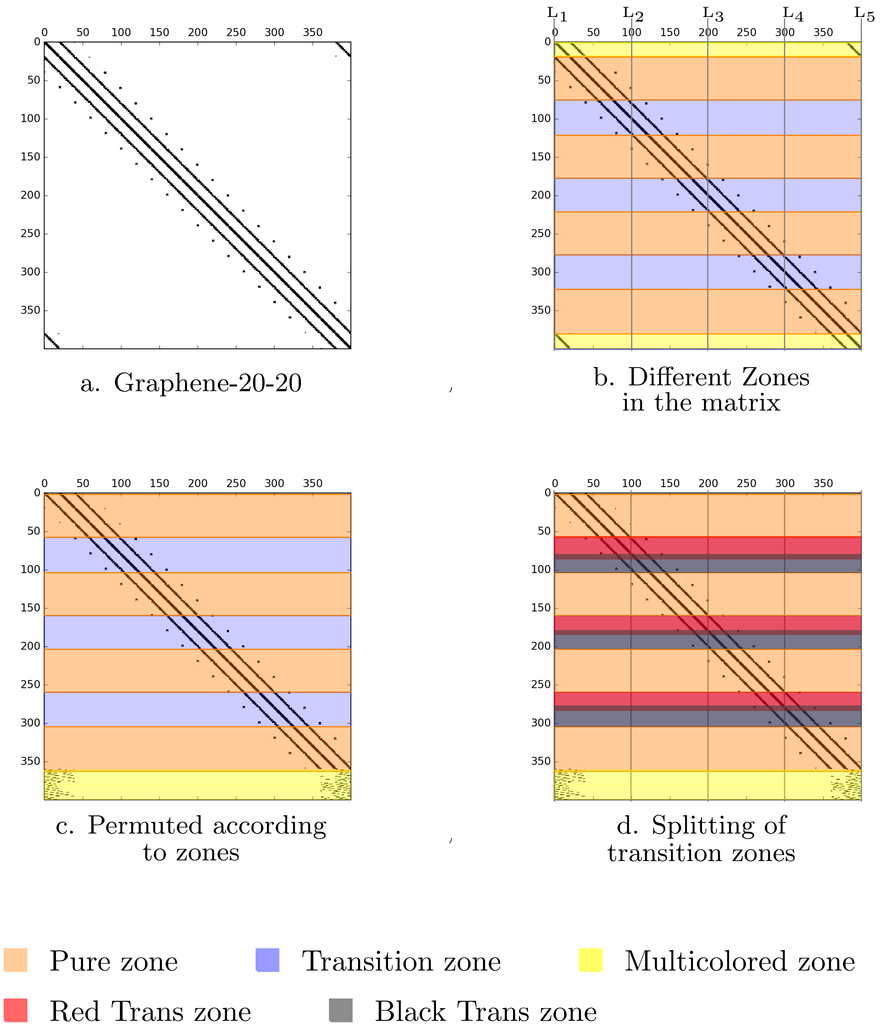


Figure 6.4: BMC on Graphene-20-20 for 4 threads

Emergence of Multicoloring

Even though the original condition is relaxed by a factor of two by the above approach, we might still encounter matrices which do not satisfy the condition stated in Eqn. 6.3. So for a matrix that violates this condition, we implicitly substitute $bw(A)$ with an effective bandwidth $bw_{eff}(A)$ (procedure to find $bw_{eff}(A)$ is discussed in the next subsection) depending on $nthreads$ and distinguish between the rows that do and do not satisfy the condition. Eventually the rows that do not satisfy the condition 6.3[6] are

[6]Note: $bw(A)$ is substituted with effective bandwidth $bw_{eff}(A)$

permuted to the end, and the normal multicoloring would be applied to these rows. This approach enables us to enjoy all the benefits of BMC method to its maximum and at the same time eliminate all the constrains required. For most of our test matrices the percentage of rows multicolored is really small, and in most cases this arises on rows that correspond to periodic boundary conditions (PBC). By the above hybrid method such conditions can be treated without any (or negligible) performance drop due to MC, since except for boundary rows (here) BMC method can be employed on all other rows.

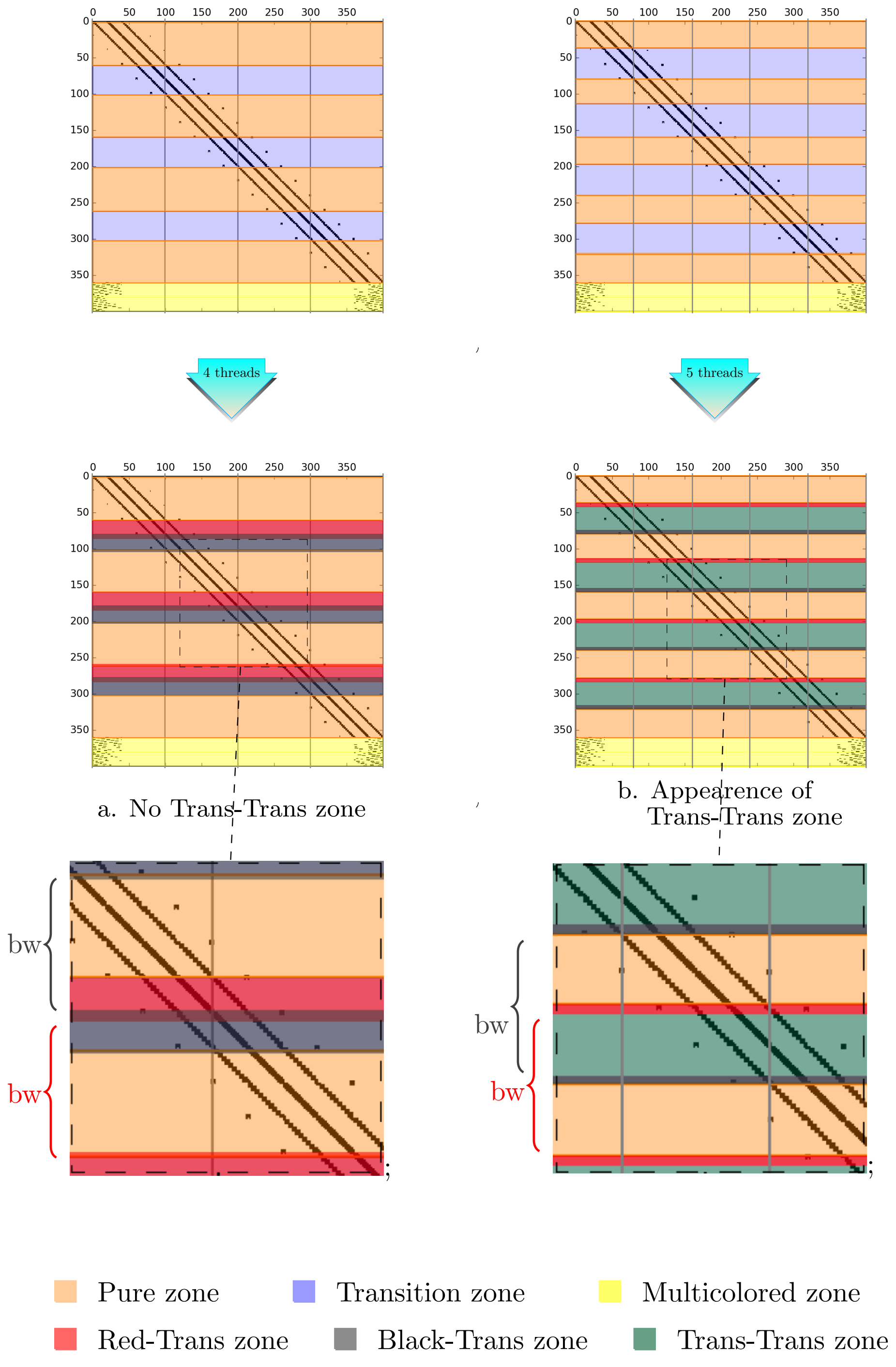


Figure 6.5: :Comparison between 4(Left) and 5(Right) threads

Fig. 6.4 illustrates the procedure to split the transition zone and filter out the rows to be multicolored, for Graphene-20-20 matrix with periodic boundary conditions. Due to its periodicity we see a small off-diagonal to the extreme top-right and bottom-left shown in Fig.6.4.a. This blows up the bandwidth to almost 2*nrows, thereby violating the condition in Eqn. 6.3 for $nthreads \geq 2$. Therefore we have to replace $bw(A)$ with effective bandwidth $bw_{eff}(A)$ as discussed in the following.

Finding the effective bandwidth

A clear choice of effective bandwidth is the bandwidth of the sub-matrix excluding the rows corresponding to multicolored zone (see Fig.6.4.a, here the rows corresponding to the boundary). But initially determining this for an arbitrary matrix might not be straight-forward since the number of multicolored rows and the effective bandwidth have a cyclic dependency: in order to know the rows to be multicolored we need effective bandwidth and to know the effective bandwidth we need to know the rows to be excluded (i.e multicolored rows).

In order to resolve this, explicit calculation of bw_{eff} is avoided if the total bandwidth of the matrix violates requirement 6.3. Instead we divide the matrix into $nthreads$ equal parts column-wise using separators L_1 to $L_{nthreads+1}$ as shown in Fig.6.4.b for $nthreads$ = 4. Then the rows are initially classified as follows:

- Pure zone (P_k)- if the column indices of the corresponding row are between L_k and L_{k+1}.
- Transition zone (T_k) - if the column indices of the corresponding row are between L_k and L_{k+2}.
- Multicolored zone (MC) - if the row does not belong to P_k and T_l; $\forall\ k \in [1, nthreads]$ and $l \in [1, nthreads - 1]$

The zones are then arranged as P_1, T_1, P_2, T_2, . . . , $T_{nthreads-1}$, $P_{nthreads}$, MC. In Fig. 6.4.b to 6.4.c we see the permutation of MC zone to the end and multicoloring being applied. It is to be noted that in the previous procedures all the splitting was done analytically and the only input required was the bandwidth of the matrix, but as we see for this case it is not possible to do it analytically and one would require to scan the matrix, to determine the zones, thus adding an overhead of $\mathcal{O}(nnz)$. During the scanning, we also check to do additional permutations to enable BMC on more rows. For example if a row contains only column indices between L_1 and L_2, it would be row permuted to the pure zone P_1 (first orange region in Fig. 6.4) if found somewhere else.

Now knowing the rows to be multicolored, one could find the bw_{eff} as the bandwidth of sub-matrix excluding the multicolored zone. This then enables us to continue the classification of transition zones based on the effective bandwidth (bw_{eff}) of the matrix as explained in the previous sections. Fig. 6.5 shows this splitting for 4 (no TT zones) and 5 (TT zones visible) threads on Graphene-20-20 with PBC boundary.

Remark on TT zone

If in TT zone only half the available threads could be used, one has to be careful, since if this zone is relatively large it could reduce the available parallelism to a level where it hurts performance. In some situation this would then mean multicoloring this region would be more beneficial than using half the threads. Currently in our implementation this decision is not made automatically, and the user has to do it explicitly. In principle, automation could be done by running some small benchmark prior to matrix permutations.

6.4 Bandwidth Reduction algorithms

Symmetric Matrix

From the previous sections on the BMC method we saw that the effectiveness of the BMC method increases if the bandwidth of the matrix is below certain limits, else it might lead to non-utilization of the available resources or performance drop due to an increasing number of rows being multicolored. Due to this strong dependency it is often advantageous to reduce the bandwidth of the matrix. Bandwidth depends on the numbering of the nodes. A bandwidth reduction algorithm tries to reorder the node numbering such that bandwidth is made as small as possible.

A commonly used bandwidth reduction algorithm for symmetric matrices is Reverse Cuthill-McKee (RCM), a variant of breadth-first search algorithms. The algorithm operates on a graph G of the matrix. The $k-th$ step of the algorithm is as follows [51]:

1. k = 1: Choose a peripheral vertex, i.e a vertex with lowest degree [7], which gets the number 1. This starting node forms the level 1.
2. k > 1: Terminate if all nodes are already numbered, else form the level k with all nodes that are not numbered and neighbors of nodes in level $k-1$. Sort the level in ascending order of vertex degree and number them consecutively.
3. if all nodes are numbered, inverse the numbering i.e.:

$$\text{new node number} = \text{Total nodes} + 1 - \text{old node number}$$

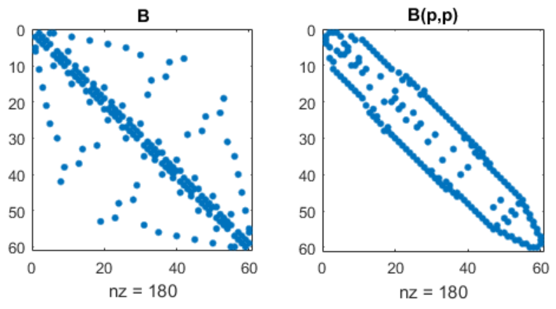


Figure 6.6: A symmetric matrix before and after applying RCM (taken from MATLAB documentation [12])

Bounds for bandwidth after RCM can also be established (see Section 2.5 of [51]). Fig. 6.6 shows a matrix before and after RCM. In our implementations we used RCM to reduce the bandwidth, but there are other algorithms like GPS [50] and JCL [46] which also serve the same purpose.

Non Symmetric Matrix

Previously, we saw methods to reduce the bandwidth of a symmetric matrix. But we have lots of counter-examples to symmetry which arises from the problem itself or the permutations applied to it. For example the permutations we saw in Section 6.3, if matrix does not satisfy Eqn. 6.3 can make it unsymmetric, or a problem with convection terms can be unsymmetric. Thus for an effective KACZ_BMC method we need a bandwidth reduction algorithm for these non-symmetric cases too.

For non-symmetric matrix the previously stated methods like RCM wouldn't work, since the graph of the matrix A is no more undirected. Contrary to the symmetric case the lower bandwidth l and upper bandwidth u differs, but for the KACZ_BMC, aim of such a bandwidth reduction algorithm should be to minimize the total bandwidth $l+u$. A method to do such a reduction is to create an undirected graph of the matrix A, apply the available RCM method to it, and finally relate the derived permutations for undirected graph to the graph of matrix A. A lot of study in this direction has been carried out, a good overview and comparisons between different approaches could be found in [39].

In our current implementation we use bipartite graph of A (referred as $\hat{A}$) to form the undirected graph of

[7]degree of a vertex (node) $\mathbf{a}_i$ is defined as the number of neighbors of $\mathbf{a}_i$

A. The matrix related to $\hat{A}$ (adjacency matrix) is formed from A as:

$$\hat{A} = \begin{bmatrix} 0 & A \\ A^{\intercal} & 0 \end{bmatrix} \tag{6.5}$$

Note that $\hat{A}$ is symmetric thus the graph is undirected. Now a normal RCM or a much simpler BFS (Breadth First Search) method can be carried out on $\hat{A}$. Due to the special arrangement of $\hat{A}$ the levels arising (see step 2 in RCM algorithm) from these algorithms are alternately row and column level sets. After the BFS (or RCM) on $\hat{A}$, the rows of matrix A are permuted according to the numbering in levels comprising of rows (row level set) and columns are permuted with the levels comprising of columns (column level set).

The SpMP [14] library contains routines for bandwidth reduction on non-symmetric matrices. The implementation for non-symmetric matrices works with the standard Breadth First Search (BFS) method instead of RCM, and the library is capable of creating the associated bipartite graph. The major difference between BFS and RCM is that BFS does not have the requirement to sort the nodes in each level, contrary to the step 2 mentioned for RCM in previous subsection.

Fig. 6.7, shows an example of matrix before and after bipartite-BFS bandwidth reduction.

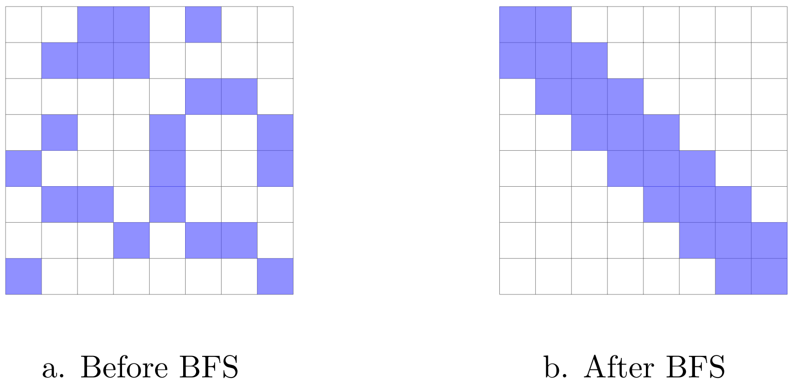


Figure 6.7: Bandwidth reduction of an unsymmetric matrix using BFS algorithm

6.5 SMP and Block Multicoloring

Shared memory parallelisation of KACZ using BMC method is different from the MC method, since in BMC we assign group of blocks to different thread and then the assigned thread is responsible for sweeping through all the rows within the blocks. This means that a simple worksharing construct like *omp parallel for* is not going to work. Implementation details will be dealt in the Section 6.6. Below we revisit the performance issues we faced for MC and analyze it for BMC method.

α effect and False sharing

Compared to MC method we see that there is no scattering of data in BMC method, but in fact we even try to reduce the bandwidth by using bandwidth reducing algorithms mentioned in the previous Section 6.4. As a consequence, the α factor of the matrix is not increased (it might even be decreased due to bandwidth reductions). This eliminates the extra data transfers which we had seen for multicoloring in Section 5.5. It also further reduces the amount of false sharing to just the rows near the boundary between two threads. Fig 6.8 illustrates the maximum possible false sharing for the example shown in Fig. 6.2 during its red sweep with a fictitious cache line size of two elements. The amount of false sharing is similar to the amount encountered in the SPMV algorithm.

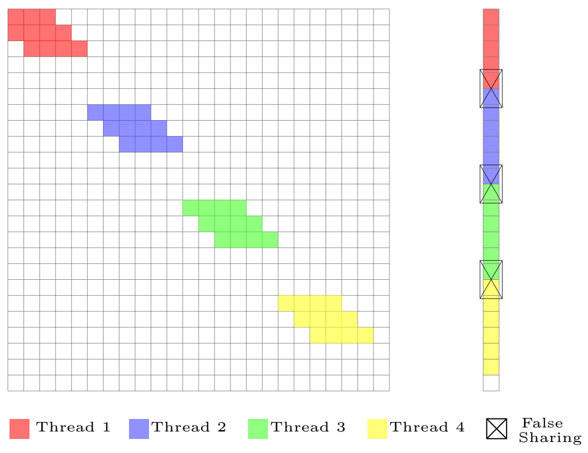


Figure 6.8: Illustration of False sharing in BMC, with Cache line size = 2

Global Barriers

Yet another advantage of the BMC method is the smaller number of global barriers. The maximum barriers required is equal to the number of zones present for the current matrix, since we only need one barrier after each zone.

	Condition	Total Barriers
1	$bw \le \frac{nrows}{2*nthreads}$	2
2	$\frac{nrows}{2*nthreads} < bw \le \frac{nrows}{nthreads}$ & no TT zone	3
3	$\frac{nrows}{2*nthreads} < bw \le \frac{nrows}{nthreads}$	4
4	$bw > \frac{nrows}{nthreads}$	4 + no. of colors in MC zone

Table 6.2: Maximum global barriers required for BMC method

Table 6.2 shows the number of barriers required for BMC method, depending on the condition satisfied by the matrix. The very low amount of barriers for condition 1-3 implies the barrier cost is almost negligible. For the condition 4 it strongly depends on the number of rows multicolored and number of colors required for these rows.

In the case of BMC, one could even replace the global barriers by mutual synchronization with the following thread (or the preceding thread in case of the backward sweep), eliminating global barriers all together.

Data Placement

The optimal data placement strategy is not exactly the same as for SPMV, but generally similar. Only few rows near the boundaries between adjacent NUMA domains might be placed improperly. The placement also reduces greatly remote false sharing compared to the MC method.

Load Balancing

One more thing worth mentioning is that if the matrix bandwidth and number of threads ($nthreads$) used satisfies the requirement mentioned in the conditions 1, 2 or 3 (see Table 6.2) a load balancing scheme could be employed, if we are not at the extreme limit of bw = $\frac{nrows}{2*nthreads}$ for condition 1 or bw = $\frac{nrows}{nthreads}$ for condition 2 and 3. A load balancing scheme for condition 2 or 3 might be tricky since changing loads might lead to emergence (in case of condition 2) or increase (in case of condition 3) of TT zone leading to performance degradation if only half the threads can work on it. Currently, in our implementation, a load balancing based on the number of non-zeros (nnz) is employed if the matrix satisfies condition 1.

6.6 Implementation

Pre-Processing

Since the block multicoloring depends heavily on bandwidth, all the permutations that change the bandwidth has to be carried out first and no further permutations are allowed after BMC. For example in GHOST if the user allows for bandwidth reduction matrix is permuted by RCM algorithm (in case of symmetric matrix) or BFS (in case of unsymmetric matrix) as described in Section 6.4. Bandwidth reduction in GHOST is carried out using an external library called **sparse matrix pre-processing library (SpMP)** [14]. After all the permutations the matrix bandwidth is determined.

If the condition in Eqn. 6.3 is satisfied, no further permutations of the matrix need to be carried out and one could ensure the absence of any multicoloring. All the zones here could be determined analytically by just knowing the bandwidth (bw) of the matrix and further knowledge of the actual sparsity pattern of the matrix is not needed. The BMC pre-processing time in such a case is negligible. The following algorithm is used after the matrix assembling stage to derive the zones:

BMC analytical pre-processing

```
for $k = 0 : nthreads - 1$ do
    $ps_k = k * floor(nrows/nthreads)$
    $pe_k = ps + floor(nrows/nthreads - bw)$
    $P_k \in [ps_k, pe_k]$
end for
for $k = 0 : nthreads - 2$ do
    $re_k = pe_{k+1} - bw$
    $bs_k = ps_k + bw$
    if $re_k > bs_k$ then
        $re_k = bs_k = (re_k + bs_k)/2$
    end if
    $RT_k \in [pe_k, re_k]$
    $BT_k \in [bs_k, ps_{k+1}]$
    $TT_k \in [re_k, bs_k]$
end for
```

In our implementation we further differentiate between condition in Eqn. 6.1, since if this is satisfied we have only 2 zones and we are sure not to have transition zones, thereby enabling some effective load-balancing scheme. The only condition in this case is that none of the blocks after load balancing should have fewer than bw rows.

If the matrix does not satisfy Eqn. 6.3, a pre-processing step has to be carried out where the rows of the matrix would be scanned, and depending on the minimum and maximum column index in that row, the row would be classified to one of the following zones P-pure zone , T-transition zone or MC-multicolored zone as described in Section 6.3. Note that it is not just the rows that belong to MC zone that get permuted as shown in Fig. 6.4; the rows in other zones can also get permuted if they originally are in a different position. MC zone is further multicolored using COLPACK. These kind of permutations would lead to an unsymmetric matrix since here its only the rows that are being permuted. After the initial permutation the division of transition zone (T) is done, by which we divide it into red-trans zone (RT), black-trans zone

(BT) and trans-trans zone (TT) as shown in Section 6.3. This is based on a purely analytical method and does not require any permutations.

A summary of pre-processing steps required for a matrix that does not satisfy the bandwidth requirement is as follows.

1. Read the row pointers and column entries
2. Determine separators L_1 to $L_{nthreads+1}$, where $L_{k+1} - L_k = ncols/nthreads$
3. Determine P_k, T_k and MC as mentioned in sec. 6.3
4. Apply multicoloring to MC zone
5. Store the permutations and inverse permutations of the matrix, in the order P_1, T_1, P_2, T_2, . . . , $T_{nthreads-1}$, $P_{nthreads}$, MC
6. Assemble the matrix using the permutation in a format of choice
7. Split the transition zones into RT, BT and TT zones as mentioned in the Eqn. 6.4.

For convenience the zones are referenced by pointers to their start and end rows, respectively. Therefore we have a $zone\_ptr$ array which stores the row indices in the following order: ps_k, pe_k, re_k, bs_k, ps_{k+1}, pe_{k+1}, re_{k+1},.... This implies the length of the array is $4 * nthreads$.

TT zone

One more thing to determine is whether all threads can simultaneously work on TT zone (trans-trans zone) or only half at a time. We distinguish 2 cases using the following term:

- BMC_one_sweep - refers to the fact that TT zone can be done in one sweep using all threads simultaneously
- BMC_two_sweep - refers to the fact that TT zone can only be done in two sweep using half the threads at a time.

In order to determine under which category the matrix falls one could scan the transition zones and check if some column indices are overlapping (conflicting) with nearest neighbor. If it overlaps in any one point the algorithm could return the mode as BMC_two_sweep. The algorithm is as follows:

Determination of TT-zone mode

```
$mode = BMC\_one\_sweep$
**for** $k = 1 : nthreads - 1$ **do**
  $lower$ = min. column index in the $TT_k$ range, i.e. $zone\_ptr[4*k+2] : zone\_ptr[4*k+3]$
  $upper$ = max. column index in the $TT_{k-1}$ range, i.e $zone\_ptr[4*k-2] : zone\_ptr[4*k-1]$
  **if** $lower \leq upper$ **then**
    $mode = BMC\_two\_sweep$
    $break$
  **end if**
**end for**
```

Remark: Unsymmetric Permutation

Unsymmetric permutations need to be handled with extreme care to avoid wrong results and/or large performance penalties. This arises mainly due to the reason that column space is no more equal to row space, as opposed to symmetric permutations. This implies each matrix and vector arising in the system should be checked for compatibility. Below are some guidelines:

- The (block) vector can either belong to row space or column space. Each kernel implemented should check for the consistency and (un)permute them if required.

- One should also take care, which permutation (row or column) to be applied to a (block) vector . For example on a system $Ax = b$ after KACZ_sweep we need to permute back our vector x (in column space) using column permutation to get back the result in original unpermuted indices but after SPMV we need to apply row permutations to vector b (in row space).
- Due to these 2 types of vector one might also need routines to switch between vector types from row space to column space or vice-versa. For example taking a transpose might require one to switch between spaces.

Since there are many ways to make the vectors compatible, selecting the proper sequence is also important from a performance point of view. Since each permutation or switching operations require copying data to other array and indirect reads/writes, it might affect performance if not kept to a minimum. Even storing or permuting sparse matrices should be avoided as much as possible. In general one should always try to compute in the permuted space as long as possible and permute the results back only at the very end. In Chapter 8 an example for such kind of permutations are presented.

From GHOST v1.1 onwards, the user does not need to worry about these unsymmetric permutations if they solely use GHOST kernels for computations. In the new version each densematrix is attached with a map and its auto-permute facilities kicks in (if enabled) if it finds the vectors are not compatible. Defining a function outside GHOST would then require the user to add compatibility checks explicitly.

KACZ_BMC sweep

After the pre-processing step, the (permuted) matrix along with the derived $zone\_ptr$ and $color\_ptr$ (if rows are multicolored) could be used to implement the KACZ sweep using BMC. Since we create only $nthread$ number of blocks in each zone, we just need to assign each block to a thread using its $thread\_id$, but note that the last thread with $thread\_id$ = $nthread - 1$ will not have any transition zone in case the requirement in Eqn. 6.3 is not satisfied, since transition zones arise only between 2 pure zones. One could either leave it as it is, since the extra load is distributed to all the remaining threads or even one could explicitly assign some rows to the last thread's transition region. However, to find the no. of rows to be assigned as penalty is difficult for a general case. In our current implementation we do not assign any transition regions for the last thread and leave it as $re_{nthread-1}$ = $bs_{nthread-1}$ = $pe_{nthread-1}$. The basic algorithm for KACZ_BMC using OpenMP is in this fashion shown in Algorithm 7 and Algorithm 8:

Algorithm 7 KACZ_BMC algorithm - Forward direction

```
if $dir == FORWARD$ then
    $stride = 1$
    #pragma omp parallel
    {
    $tid = omp\_get\_thread\_num()$
    $start[4] = 0;$
    $end[4] = 0;$
    for $zone = 0 : 3$ do
        $start[zone] = zone\_ptr[4 * tid]$
        $end[zone] = zone\_ptr[4 * tid + 1]$
    end for
    if $method == BMC\_one\_sweep$ then
        for $zone = 0 : 3$ do
            $KACZ\_SWEEP(start[zone], end[zone], stride)$
            #pragma omp barrier
        end for
    else
        for $zone = 0 : 1$ do
            $KACZ\_SWEEP(start[zone], end[zone], stride)$
            #pragma omp barrier
        end for
        $\{TT - ZONE\}$
        if tid%2 == 0 then
            $KACZ\_SWEEP(start[2], end[2], stride)$
            #pragma omp barrier
        end if
        if tid%2 != 0 then
            $KACZ\_SWEEP(start[2], end[2], stride)$
            #pragma omp barrier
        end if
        $KACZ\_SWEEP(start[3], end[3], stride)$
    end if
    }
    for color=0:ncolors-1 do
        #pragma omp parallel for
        $KACZ\_SWEEP(color\_ptr[color], color\_ptr[color + 1], stride)$
    end for
end if
```

Algorithm 8 KACZ_BMC algorithm - Bckward direction

```
if $dir == BACKWARD$ then
    $stride = -1$
    for color=ncolors:1 do
        #pragma omp parallel for
        $KACZ\_SWEEP(color\_ptr[color] - 1, color\_ptr[color - 1] - 1, stride)$
    end for
    #pragma omp parallel
    {
    $tid = omp\_get\_thread\_num()$
    $start[4] = 0;$
    $end[4] = 0;$
    for $zone = 3 : 0$ do
        $start[zone] = zone\_ptr[4 * tid + 1] - 1$
        $end[zone] = zone\_ptr[4 * tid] - 1$
    end for
    if $method == BMC\_one\_sweep$ then
        for $zone = 0 : 3$ do
            $KACZ\_SWEEP(start[zone], end[zone], stride)$
            #pragma omp barrier
        end for
    else
        $KACZ\_SWEEP(start[0], end[0], stride)$
        #pragma omp barrier
        $\{TT - ZONE\}$
        if tid%2 != 0 then
            $KACZ\_SWEEP(start[1], end[1], stride)$
            #pragma omp barrier
        end if
        if tid%2 == 0 then
            $KACZ\_SWEEP(start[1], end[1], stride)$
            #pragma omp barrier
        end if
        for $zone = 0 : 1$ do
            $KACZ\_SWEEP(start[zone], end[zone], stride)$
            #pragma omp barrier
        end for
    end if
    }
end if
```

Flowchart

The entire procedure for BMC method is summarized in the flowchart shown in Fig.6.9 .

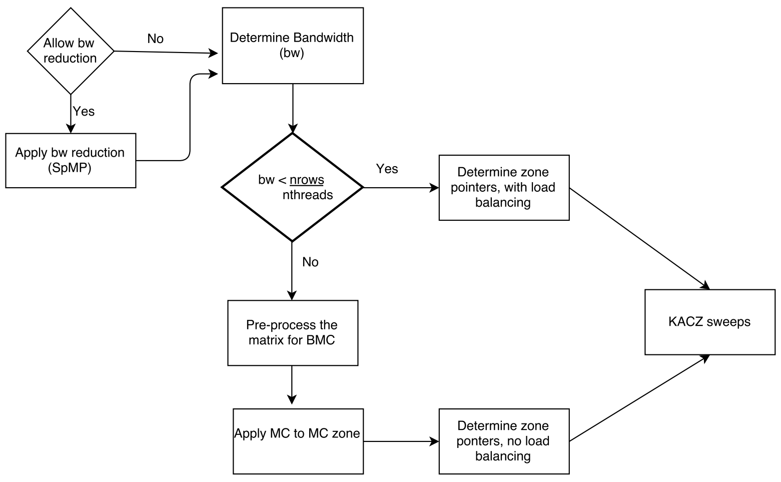

Allow bw reduction
No
Yes
Determine Bandwidth (bw)
Apply bw reduction (SpMP)
bw < nrows / nthreads
Yes
Determine zone pointers, with load balancing
No
KACZ sweeps
Pre-process the matrix for BMC
Apply MC to MC zone
Determine zone ponters, no load balancing

Figure 6.9: Flowchart KACZ BMC

6.7 Results and Discussions

Node-level performance

In order to see the performance of the KACZ_BMC on a single node we conduct scaling test as done in the previous chapter for multicoloring.The test scenario is similar to the one mentioned in Section 5.5. Fig.6.10 shows the test results for PWTK and Graphene-4096-4096 matrix (see Appendix 10.1.3 and 10.1.2 for matrix details).

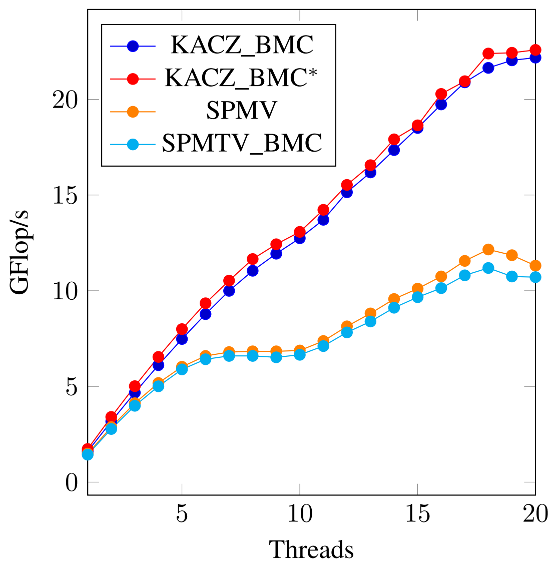
For the PWTK matrix with an initial total bandwidth of 378662, a RCM permutation is conducted which reduces the bandwidth to 4062. This makes it possible to use all the 20 cores (without hyper threading) available on a single node of Emmy. Moreover, none of the rows need to be multicoloring and all the threads are able to work simultaneously in the TT-zone (i.e BMC_one_sweep method). Similar is the case with Graphene matrix.

We see that performance of KACZ_BMC on PWTK matrix is same as that of SPMV at one thread, then it increases linearly until it hits the memory bandwidth saturation point at approximately ten threads on a single socket, where the likwid-perfctr [63] measures a memory bandwidth of 37 GB/s, close to the full socket memory bandwidth of $\approx$ 40 GB/s on Emmy. One could further observe that when the kernel operates at the saturation point the performance of KACZ_BMC is double that of SPMV, satisfying our basic prediction introduced in Section 5.5.

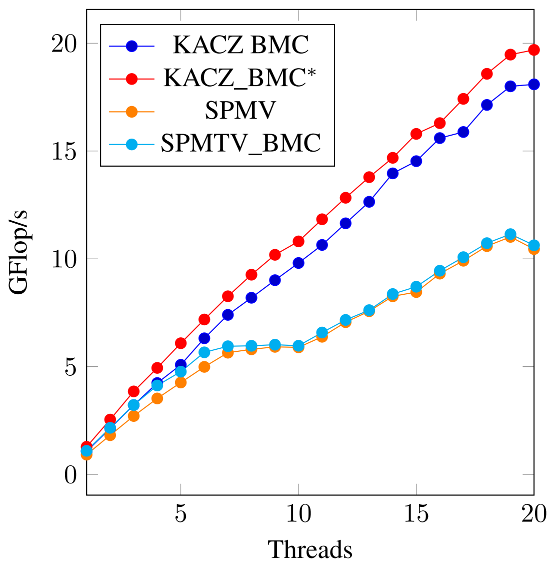
The Graphene matrix also follows the same pattern, but here it could not saturate the main memory bandwidth at ten threads. However one can note that at ten threads the performance is close to double that of SPMV (SPMV: 5.84 GFlop/s and KACZ_BMC: 9.65 GFlop/s, KACZ_BMC*[8]: 10.81 GFlop/s) indicating it is close to saturation. An explanation for the delayed saturation in Graphene matrix compared to PWTK matrix could be the small number of non-zeros per row ($\approx$ 13), inhibiting any vectorization along the row and effective modulo variable expansion for the loop-carried dependency on the summation variable in the first loop.

The reason for the same performance at one thread and the delay in saturation of KACZ_BMC compared to SPMV is because of the two loops in KACZ_BMC kernel, compared to one in SPMV. Even though all

[8]*- represents KACZ_SWEEP was done on row-nomalized system



(a) PWTK



(b) Graphene-4096-4096

Figure 6.10: Scaling test of KACZ_BMC method on 1 socket of Emmy, Performance of SPMV and SPMTV_BMC is plotted on its side for performance comparison. *: represents KACZ carried out on normalized system.

the data required for the second loop comes from the L1 cache (since nnz per row = 53 for PWTK and 13 for Graphene), but the data has to be loaded form L1->Reg, which costs cycles. Since each core has its own individual L1 cache, we see a linear scaling due to this bottleneck until it hits the pure main memory bottleneck. Furthermore, the rownorm computation costs some cycles which is observed as the difference between KACZ on pre-normalized system (red line) and on the original system (blue line) in Fig. 6.10. A thorough investigation of this is conducted in the Section 7.1, where the results are compared with a performance model.

The scaling across sockets seems to be quite good, since we observe a factor of almost 2 in the performance from thread 11-20 compared to threads 1-10. This is due to the availability of a second NUMA domain. The perfect scaling across sockets also proves the correct data placement across the socket.

Section 6.8 provides a detailed comparison of the KACZ_BMC method and KACZ_MC method in terms of performance.

Variants

In the previous result we couldn't observe the performance results of all the KACZ_BMC variants since the ratio of rows to bandwidth was sufficiently small to enable the most optimal variant. In order to observe performance of all the variants we choose a small matrix from the ESSEX project called Topi-36-36-36 (see Appendix 10.1.6). Fig. 6.11 shows this result.

The matrix has 186624 rows and a total bandwidth of 10414 with RCM. The red dashed line (a-d) in the Fig. 6.11, are the significant breakpoints at which modes change

- region $[o, a)$ - In this region we have only 2 zones (or block colors) as explained in Section 6.1. Load balancing between threads is possible in this region.
- region $[a, b)$ - Line a signifies one thread after the limit of equation 6.1 i.e $\frac{bw}{2*nrows} = \frac{186624}{2*10414} \approx 8.9$. In this region there are 3 zones, namely pure (P), red-trans (RT) and black-trans (BT)
- region $[b, c)$ - At b the TT zones (4^{th} zone) appear, but all the available threads can be used in TT zones.

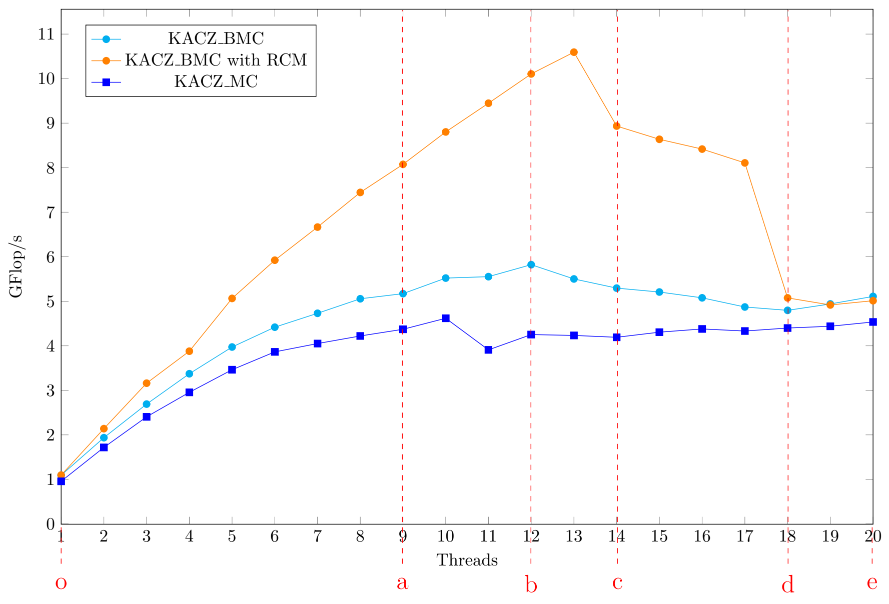


Figure 6.11: Performance variation of KACZ_BMC as it passes through various regions is observed for Topi-36-36-36 matrix.

- region [c, d) - From c only half the available threads can be used in TT zone, a performance drop can be observed since we under-utilize the available resources.
- region [d,e] - Line d signifies one thread after the extreme limit of the equation 6.3, i.e,$\frac{bw}{nrows} = \frac{186624}{10414} \approx 17.9$. In this region some rows are multicolored, and all the TT- zones can only be processed with half the available threads. Performance degradation from the previous region [c,d) is to be expected since we have multicoloring of some rows.

By combining both of the methods we see that, we could harness maximum performance using KACZ_BMC whenever possible and if the requirements for KACZ_BMC starts to violate the performance begins to drop until we reach the limit of the normal KACZ_MC method (blue line in Fig. 6.11).

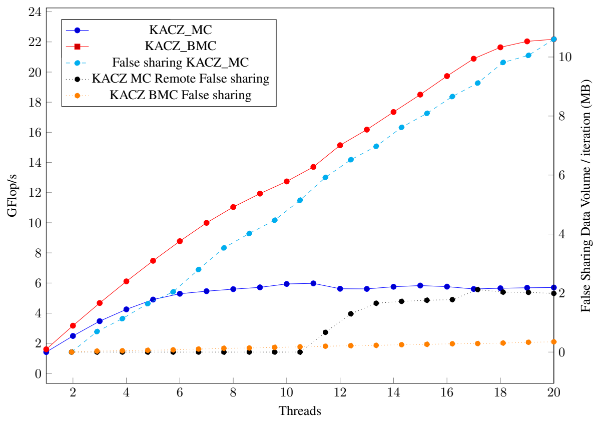
The cyan colored line in Fig.6.11 shows the performance of KACZ_BMC without doing any bandwidth reduction (RCM), here the total bandwidth is high such that from 3 threads onwards multicoloring kicks in destroying the performance.

6.8 Multicoloring vs Block multicoloring

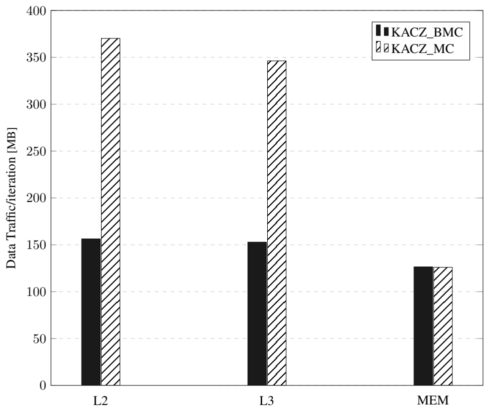
Currently we saw two different kinds of implementations for the Kaczmarz kernel. In this section we briefly compare both of them, from two major aspects of a linear solver: performance and convergence behavior.

Performance Results

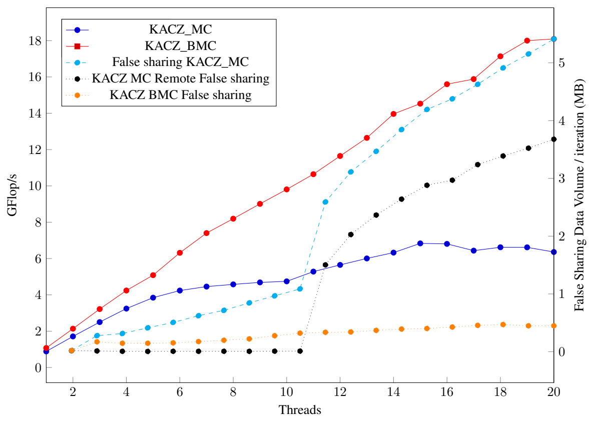
Fig. 6.12.a and 6.12.c, shows the performance comparison between the two methods on the PWTK and Graphene-4096-4096 matrix where we clearly see the superiority of the KACZ_BMC method. Note that here we have used unsymmetrically permuted multicoloring for comparison, since in Chapter 5 we saw that this performs better than the symmetrically permuted counterpart. On the right axis we measure the false sharing data volume in MB using likwid-perfctr. We see strongly increasing amount of false sharing for the KACZ_MC, whereas for KACZ_BMC it is almost negligible. One more thing to note is the amount of remote false-sharing occurring for KACZ_MC kernel. The black dotted line illustrates this, we see on 2



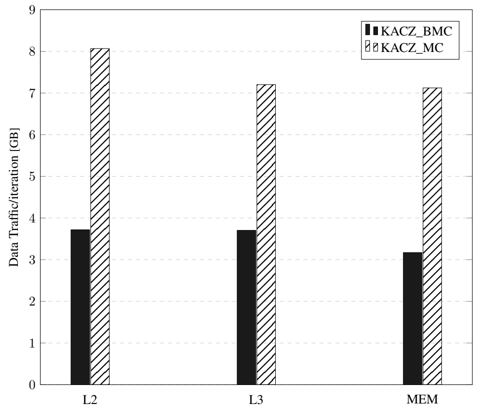
(a) PWTK: Performance and false sharing



(b) PWTK: Data Traffic



(c) Graphene: Performance and false sharing



(d) Graphene: Data Traffic

Figure 6.12: Performance comparison between KACZ_BMC and KACZ_MC methods. Fig. 6.12a and 6.12c show the Performance comparison and the change in amount of False sharing. Fig. 6.12b and 6.12d show the comparison between data traffic measured by LIKWID.

sockets 20% of false sharing in the case of PWTK matrix and 80% of false sharing in the case of Graphene matrix is due to remote false sharing, which costs lot more cycle than the local false sharing.

Fig. 6.12.b, shows the overall data traffic from various memory hierarchies for PWTK matrix. We clearly see for KACZ_BMC the amount of data required from L2 and L3 is almost the same as that from main memory (MEM), indicating an almost pure "streaming" access pattern, whereas for KACZ_MC L2 and L3 data increases due to the reasons mentioned in Section 5.2. For Graphene matrix one could also observe additional traffic from main memory (see Fig. 6.12.d) in case of KACZ_MC, indicating a greater influence of α factor.

Convergence behaviour

The study of convergence behavior is especially important, since by permuting the matrix or by changing the order in which we project to the row we change the convergence behavior. The fact can be compared to the normal Gauss-Seidel and Red-Black Gauss-Seidel. In our case KACZ_MC, KACZ_BMC and serial KACZ, all have different order of projection to the rows. It has been shown that, independent of the order, the KACZ method should converge to the solution [52]. But the rate at which the method converges might differ. Here we study this behavior for the three versions of KACZ on two matrices, PWTK (see Sec.10.1.3) and Spin-24-24-24 (see Sec.10.1.4) matrix. To conduct the test the system is set up such that the solution to the system is a vector with all its elements set to 10, and the initial guess for the system is prescribed as the 0 vector. The absolute error in the L2 norm is then computed to measure the quality of convergence as the number of iterations increase.

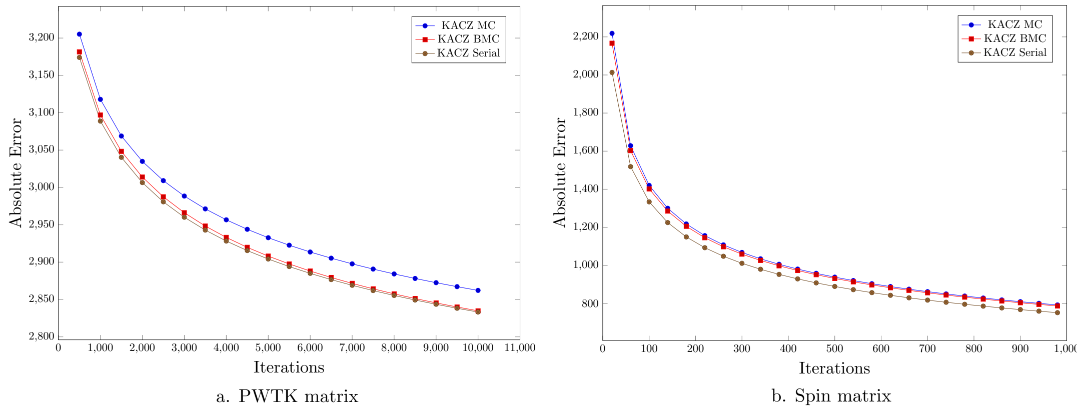


Figure 6.13: Convergence comparison between different versions of KACZ: serial, multicolored, and block multicolored for PWTK and Spin matrices. Absolute error in L2 norm vs number of iterations are plotted for a known solution.

In the first test the number of KACZ iterations needed to reach a particular error is plotted. Figure 6.13.a and 6.13.b shows convergence of the methods in comparison to the serial version of KACZ. In general one could observe the smooth convergence property of geometrical row-projection method. Further we could observe that for both of the matrices multicoloring (KACZ_MC) performs, worser than the serial implementations, while the Block multicoloring (KACZ_BMC) technique lies in between. The reason for this might be attributed to the strong relation with neighbors within a block in BMC method as opposed to disjoint neighbors in the MC method.

The second test (Figure 6.14.a and 6.14.b) compares the time taken by CGMN algorithm to reach a particular error norm. It has to be noted that the time axis is in log-scale to enhance the readability. Fig.6.14.a elucidates the lack of scalability of KACZ_MC method for the PWTK matrix where the time is not affected with increasing number of threads, whereas KACZ_BMC scales well. Spin matrices Fig.6.14.b, on the other hand are matrix with large bandwidth and hence KACZ_BMC can only use half the available threads in transition sweep for 15 threads, which affects KACZ_BMC's scalability from 10-15 threads. But the superiority of KACZ_BMC method compared to KACZ_MC method is well evident.

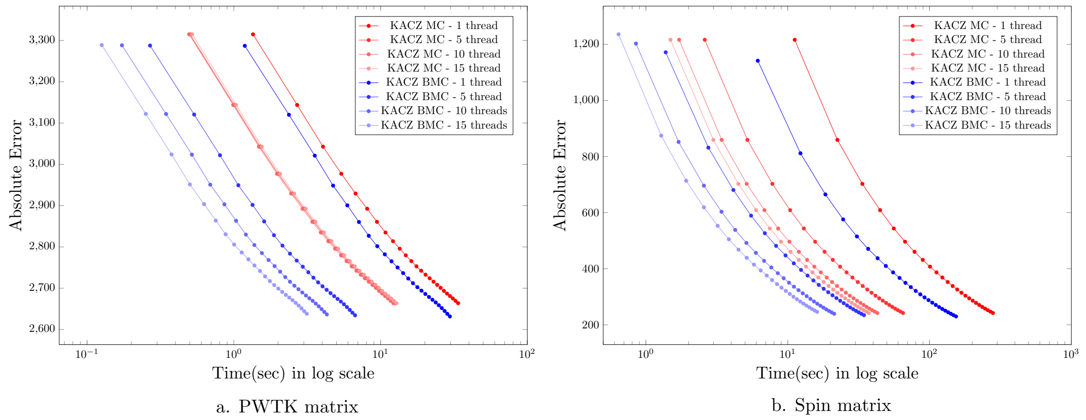


Figure 6.14: CGMN Time To Solution: Time taken to reach a particular error level(L2 norm) is plotted for CGMN with multicoloring(red) and block multicoloring(blue) . Results also show the dependence on number of threads for both cases. Note that time axis(x-axis) is in log scale.

Advantages and Disadvantages

	KACZ_BMC	KACZ_MC
α effect	α comparable to SPMV since data locality is maintained	Scattering effect of multicoloring causes, α to increase drastically
False sharing	False sharing occurs only at thread boundaries, comparable to SPMV	Abundant false sharing, which increases with chromatic number
Global Barriers	Very few global barriers, barrier cost almost negligible. Locks can avoid global barriers.	Requires many global barriers which increases with chromatic number. Barrier cost can be significant
Intra-Socket scaling	Good scaling within a socket, reaches twice the performance of SPMV if memory bandwidth can be saturated	Poor scaling within a socket, does not saturate memory bandwidth
Inter-Socket scaling	Good inter-socket scaling, since data placement almost similar to the normal SPMV style placement	Inter-socket scaling limited by poor data placement and wide-occurrence of remote false sharing
Restrictions	A pure KACZ_BMC without multicoloring has restrictions seen in Eqn. 6.1 and 6.3	No restrictions
Fine-grained parallelism	Fine-grained parallelism beyond thread level difficult. Cannot vectorise easily, therefore no benefit from SELL-C-σ data format.	Easy vectorisation possible, SELL-C-σ data format can be used effectively
GPU	Because of the above limitation GPU-implementation might be difficult	Can be easily implemented for GPU

Table 6.3: Summary of the advantages and disadvantages between KACZ_BMC and KACZ_MC

Table 6.3 shows a comparison of the KACZ_MC and KACZ_BMC methods, based on the various aspects which we have looked into.

Note that more fine-grained parallelism is possible by permuting the matrix after KACZ_BMC pre-processing stage. For example if we want to AVX-vectorise the code using SELL-C-σ, and suppose we could find 4*nthreads blocks in each zone, we could permute each 4 blocks chunk (say a,b,c,d) to a single block that contains the rows in following order a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2,.... , where the subscript refers to the row index. Currently this has not be done and is a field to be investigated in future.

Further more with Intel's new Conflict Detection Instructions [1] for 512-wide vector units (available as AVX512CD instructions) the vectorization issues can be resolved. These instructions generate a mask with a subset of elements that are guaranteed to be conflict free.

But it has to be kept in mind that all the algorithm and matrices need not benefit from vectorization, especially if they are memory bound. In general most of the sparse matrix applications are capable to saturate memory bandwidth on full-socket.

6.9 Validation

In this section we perform some performance validation of our approach using block multicoloring for different matrices. The validations are carried out on 1 Socket of two different Intel architectures: Ivy Bridge and Broadwell. Choice of 1 socket is to avoid disadvantage of the multicoloring method due to incorrect data placement. The architecture details are as follows:

System name	Emmy	broadep2
Architecture	Ivy Bridge	Broadwell
Clock	2.2 GHz	2.3 GHz
Sockets	2	2
Cores	10×2	18×2
L1 Cache	32 KB	32 KB
L2 Cache	256 KB	256 KB
L3 Cache(shared)	25 MB	45 MB
Main Memory	32 GB $\times$ 2	64 GB $\times$ 2
Bandwidth per Socket	41 GB/s	62.7 GB/s
Compiler	Intel icc v 15.0.5	Intel icc v 16.0.3
Compiler flags	-fno-alias -xHost -O3	-fno-alias -xHost -O3

Table 6.4: Architecture used for validation

Matrices chosen for validation come from various applications and reflect broader classes of problems.

- Graphene - Reflects 2D Stencil case. Applicability of this matrix in FEAST algorithm (refer to Chapter 8) by using CARP-CG has been proven in [32].
- BENCH3D - 3D Stencil derived from the Laplace operator. CARP-CG's effectiveness for these kind of matrices have been proven in [27].
- SpinChainSZ matrices - Derived from the Hamiltonian operator. Since these matrices are not related to a mesh, they have harder sparsity pattern.
- PWTK - Stiffness matrix of a wind tunnel.
- TAU - Derived from a aerodynamic problem. 5 degree of freedom (DOF) per cell.

The first three matrices come from the ESSEX project, PWTK comes from structural problem and TAU from computational fluid dynamics. More on these matrices can be found in Appendix 10.1.

Tables 6.5 and 6.6 show the validation result on the 2 architectures. These tables give us some useful insights on the BMC method as stated in the following.

Matrix Size: Comparison between different sizes of Graphene and BENCH3D matrices reveals increasing effectiveness of KACZ_BMC method as the size increases, this is due to the fact that, as the size increases α factor of the multicolored matrix grows and eventually affects the Main memory reducing KACZ_MC performance. Since ESSEX project deals with large matrices, this factor is quite important.

Cache Size : Another related observation is the connection between cache size and effectiveness (ratio of KACZ_BMC performance to KACZ_MC performance). This could be seen clearly for BENCH3D-A0-128, where KACZ_MC scales not too badly for Broadwell architecture, compared to Ivy-Bridge. The reason for this is because of higher capacity cache in Broadwell architecture, enabling the x-vector in KACZ_MC still to remain in cache, incurring a smaller effect from α factor. This was not the case for Ivy-Bridge.

TT-zone, BMC_two_sweep mode: Spin chain matrix has an unfavorable sparsity pattern and the bandwidth remains pretty big compared to matrices like Graphene and BENCH3D. KACZ_BMC method scales quite well until 15 threads (see Table 6.6), after which it is affected by the TT zones, where only half the available threads (BMC_two_sweep mode) could be used. Nevertheless in comparison with MC method,

Matrix	Threads	MC (%)	Half Thread (%)	BMC Perf. (GFLOP/s)	MC Perf. (GFLOP/s)	Ratio
Graphene-2048-2048	1	0.00	0.00	1.085	0.880	1.232
	5	0.00	0.00	5.315	3.646	1.458
	10	0.00	0.00	10.152	4.636	2.190
Graphene-4096-4096	1	0.00	0.00	1.061	0.918	1.156
	5	0.00	0.00	5.199	3.945	1.318
	10	0.00	0.00	9.641	4.821	2.000
Spin-24-24-24	1	0.00	0.00	1.009	0.526	1.916
	5	0.00	0.00	4.848	2.365	2.050
	10	0.00	0.00	8.970	3.827	2.344
BENCH3D-A0-128	1	0.00	0.00	0.891	0.522	1.708
	5	0.00	0.00	4.314	2.424	1.780
	10	0.00	0.00	8.186	4.338	1.887
BENCH3D-A0-256	1	0.00	0.00	0.878	0.431	2.039
	5	0.00	0.00	4.186	1.928	2.171
	10	0.00	0.00	7.628	3.061	2.492
PWTK	1	0.00	0.00	1.610	1.418	1.135
	5	0.00	0.00	7.499	4.919	1.525
	10	0.00	0.00	12.809	5.971	2.145
TAU	1	0.00	0.00	1.496	1.217	1.230
	5	48.95	0.00	4.841	4.764	1.016
	10	58.30	0.00	7.396	7.445	0.994

Table 6.5: Performance validation on Ivy Bridge(Emmy). **MC(%)** - Percentage of rows multicolored, **Half Thread(%)** - Percentage of rows on which only half the threads can be used (TT zone if in BMC_two_sweep mode), **BMC Perf.**- Performance of KACZ_BMC kernel in GFlop/s, **MC Perf.** - Performance of KACZ_MC kernel in GFlop/s, **Ratio** - Ratio of BMC Perf. to MC Perf.

BMC method is still a better choice. It should be noted that in BMC_two_sweep mode as TT-zone grows, the performance would drop as mentioned earlier and finally reach half the performance that could be achieved with the available threads.

MC-zone : TAU matrix is one of the hardest matrices to apply the BMC method to since it has a very high bandwidth and even reducing with RCM doesn't enable us to use 3 or more threads satisfying the requirement in Eqn. 6.3, making it a worst case scenario. The TT zone has 0% since it was switched off to avoid the disadvantage due to under utilization of threads as mentioned towards the end of Section 6.3. Due to multicoloring of its rows, KACZ_BMC couldn't prove to be effective and performance converges to that of KACZ_MC in this case.

Sparsity Pattern : Sparsity pattern of the matrix is quite important. In general if matrices of similar sizes are compared, performance of KACZ_MC worsens as the pattern goes from stencil-like type to more general pattern due to increase in irregular access and false sharing, as mentioned in Chapter 5. This could be observed for example by comparing stencil like matrices Graphene-2048-2048, BENCH3D-A0-128 with Spin-24-24-24 and PWTK. The increased effectiveness of KACZ_BMC for second class of matrices (except Spin-24-24-24 18 threads on Broadwell, where a different effect kicks in) proves this. On the other hand if the matrix bandwidth does not enable effective application of the KACZ_BMC method, the performance would drop and finally reach the KACZ_MC performance as seen for the TAU matrix.

Matrix	Threads	MC (%)	Half Thread (%)	BMC Perf. (GFLOP/s)	MC Perf. (GFLOP/s)	Ratio
Graphene-2048-2048	1	0.00	0.00	1.514	1.171	1.293
	5	0.00	0.00	7.360	4.926	1.494
	9	0.00	0.00	12.549	6.445	1.947
	14	0.00	0.00	16.784	7.497	2.239
	18	0.00	0.00	17.933	7.885	2.274
Graphene-4096-4096	1	0.00	0.00	1.475	1.193	1.236
	5	0.00	0.00	7.289	4.042	1.803
	9	0.00	0.00	12.499	5.193	2.407
	14	0.00	0.00	16.398	5.946	2.758
	18	0.00	0.00	17.846	6.192	2.882
Spin-24-24-24	1	0.00	0.00	1.364	0.870	1.567
	5	0.00	0.00	6.543	3.502	1.869
	9	0.00	0.00	11.063	4.872	2.271
	14	0.00	0.00	13.709	6.051	2.266
	18	0.00	47.92	11.322	7.600	1.490
BENCH3D-A0-128	1	0.00	0.00	1.032	0.924	1.117
	5	0.00	0.00	5.039	4.270	1.180
	9	0.00	0.00	8.798	7.139	1.232
	14	0.00	0.00	12.998	10.312	1.260
	18	0.00	0.00	14.845	12.504	1.187
BENCH3D-A0-256	1	0.00	0.00	1.026	0.567	1.810
	5	0.00	0.00	5.109	2.271	2.249
	9	0.00	0.00	8.982	3.573	2.514
	14	0.00	0.00	12.917	4.247	3.041
	18	0.00	0.00	12.769	4.464	2.860
PWTK	1	0.00	0.00	1.903	1.630	1.167
	5	0.00	0.00	8.726	5.219	1.672
	9	0.00	0.00	14.146	6.064	2.333
	14	0.00	0.00	19.333	6.852	2.821
	18	0.00	0.00	21.414	6.966	3.074
TAU	1	0.00	0.00	1.620	1.290	1.256
	5	48.95	0.00	5.059	4.847	1.044
	9	60.32	0.00	7.060	7.054	1.001
	14	62.85	0.00	9.246	9.314	0.993
	18	68.26	0.00	10.355	10.157	1.020

Table 6.6: Performance validation on Broadwell. **MC(%)** - Percentage of rows multicolored, **Half Thread(%)** - Percentage of rows on which only half the threads can be used (TT zone if in BMC_two_sweep mode), **BMC Perf.**- Performance of KACZ_BMC kernel in GFlop/s, **MC Perf.** - Performance of KACZ_MC kernel in GFlop/s, **Ratio** - Ratio of BMC Perf. to MC Perf.

7 Model Driven Optimisation

In the previous section we have seen the performance gain by using KACZ_BMC, but still there exists questions like: Is my code optimal? Do I have more room for optimisation? What is the bottleneck of the code? Can I avoid this bottleneck? The first section of this chapter is dedicated to answer such questions about performance, we do this by creating a performance model. The second and third chapter further discuss some of the peculiarities present in the project and discusses about their performance issues and optimisation strategies employed.

7.1 Performance Model

A commonly used performance model is the Roofline model, whose origin could be found in the early 1980's ([24], [38]) and later developed by S.Williams in 2009 [66]. The Roofline model works on the basic assumption that the code is bound by either memory bandwidth or by the applicable peak performance. The Performance(P_{Roofline}) is then given by the equation:

$$P_{\text{Roofline}} = min(P_{\text{max}}, I * b_{\text{s}}) \tag{7.1}$$

where, $I-$Computational intensity of the code

$b_{\text{s}}-$Applicable peak saturation bandwidth

$P_{\text{max}}-$Maximum performance of the code assuming data comes from L1 cache

For sparse matrix algorithms like SPMV, SPMTV and KACZ (see algorithm 6) a naive application of roofline model generally gives a highly pessimistic prediction for most of the matrices, since as we have seen that the data from the x vector in case of KACZ or SPMV and b vector [1] in case of SPMTV, is commonly reused from the caches. We saw in the previous two chapters that the performance of the kernel strongly depends on this effect, which we quantified using the term α (see eqn. 5.2).

[1]x and b refers to vectors in the equation $A * x = b$ or $A^{\intercal} * x = b$

In order to properly construct a model for KACZ or SPM(T)V we need the α factor of the matrix, which typically is not easy to be determined analytically for a general sparse matrix. But by using our common knowledge that SPM(T)V are memory bound on modern architectures, one could determine the factor α as shown below.

- Determine code intensity for SPM(T)V:

$$I_{\text{SPMV}} = \frac{2}{8+4+8*\alpha+\frac{16}{nnzr}}\left[\frac{\text{Flops}}{\text{Bytes}}\right], \qquad I_{\text{SPMTV}} = \frac{2}{8+4+16*\alpha+\frac{8}{nnzr}}\left[\frac{\text{Flops}}{\text{Bytes}}\right] \tag{7.2}$$

- Measure overall data traffic (D) from main Memory, (eg. using likwid [63])
- Now plug in I and D to the equation 7.1 assuming the code is memory bound, and solve for α

$$I_{\text{SPM(T)V}} = \frac{P}{b_s}\left[\frac{\text{Flops/s}}{\text{bytes/s}}\right] = \frac{2*nnz}{D}\left[\frac{\text{Flops}}{\text{Bytes}}\right]$$

Secondly one should be careful in choosing between SPMV and SPMTV as the baseline, because they both have different α factors. In SPMV we have only reads of x (vector accounted for α), but SPMTV has both read and write of b (vector accounted for α). Thus SPMTV additionally considers how the data is being evicted within the hierarchies. Since KACZ also has reads and writes of x (vector accounted for α), SPMTV would be a natural choice to use as the baseline. Another advantage of choosing SPMTV as base line is that one could avoid the explicit calculation of the α factor as shown above and instead directly use the performance of SPMTV, since *the data required from memory per non-zero* is the same for KACZ and SPMTV. Due to this advantage from here on we will never explicitly calculate the α factor, but use the performance of SPMTV as a baseline.

Model Inputs

After determining the **performance of SPMTV**, one could use it as an input parameter for modeling KACZ algorithm. We have seen that with a naive Roofline model for KACZ algorithm one should expect exactly double the performance of SPMTV algorithm since we have 2 extra Flops/non-zero (total 4 Flops/non-zero) without any extra data transfer from the main memory, i.e $I_{\text{KACZ}} = 2 * I_{\text{SPMTV}}$. But as seen in Section 6.7 this is not the case for lower thread counts due to the extra data transfers between caches and registers. This knowledge of **extra data transfer within lower memory hierarchies** also has to be an input of the model. Clearly we also need some **knowledge about the machine** as further input.

Model

The roofline model does not account for the inner cache data transfers, so we have to use an advanced model known as Execution-Cache-Memory(ECM) [58] to account for this. But the problem with the pure ECM model is that we need the data transfers between the caches for the entire algorithm. Since we cannot establish any analytical relationships to calculate data transfers within inner cache hierarchies for a general sparse matrix, as opposed to stencil like algorithms, we would have to measure it using tools like LIKWID [63]. But this would mean too much of measured inputs for the model. In order to avoid this we combine the obtained performance of SPMTV and the ECM model to model the KACZ algorithm. The performance of SPMTV accounts for the data transfers incurred in the second loop, then we add the necessary penalties caused by the extra loads in first loop using the ideas of the ECM model. Note that here we first model the KACZ algorithm on a pre-normalized system (i.e. no $rownorm$ computation), and later on we see the effect of introducing the $rownorm$ computation. In Algorithm 9 we see the splitting of the 2 contributions: SPMTV takes care of the loop in blue color and ECM model for the loop in red.

Algorithm 9 KACZ: solve for x: $Ax = b$, on a pre-normalized system

```
for $row = 0 : nrows$ do
    $scale = b[row]$
    $norm = 0$
    for $idx = rowptr[row] : rowptr[row + 1]$ do
        $scale -= val[idx] * x[col[idx]]$
    end for
    for $idx = rowptr[row] : rowptr[row + 1]$ do
        $x[col[idx]] += scale * val[idx]$
    end for
end for
```

The procedure is as follows:

1. Determine SPMTV performance
2. Convert the obtained performance(P_{SPMTV}) in Flops/sec to cycles/Flop, i.e.

$$C_{\text{SPMTV}}\left[\frac{\text{cycles}}{\text{Flop}}\right] = \frac{f}{P_{\text{SPMTV}}}\left[\frac{\text{cycles/sec}}{\text{Flops/sec}}\right] \tag{7.3}$$

where, f is the clock frequency

3. Now calculate the penalty (T_{nOL} + T_{data}) due to the second loop from different cache hierarchies in cycles/Flop and add it to $\frac{C_{\text{SPMTV}}}{2}$, to obtain C and hence the performance P of KACZ (find detailed explanation below). The factor $\frac{1}{2}$ stems from the fact that $I_{\text{KACZ}} = 2 * I_{\text{SPMTV}}$.

$$C = \frac{C_{\text{SPMTV}}}{2} + T_{\text{nOL}} + T_{\text{data}} \tag{7.4}$$
$$P = f/C$$

Determining the second loop's penalty

To determine the second loop"s penalty (step 3), we need to know from which cache level the data has to be loaded (denoted as Q). This depends solely on the number of non-zeros per row ($nnzr$) and cache line size, since the same data was loaded before $nnzr$ inner iterations by the first loop. But $nnzr$ varies from row to row, so we consider the average $nnzr$ for determining this (assumption: variance is not too high). The fact that always a cache line is loaded causes some difficulty, if the condition determined by $nnzr$ is close to the boundary between 2 cache levels. This difficulty could be avoided if we know the general pattern of our matrix and input it to the model using a cache line safety factor (S_{CL}). S_{CL} is the inverse ratio of the usage of a cache line, i.e.,

$$S_{\text{CL}} = \frac{\text{Average number of cache lines loaded per row}}{nnzr \text{ / cache line size}}$$

For example, if only half of the elements in a cache line are used S_{CL} = 2. In order to avoid under-prediction of the performance, one could also take S_{CL}=1. The equation given below helps us to determine the cache level Q, from which the penalty has to be accounted:

$$Total\ size = nnzr * (8 + 4 + 8) * S_{\text{CL}}\,[bytes]$$
$$\implies nnzr * 20 * S_{\text{CL}}\,[bytes] < S_{\text{C}} * size(Q)$$

The factor 20 accounts for *data loaded per non zero* in the second loop of KACZ (see algorithm 6):

- 8 bytes - x vector

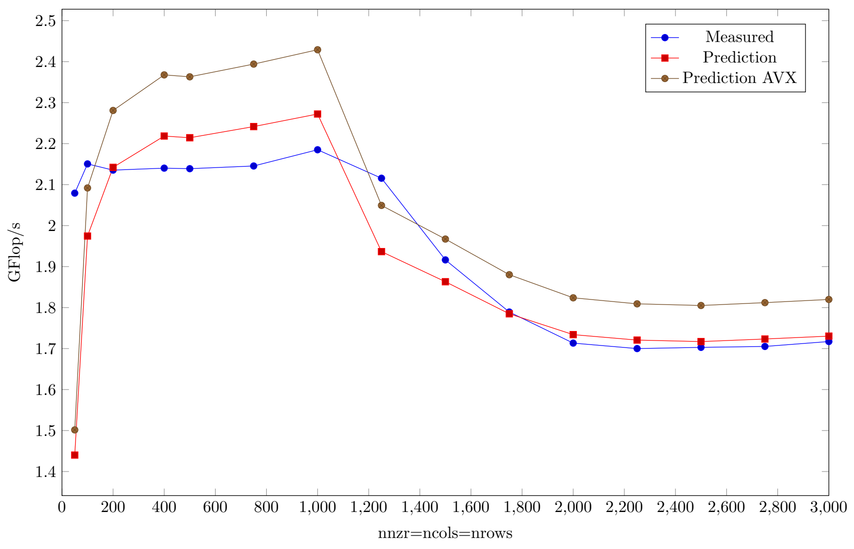


Figure 7.1: Model verification using dense matrices: Performance on a single thread of Emmy for varying size of dense matrix is observed here.

- 4 bytes - col vector
- 8 bytes - val vector

The safety factor S_C accounts for the common observation that the cache starts to spills before its physical size is reached (S_C = 3/4 wouldn't normally hurt). Now we can compute the penalty by adding the extra cycles caused by data transfers from Q to the register. In ECM modeling this is decomposed to 2 parts:

1. T_{nOL} - cycles taken for L1->Reg data transfers
2. T_{data} - cycles taken from Q->L1 data transfers

Since for sparse matrices commonly the $nnzr$ is a small number, it is common to have Q = L1, which would then imply $T_{data} = 0$.

Verification

To verify this model we first conduct tests on dense matrices with varying sizes. The machine used is a single socket of Emmy (see Section 5.5). Since it is a dense matrix we can surely take $S_{\text{CL}} = 1$.

T$_{\text{nOL}}$ is derived in the following way:

- Machine capability of Emmy 1+1 $\left[\frac{\text{LD}}{\text{cycle}}\right]$ (2 ports) = 16 bytes/cycle in double precision without vectorization
- $\implies$ it would take 4 cycle each for loading a cache line of x, col, and val. Accounting for 12 cycles in total for an update of 8 non-zeros.
- $T_{\text{nOL}} = \frac{12}{8}\left[\frac{\text{cycles}}{\text{non-zeros}}\right] = \frac{3}{2}\left[\frac{\text{cycles}}{\text{non-zero}}\right] = \frac{3}{8}\left[\frac{\text{cycles}}{\text{Flop}}\right]$, since for KACZ we have 4 Flops/nnz.

Similarly we can also calculate T_{data} depending on Q. For example if Q = L2, $T_{\text{data}} = \frac{20}{32}\left[\frac{\text{bytes/nnzr}}{\text{bytes/cycle}}\right] = \frac{20}{32*4}\left[\frac{\text{cycles}}{\text{Flops}}\right]$, here 32 $\left[\frac{\text{bytes}}{\text{cycle}}\right]$ is the data transfer rate from L2 to L1 on Emmy.

After deriving T_{nOL} and T_{data}, we could just plug in this data along with the SPMTV performance to derive the KACZ performance (P) as mentioned in eqn. 7.4

Fig. 7.1 shows the results obtained, here the x axis depicts the number of non-zeros per row ($nnzr$) which is equal to number of columns ($ncols$) of the dense matrix. We see 2 predictions: the blue line showing the model without any vectorization and brown line showing the model with an AVX vectorized inner loop. It should be noted that the access to the x vector cannot be vectorized due to non-consecutive data access and the only possible candidate for vectorized loads is the val vector. Therefore in the vectorized version T_{nOL} is taken as 10 cycles/CL instead of 12 cycles/CL, since if the access to val is vectorized we can load it in 2 cy/CL instead of 4 cy/CL.[2] .

An observation from the result in Fig. 7.1 is that the non-vectorized model is closer to the measurement than the vectorized version, the reason being that the vectorization of small loops might not give sufficient speed-up. Furthermore we can observe that the model captures the drop in performance as the cache Q changes from L1 to L2 at $nnzr \approx 1,000$.

Scaling prediction

The scalability of the code in ECM model is predicted using the scaling limit (n_s) which is given as follows [58]:

$$\begin{aligned} n_{\mathrm{s}} &= \frac{\textit{cycles per CL overall}}{\textit{cycles per CL at the bottleneck}} \\ &= \frac{\textit{cycles per Flop overall}}{\textit{cycles per Flop at the bottleneck}} \\ &\text{since here, } \textit{FLop/CL = const = 32} \end{aligned} \tag{7.5}$$

On the Ivy Bridge architecture (Emmy) the scaling bottleneck is the main memory. Till now we have not differentiated between the data transfers at various memory hierarchies in the KACZ algorithm but we accounted for it by the SPMTV performance (implicitly by factor α), so we need to measure the MEM->L3 data volume (denoted by M) explicitly using a tool like LIKWID. This would imply that the scaling limit (n$_s$) is given by:

$$\begin{aligned} \textit{cycles per Flop at the bottleneck} &= T_{\mathrm{L3MEM}} \\ &= \frac{M}{4 * nnz}\left[\frac{\text{bytes}}{\text{Flop}}\right] * \frac{f}{BW}\left[\frac{\text{cycle/s}}{\text{bytes/s}}\right] \\ &\text{where } BW \text{ - full socket main memory bandwidth} \end{aligned} \tag{7.6}$$

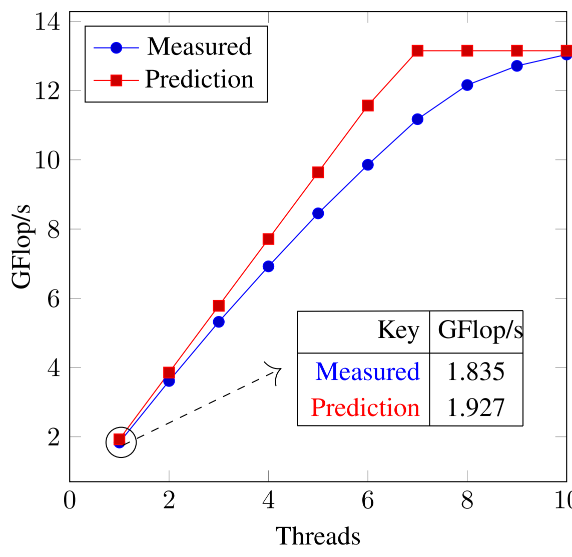
$$\implies \boxed{n_{\mathrm{s}} = \frac{C_{\mathrm{SPMTV}} + T_{\mathrm{nOL}} + T_{\mathrm{data}}}{T_{\mathrm{L3MEM}}} = \frac{C}{T_{\mathrm{L3MEM}}}} \tag{7.7}$$

Once the scaling limit (n_{s}) is derived one could linearly scale the performance derived for single thread, until n$_s$ is reached, after which the performance remains constant due to the bottleneck.

Fig. 7.2 shows a comparison between the measured value and the model for 2 sparse matrices: PWTK (see Appendix 10.1.3) and Graphene-4096-4096 (see Appendix 10.1.2). Here we have used the model with no vectorization for the comparison, as we have very small $nnzr$ ($\approx$ 53- PWTK and 13- Graphene). The prediction gives us confidence about our statement that extra inner cache transfers caused by the extra loop are to be accounted for the loss in factor 2 compared to SPM(T)V algorithm at lower thread counts.

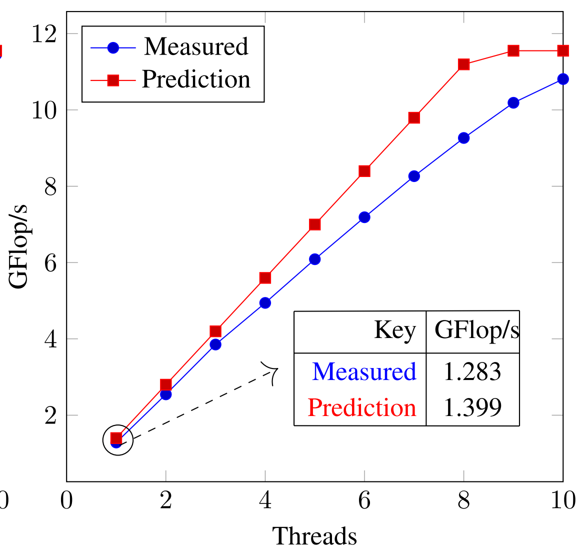
An interesting performance optimization that we were able to detect due to modeling was in the case of a non-normalized equation, where the rownorm also had to be calculated (see code snippet 7.1) . From the modeling we learn that we have mainly 2 bottlenecks: main memory and inner cache transfers due to second loop. Therefore calculating rownorm shouldn't bring a big change to the performance of our loop, since it doesn't affect any of these bottlenecks. But if one naively implements the kernel, we could see a significant performance drop. Moreover the difference between the prediction and measured value increases as the $nnzr$ increases.

[2]Note: on Emmy only half-wide vectorized loads are possible



Key	GFlop/s
Measured	1.835
Prediction	1.927

(a) PWTK



Key	GFlop/s
Measured	1.283
Prediction	1.399

(b) Graphene-4096-4096

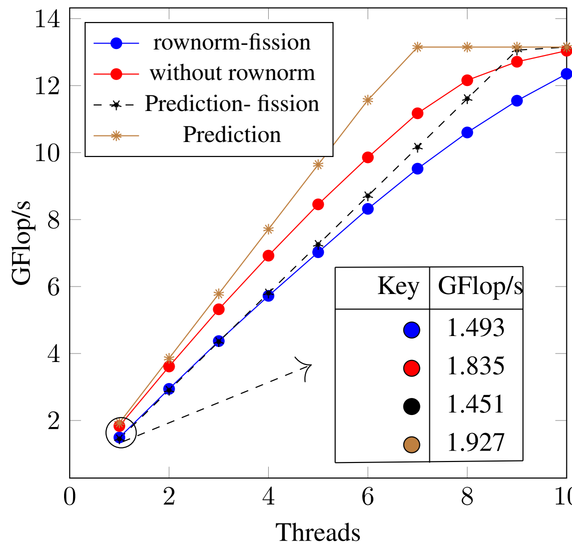
Figure 7.2: Comparison between model and measured values for PWTK and Graphene-4096-4096 matrices. Inset table shows clearly the performance for a single thread.

Code Snippet 7.1: Intel Compiler uses loop fission for 1^{st} loop

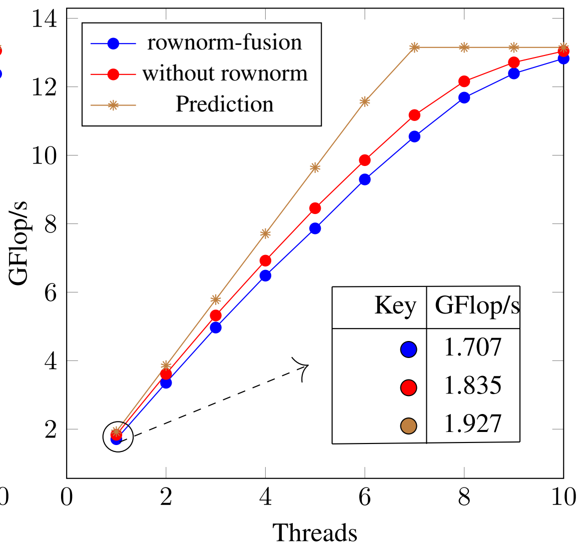
```
for (int row=start; row<end; ++row){
        double scale = 0;
        double norm = 0.;
        int  idx = chunkStart[row];
        scale  = b[row];

        for (int j=0; j<rowLen[row]; ++j) {
                scale -= val[idx+j] * x[col[idx+j]];
                rownorm += val[idx+j]*val[idx+j];
        }

        scale /= (MT)rownorm;
        scale *= omega;

        for (ghost_lidx j=0; j<rowLen[row]; j++) {
                x[col[idx+j]] +=  scale*val[idx+j];
        }
}
```

Code Snippet 7.2: Fission of 1^{st} loop is avoided

```
for (int row=start; row<end; ++row){
        double scale = 0;
        double norm = 0.;
        int  idx = chunkStart[row];
        scale  = b[row];

        for (int j=0; j<rowLen[row]; ++j) {
                double val_idx = val[idx+j];
                scale -= val_idx * x[col[idx+j]];
                rownorm += val_idx*val_idx;
        }

        scale /= (MT)rownorm;
        scale *= omega;

        for (ghost_lidx j=0; j<rowLen[row]; j++) {
                x[col[idx+j]] +=  scale*val[idx+j];
        }
}
```



(a) PWTK: loop fission



(b) PWTK: loop fusion

Figure 7.3: Shows the effect of compiler optimizations. In fig. 7.3a Intel compiler splitted the first loop, for the code snippet 7.1. In code snippet 7.2 the fission is avoided and subsequent performance gain could be observed in fig. 7.3b.

Fig. 7.3a shows the disparity between prediction (brown line) and the measured performance (blue line). The red line corresponds to the KACZ sweep for a normalized system.The difference corresponds to $nnzr$ extra loads from L1 to register, as shown by the dotted black lines in Fig. 7.3a.

If one further looks into the assembly code, the reason for this is evident since one could see that the Intel compiler performs a loop fission on the first loop, thereby making calculation of rownorm in a separate loop[3]. In order to prevent this we just need to pre-load the value and store it in a temporary as shown in code snippet 7.2. Fig. 7.3.b shows the result after making the corresponding change, where the blue curve (with rownorm) is almost close to the red curve(without rownorm).

7.2 Block Vectors and Layer Condition

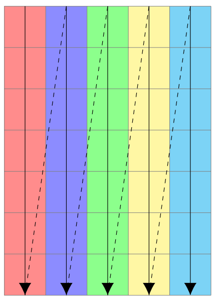
In the previous chapters we analyzed the KACZ kernel applied to a single right hand side (RHS), but normally we have multiple RHS to solve. One approach is to solve them one by one. But since all systems are independent to each other we could use this to our advantage and solve all of them together, thereby benefiting from:

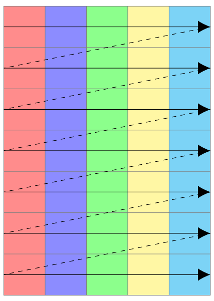
1. Lesser overall data traffic, since the matrix needs not be reloaded.

2. Additional parallelization (vectorization), since the systems are independent to each other.

[3]We think this is because after automatic unrolling (Modulo variable expansion) the compiler assumes there wouldn't be sufficient registers

In order to exploit the advantages mentioned above one has to store the vectors in row-major format as shown in the figure (right). The only difference from the previous KACZ code shown in code snippet 7.2 is that one has to add an additional loop over the number of RHS vectors (i.e. number of columns of the block vector in row-major format, denoted as $nblocks$). The code then takes the form shown in code snippet 7.3


1. Column-major


2. Row-major

Code Snippet 7.3: Block vector

```
for (int row=start; row<end; ++row){
        double scale[nblocks] = {0};
        double norm = 0.;
        int  idx = rowptr[row];
        for(int block=0; block<nblocks; ++block) {
                scale[block]  = b[row*nblocks+block];
        }

        for (int j=0; j<rowLen[row]; ++j) {
                double val_idx = val[idx+j];
                rownorm += val_idx*val_idx;
                for(int block=0; block < nblocks; ++block) {
                        scale[block] -= val_idx * x[col[idx+j]*nblocks + block];
                }
        }

        for(int block=0; block < nblocks; ++block) {
                scal[block] *= omega/(MT)rownorm;
        }

        for (ghost_lidx j=0; j<rowLen[row]; j++) {
                double val_idx = val[idx+j];
                for(int block=0; block < nblocks; ++block) {
                        x[col[idx+j]*nblocks + block] += scale[block]*val_idx;
                }
        }
}
```

The code intensity of the above code is then given by:

$$I_{nblocks} = \frac{4 * nblocks}{\underset{\uparrow\,\text{val}}{8} + \underset{\uparrow\,\text{col}}{4} + (\underset{\uparrow\,\text{x}}{16} * \alpha + \underset{\uparrow\,\text{b}}{\frac{8}{nnzr}}) * nblocks} \quad \frac{Flops}{Bytes} \tag{7.8}$$

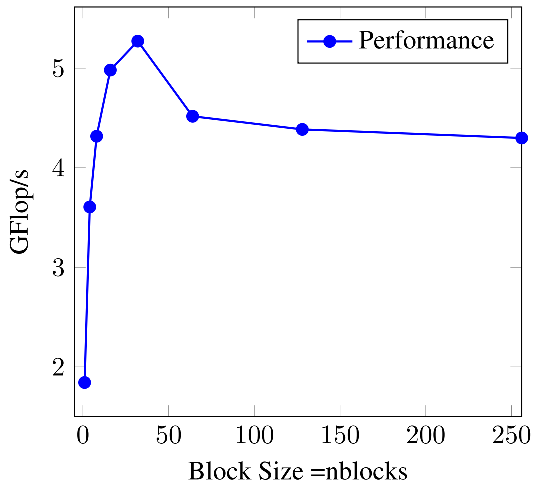
The reason for numerator being 4 instead of 6 is to have a common baseline for non-normalized and normalized system as mentioned in Section 5.5. Equation 7.8 shows that with increasing $nblocks$ the code intensity increases until it hits the limit:

$$I_{\infty} = \frac{4}{16 * \alpha + \frac{8}{nnzr}} \quad \frac{Flops}{Bytes} \tag{7.9}$$

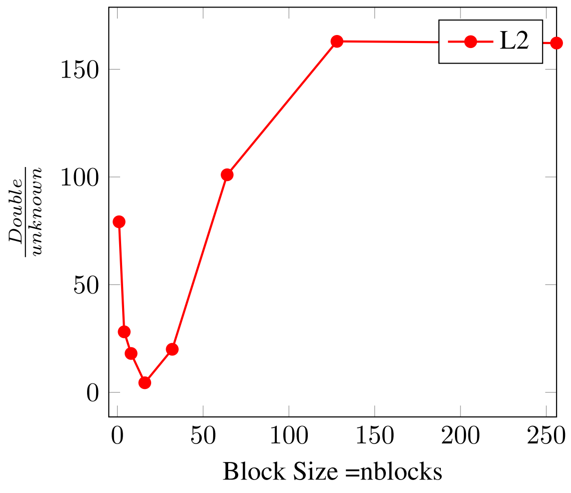
where the contribution from the matrix (i.e. val and col) completely disappears. But practically this limit (I_{∞}) is not achievable, since as the number of blocks exceeds a certain limit the data traffic violently increases. This limit is governed by two factors:

1. reuse distance of the matrix
2. cache sizes

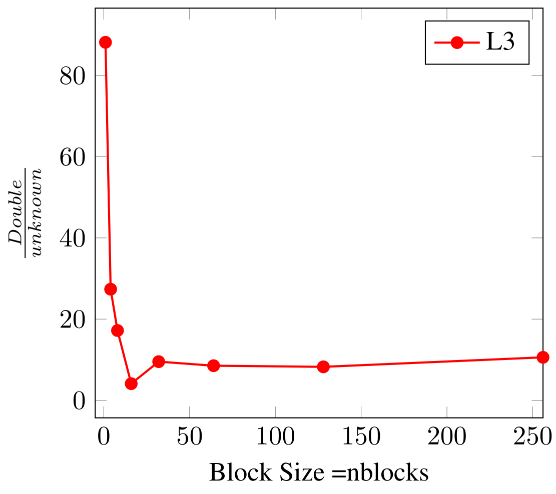
In order to illustrate this effect we run the code serially with varying block sizes and measure the performance and data transfer volume for various memory hierarchies.



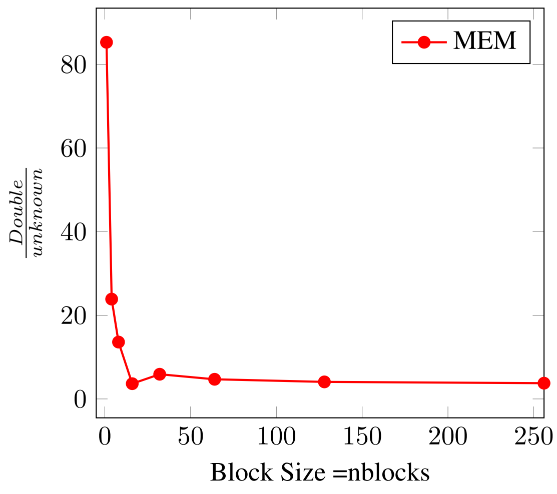
(a) Block Size vs Performance



(b) Data traffic L2



(c) Data traffic L3

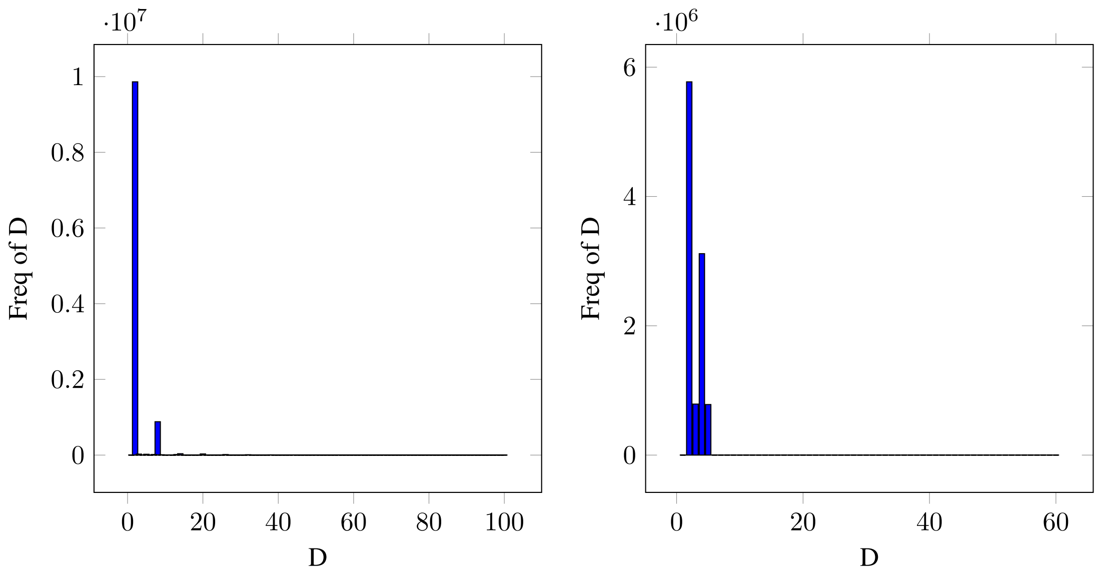


(d) Data traffic MEM

Figure 7.4: Effect of increasing block vector size on PWTK matrix for a single thread. Fig. 7.4a shows the effect on performance as block vector size increase. Fig. 7.4b, 7.4c, 7.4d shows the corresponding data traffic normalized to the number of unknowns (elements in x-vector) as measured using LIKWID [63] for L2, L3 and Memory respectively.

Fig. 7.4a shows the performance as a factor of block size ($nblocks$) for the PWTK matrix. We see initially a steep increase in performance since the intensity of code increases, but then it starts to drop at a block size of about 32, due to the increase in data-traffic. Fig. 7.4b, 7.4c and 7.4d plot the number of words (doubles) per unknown (elements in x vector) as measured by LIKWID for L2, L3 and main memory. We see initially for a single vector ($nblocks$=1) around 85 words/unknown from all memory hierarchies, this comprises of 53 ($nnzr$ for PWTK) doubles from val, 53 integers from col, 1 double from b and almost 2 doubles[4] from x (assuming $\alpha = 1/nnzr$), accounting for 82.5 words (see Code Snippet 7.3 for the notation). As the number of blocks increases we see a steep drop of the required $\frac{double}{unknown}$, as predicted by Eqn.7.8. At about 16 $nblocks$ it reaches very close to the theoretical minimal data traffic of 3 $\frac{double}{unknown}$, where we have effectively decoupled from the matrix and only contributions from the vectors x and b exist. Note that at this point the data traffic is almost the same as for stencil like algorithms, where we do not explicitly

[4]2 - since we need to consider load and store



(a) PWTK **(b)** Graphene-1024-1024

Figure 7.5: Histogram: D vs Frequency of D. The histogram reveals clustering of values near to small value of D, implying good reuse of caches.

store the matrix. But then we see at some point between 16 and 32 blocks the traffic starts to increase, and particularly the traffic between L2 and L1 increases violently to 160 words/unknown (see Fig. 7.4b), which is due to a violation of layer condition (see below).

Even though the values that we see in the above example are specific for the matrix under consideration, the behavior is similar for other matrices.

Layer Condition

The reuse of an element $x[i]$ of the RHS vector x at position 'i' depends on the column entries of the 'ith' row of the matrix A. For a sparse matrix the reuse distance (RD) (for more details see [26]) can be formulated as the number of distinct elements accessed between 2 successive 'ith' column entries. If the number of rows between the 2 successive 'ith' entries is D we can approximately express RD as:

$$RD = \underset{\uparrow\ \text{val}}{nnzr * D} + \underset{\uparrow\ \text{col}}{\frac{nnzr * D}{2}} + \underset{\uparrow\ \text{b}}{D} + 2 * \underset{\uparrow\ \text{x}}{D * nblocks * \alpha * nnzr} \tag{7.10}$$

This implies if the RD elements can be kept in a particular cache, the next access to $x[i]$ wouldn't need to be reloaded from the outer memory hierarchies,i.e., we need:

$$RD * 8 \leq cache\_size \tag{7.11}$$

This condition is known as the layer condition, a violation of this condition might result in excessive data transfer from the memory hierarchies below it. Fig. 7.5 shows histogram calculating the frequency of occurrence of the distance D for PWTK and Graphene-1024-1024. For these matrices we see the clustering of values at small row distances D. Note that the value of D shown in the figure is limited to the area where the frequency is significant, outside this area values are zero (or negligible in comparison). This implies that for these matrices most of the elements gets reused in the lower level cache (like L1).

But as we increase the block vector size ($nblocks$), we see from the Eqn. 7.10 a subsequent increase in RD. This increase would further lead to a violation of the layer condition (eqn. 7.11), for some distance D. The seriousness of the violation greatly depends on the frequency of D.

In our previous example in Fig. 7.4, we see a high frequency of D occuring below the value 10 (see Fig.7.5a), which implies a violation of lowest level cache (L1) would occur first as we increase the block size ($nblocks$) , thus leading to the drastic increase in data transfers from L2 which can be observed from Fig. 7.4b. This finally leads to the performance drop seen in Fig.7.4a.

Hybrid Storage

In order to avoid the performance drop a simple approach is to have a hybrid data storage, between column and row major format as shown in Fig. 7.6. If the block size ($nblocks\_opt$) is chosen as the optimum size where the performance is maximum, the approach could maintain this performance throughout all block sizes. In the current version of GHOST [44] it is left to the user for storing in such a data format, however the implemented KACZ kernel has a feature which suggests an optimal block size ($nblocks\_opt$) to the user for a given matrix.

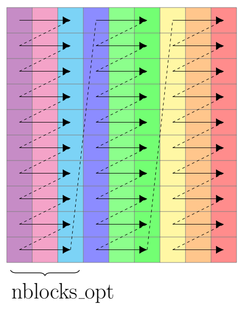


Figure 7.6: Hybrid Storage

7.3 Case study of complex shifts

An important application of the KACZ algorithm in the ESSEX project is to develop an eigenvalue solver, which will be explained in Chapter 8. But in order to implement it we need the KACZ solver to be able to deal with complex numbers and shifts. So, in this section we discuss various strategies we employed to optimize the performance of this complex variant of KACZ. The basic equation we need to solve is as follows :

$$(A - \sigma I) * x = b \tag{7.12}$$

where, σ - shift

In a typical example we have A and b as real matrix and vector respectively, but σ being a complex number. This implies the unknown vector x is complex. Implementation is straight forward by using the complex class from the C++ standard library. The algorithm can be found in the Appendix 10.2

Since here we have a multiplication between a real (d) and a complex (z) number we expect almost the same or higher performance compared to a real (d)-real (d) multiplication. If the x vector has to be loaded from the main memory we have 2 $\times$ more Flops and 2 $\times$ increased data traffic from the x vector, which in effect gives almost the same code intensity (assume $nblocks$ is sufficiently high to be decoupled from matrix traffic). But if the x vector comes from cache we expect in general a higher performance of the shift (d-z) variant due to increased code intensity. Typically, if all the caches are scalable one could expect a factor of 2 increase in performance of the shifted version at saturation.

Fig. 7.7 shows the performance gain at different stages of optimization carried out on this kernel, for a block size of 16 (to decouple from matrix contribution) and one complex shift on PWTK matrix. The bar on the extreme right is the case with no shift, (i.e. no complex computation), which acts as our baseline.

The first (left most) bar shows the performance of the kernel implemented with the complex class from the standard library. A comparison with the no-shift case reveals a 14 $\times$ lower performance of the shifted variant, indicating far from optimal performance. A study of the assembly code reveals that with this implementation the compiler was unable to vectorize the code.

The second (from left) bar makes clear the inability of the compiler to handle multiplications of type d*z properly. In this version we explicitly computed the multiplications of type $d * z$, without using the standard library, which immediately yields a factor of 7 compared to the naive version. A part of the gain results from vectorization.

But still we are not at the limit, since we have a factor of two between the code with explicit complex multiplication and the double- double case. Finally we use intrinsics to have better vectorization, the performance is shown as 3^{rd} bar from left in Fig.7.7. Now we see the shift version wins over the non-shifted (d-d) version by a factor of 1.35 $\times$, matching our prediction since we know for PWTK matrix the x vector ($nrows$ = 217918) is sufficiently small to reside in L3 cache, causing an increased code intensity compared to the non-shifted (d-d) version. But note that since the code was not able to saturate the main memory (compare the linear scaling between 1 and 10 threads: performance at 1 thread= 5.67 GFlop/s and at 10 threads = 55.4 GFlop/s) we couldn't see a factor of 2 difference.

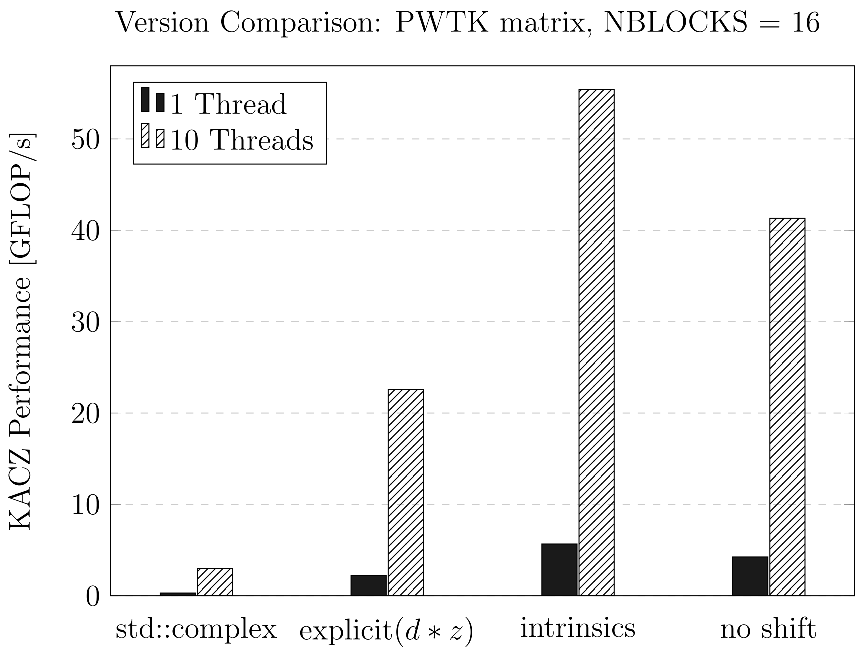


Figure 7.7: Performance comparison at various stages of optimization

8 Application Run - FEAST

FEAST [54] is a numerical algorithm to find inner eigenvalues of Hermitian and non-Hermitian matrices. It was devised by Eric Polizzi in 2009. Contrary to the conventional Krylov subspace methods or Jacobi-Davidson techniques, FEAST works on the basis of numerical contour integration and it is density matrix-based algorithm. The numerical integration requires a solver for a shifted system and it has been shown in [32] that CGMN (CARP-CG) is a good alternative to the existing direct methods to solve the inner system in FEAST. In this chapter we discuss about an implementation of FEAST using the proposed KACZ_BMC kernel for the CARP-CG solver. Furthermore, a comparison with the Intel® MKL Extended Eigensolver (FEAST), will be illustrated.

8.1 Eigenvalue Problem

In linear algebra, a generalized eigen value problem is defined as follows:

For given $A \in \mathbb{R}^{N\times N}$ and $B \in \mathbb{R}^{N\times N}$, find eigenvalues λ_i (scalar) and eigenvectors $v_i \in \mathbb{R}^N$, such that they satisfy:

$$Av_i = \lambda_i B v_i \tag{8.1}$$

In this thesis we restrict ourselves to solving symmetric-definite eigenvalue problems where A and B are real symmetric matrices ($A = A^\intercal$ and $B = B^\intercal$) and B is positive definite. Due to this restriction the problem has desirable properties like all eigenvalues (λ_i) are real and the eigenvectors can be made mutually B-orthogonal to each other,i.e.,

$$V^\intercal B V = I \tag{8.2}$$

$$\text{where, } V = \{v_1, v_2, v_3, ..., v_N\} \in \mathbb{R}^{N\times N}$$

Often in practical applications of large sparse matrices one is interested only in a subset of interior eigenvalues [λ_{min}, λ_{max}] and corresponding eigenvectors . A common procedure to solve such a problem is known as the Rayleigh-Ritz method [59]. In this method one tries to reduce the problem into a suitable subspace of dimension $m \leq N$ containing the region of interest and then solve the reduced dense problem using a direct method. From a performance point of view the method used to construct the subspace greatly matters since in this stage one needs to deal with systems as large as the dimension of the original problem, compared to the reduced size problem which would be of the order of few 100-1000's. Many algorithms like Jacobi Davidson [56, 59] and Krylov based methods [59] exist to construct the subspace. A relatively modern approach in this category is the FEAST algorithm, which is discussed in the next section.

8.2 FEAST algorithm

Let $I_\lambda = [\lambda_{min}, \lambda_{max}]$ be the sought after eigenvalue interval and let M be the reduced subspace dimension. The FEAST algorithm generates a subspace (Ritz basis) using a spectral projection technique [61].

It basically has 2 steps

1. Subspace construction - Contour integration of resolvent function $G(\sigma) = (\sigma B - A)^{-1}$, post multiplied with a set of M linearly independent vectors $Y \in \mathbb{R}^{N \times M}$ i.e.

$$Q = \frac{1}{2\pi i} \int\limits_C G(Z) Y dZ \tag{8.3}$$

   where, C is the contour of integration containing I_λ

   $\sigma \in \mathbb{C}$

   $Y = y_1, y_2, y_3, ..., y_M$ set of linearly independent vector

   The generated vector $Q = q_1, q_2, q_3, ..., q_M$ is a new set of independent vector.

2. Rayleigh Ritz procedure using q_i as Ritz basis.

The FEAST algorithm as illustrated in [54] is shown in Algorithm 10. The algorithm is presented here for clarity in the rest of the explanations.

Algorithm 10 FEAST algorithm as proposed in [54]

```
1:  Select M_o > M random vectors Y ∈ $\mathbb{R}^{N \times M_o}$
2:  Set Q = 0, Q ∈ $\mathbb{R}^{N \times M_o}$; $r = (\lambda_{max} - \lambda_{min}/2)$
    {Gauss integration, $N_e$ = No. of integration points}
    {$x_e$ = Gauss points, $\omega_e$ = Gauss weights}
3:  for $e = 1, ..., N_e$ do
4:      Compute $\theta_e = -(\pi/2)(x_e - 1)$
5:      Compute $\sigma_e = (\lambda_{max} + \lambda_{min})/2 + r\ exp(i\theta_e)$.
6:      Solve $(\sigma_e B - A)Q_e = Y$; $Q_e \in \mathbb{C}^{N \times M_o}$
7:      Compute $Q = Q - (\omega_e/2)\Re(r exp(i\theta_e)Q_e)$
8:  end for
    {$Rayleigh - Ritz$}
9:  Set $A_Q = Q^\intercal AQ, B_Q = Q^\intercal BQ$; $A_Q \in R^{M_o \times M_o}$ and $B_Q \in R^{M_o \times M_o}$
10: Solve reduced eigen value problem $A_Q \phi = \epsilon B_Q \phi$; for eigenvalues $\epsilon$ and eigenvectors $\phi \in R^{M_o \times M_o}$
11: Set $\lambda_m = \epsilon_m$ and compute $X = Q * \phi$, If $\lambda_m \in [\lambda_{max}, \lambda_{min}]$, $\lambda_m$ is an eigenvalue solution and its
    eigenvector is $X_m$(m^th column of X).
12: Check convergence for trace of eigenvalue. If convergence not satisfied, set $Y = BX$ and go to line 2.
```

The integration in step 1 (Eqn.8.3) (marked as red in Algorithm 10) is commonly performed using a numerical integration scheme known as Gauss-Legendre quadrature. In order to compute the integrand (eqn.8.3) at each Gauss node we need to solve an M-block linear system as follows:

$$(\sigma B - A)V = Y \tag{8.4}$$

This is computationally very demanding since we need to solve M systems of dimension $N \times N$ at each Gauss point. Fig. 8.1 illustrates the Gauss nodes for circular integration contour. For small and moderate size matrices direct methods have been shown to be efficient for solving shifted system. But for large sparse matrices this becomes increasingly difficult. The iterative solution of these linear systems remains also challenging because they are highly indefinite and ill-conditioned as shifts approach imaginary axis. The spectrum can also be very dense in the interval of interest, making popular preconditioners inefficient. In [32] it was shown that CARP-CG has the potential to tackle these systems in some cases. In the rest of the chapter we discuss the implementation of a version of FEAST using CARP-CG (KACZ_BMC kernel) as the solver.

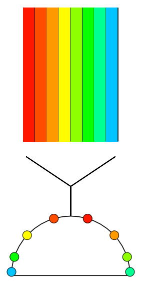


Figure 8.1: Solve M system at each Gauss points (adapted from [64])

Some Remarks for Implementation

In our current implementation of FEAST we restrict ourselves to the standardized eigenvalue problem,i.e., the case where $B = I$. 10.

We usually need to solve for many y_i (i.e., Y is a block vector), this can be used to our advantage by solving multiple equations at the same time as shown in Section 7.2. One has to be also careful not to violate layer conditions (see Section 7.2) that might degrade the performance. In order to do this we initially find an optimum block size for a given matrix and arrange the blocks in hybrid storage as mentioned in Section 7.2. Section 7.3 also shows different strategies employed to achieve optimum performance for complex shifted systems which we encounter due to Eqn. 8.4 where σ is a complex number.

Parallelization along the number of shifts (different colors in Fig. 8.1) is also possible. Normally we avoid this due to problems that happens with load balancing since the number of iterations generally increase as the shifts get closer to the imaginary axis owing to the poor condition number.

A major confusion that arises while using KACZ_BMC is due to unsymmetric permutations. It can be shown that eigenvalues of a symmetrically permuted matrix (see sec. 2.2) remains the same as that of the unpermuted matrix. However, this does not hold in case of an unsymmetric permutation. The reason for this follows from Eqn. 2.3 as shown below:

$$\begin{aligned}
&Ax = \lambda x && \\
&AP_{\pi_r}^{\mathsf{T}} x = \lambda P_{\pi_r}^{\mathsf{T}} x && \text{Post-multiply with } P_{\pi_r}^{\mathsf{T}} \\
&P_{\pi_r} A P_{\pi_r}^{\mathsf{T}} x = \lambda P_{\pi_r} P_{\pi_r}^{\mathsf{T}} x && \text{Pre-multiply with } P_{\pi_r} \\
&P_{\pi_r} A P_{\pi_r}^{\mathsf{T}} x = \lambda x && P_{\pi_r} P_{\pi_r}^{\mathsf{T}} = I \text{ (orthogonal matrix)}
\end{aligned}$$

In order to resolve this problem one has to take extreme care of the permutations applied to each vector and ensure that they appear in correct space (domain or range space), such that the reduced system remains unaffected by the permutations. There are basically 2 approaches to this problem:

1. A straight forward approach is to store both the original unpermuted matrix A and the permuted matrix A_{π_r,π_c}. The permuted matrix is then used only for KACZ kernels, and once these kernel operations are done the result gets permuted back to original indices. In algorithm 10 this would imply to use A_{π_r,π_c} instead of A in line 6 and a permutation of vector Q to original index just before line 9 using the column permutation (π_c).
2. A much cleverer approach to solve this problem is to just switch the side of the vector Q^{T} appearing in line 9 from right-sided (column space) to left-sided (row space).

Of the 2 approaches the 1^{st} approach is expensive and requires storage of another large sparse matrix. The 2^{nd} approach has just the cost of two permutations (switch operation) on a (block) vector. Although this approach seems to be surprising at first look, the principle is fairly simple. The main aim of any permutation is to get the reduced problem in its original indices (i.e., without any permutations), therefore A_Q and B_Q has to be in original indices. Below it is shown why switching $Q^\intercal$ satisfies this requirement. The subscript RP, CP, and NP refers to row permutation (π_r), column permutation (π_c) or no permutation of the corresponding dimensions, the line numbers refer to the lines in Algorithm 10.

- line 6 : $(\sigma_e B - A)_{(RP,CP)}\; Q_{e(CP,NP)} = Y_{(RP,NP)}$; note that the 2^{nd} dimension of Y and Q_e is unpermuted since we don't permute between different y_i.
- line 7: $Q_{(CP,NP)} = Q_{(CP,NP)} - (\omega_e/2)\Re(rexp(i\theta_e)Q_{e(CP,NP)})$
- line 9: We break line 9 to small parts for clarity:
  - $temp_{(RP,NP)} = A_{(RP,CP)}Q_{(CP,NP)}$
  - $\boxed{\bar{Q}^\intercal_{(NP,RP)} = switch(Q^\intercal_{(NP,CP)})}$ $\leftarrow$ only extra step
  - $A_{Q(NP,NP)} = \bar{Q}^\intercal_{(NP,RP)} temp_{(RP,NP)}$
  - Similarly for $B_{Q(NP,NP)}$

Note that at the last step A_Q and B_Q are without any permutations, leaving it in unpermuted state. But the problem with such an approach compared to the 1^{st} approach is that such a transformation requires knowledge of the algorithm or at least a check in each of the existing kernels for compatibility. Whereas in the 1^{st} approach permutations only have to be taken care of in KACZ (CARP).

As mentioned earlier in Chapter 6, automatic and transparent permutations have been made optionally available in GHOST which simplifies the implementation of such complex scenarios.

8.3 Test and Results

FEAST tests are run using 1 socket of Emmy (see Section 5.5 for more details). The test is set up to solve for 10 inner eigen values of the Laplacian problem on a 3D cubic domain. The interval $[\lambda_{min}, \lambda_{max}]$ is controlled in a manner to contain only 10 eigen values by using the relation $\lambda_{min} = 0.455/(4^{\log_2(n)-3})$ and $\lambda_{max} = 1.8/(4^{\log_2(n)-3})$, where n is the dimension of 1 side of the cubic domain. The starting vector space Y is chosen to be a random vector $\mathbb{R}^{N\times M}$ vector, where the initial size of search space (M_o) if chosen to be 16. The convergence of the algorithm is checked using the trace of the eigenvalue and the stopping criterion is as follows:

$$\frac{|trace_{k+1} - trace_k|}{max(\lambda_{min}, \lambda_{max})} < \epsilon \tag{8.5}$$

where, $k-$refinement level (outer iteration)

$trace_k-$sum of eigen values in level k

The test tolerance (ϵ) is chosen to be 10^{-7}. Until this tolerance level is achieved iterative refinement is carried out.

The test problem is solved using two implementation of FEAST, one with our CARP-CG (CGMN) solver and the second using the Intel MKL Extended Eigensolver [8] which uses PARDISO [10] to solve the inner linear system. The time required to arrive at the solution as well as the memory footprint of the algorithms are measured.

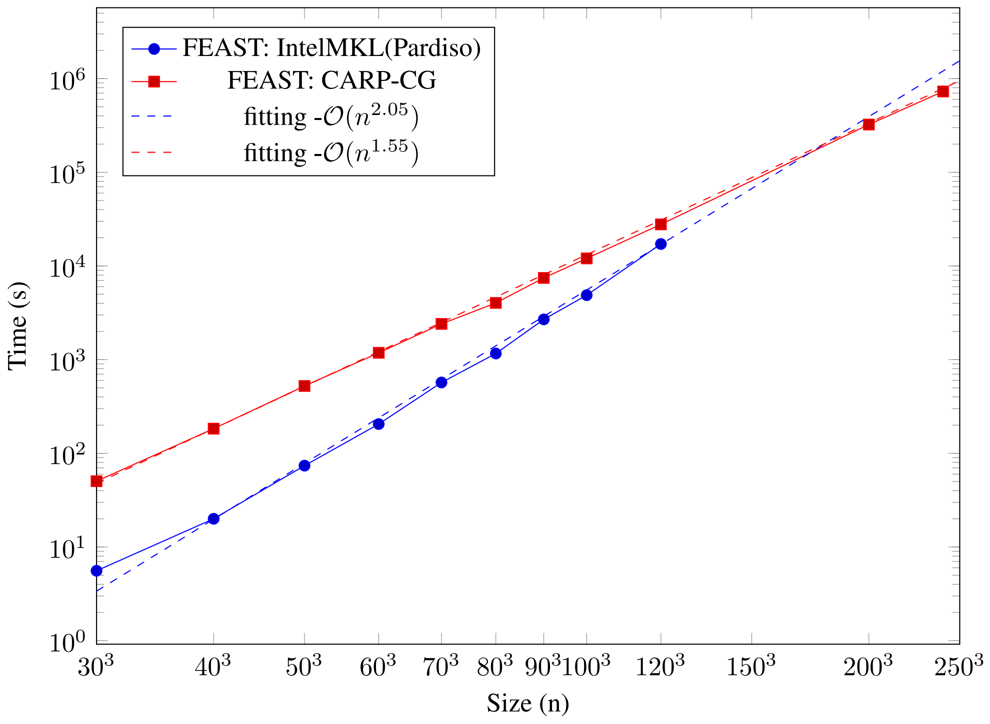
Fig. 8.2a shows the time required to reach the specified tolerance level by both versions of FEAST. We see that initially for small problem sizes Intel MKL has an advantage over the CGMN (CARP-CG) version. But as the size increases we see the difference between the two gets smaller and smaller. The reason for this is because Intel MKL uses PARDISO to solve the inner linear system, which in turn is a direct solver. Being a direct solver the computational complexity of the algorithm is high, $\mathcal{O}(n^2)$ in this case. The fitting line (dotted line in Fig. 8.2a) underlines this fact. Whereas on the other hand CGMN (CARP-CG) being an indirect solver with theoretical complexity of $\mathcal{O}(n^{3/2})$ [1], the slope of the time curve is not as high as that of direct methods. But due to the higher value of constant in front of the Big $\mathcal{O}$ notation in case of indirect method, initially the MKL version outperforms our implementation of FEAST. Subsequently, as the size is increased we see that at some point (here about n=160), a breakpoint is reached where both performs equally good and after that our implementation dominates.

Unfortunately we couldn't run the MKL version of FEAST till we reach the breakpoint since it went out of memory before this could happen. In order to study how the memory requirement of the solver evolves with size we plot the maximum memory footprint of both algorithms in Figure 8.2b. Here we see for the direct solver method that a fitting of the curve corresponds to $\mathcal{O}(n^{1.4})$ memory complexity which is close to the theoretical complexity of $\mathcal{O}(n^{4/3})$. But on the other hand our method has a complexity of $\mathcal{O}(n)$. If one observes carefully at smaller domain sizes (n = 30 - 60) CGMN has a higher complexity than $\mathcal{O}(n)$, we think this is because we use direct methods available in Intel MKL to solve the reduced eigen value problem and at this region the complexity of this reduced problem is dominant. But as we see from the fit asymptotic behavior of the measurements is $\mathcal{O}(n)$. This is another big advantage of an indirect method like CGMN or CARP-CG, where the memory required is less, enabling larger problem to be run easily, which is in particular the focus of ESSEX project.

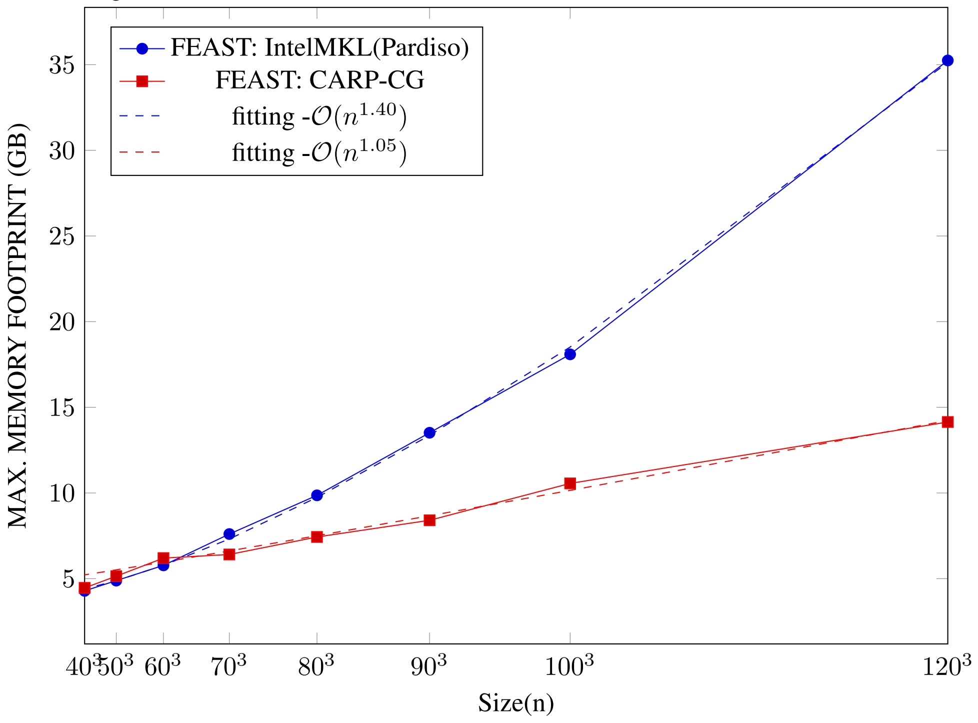
Yet another important point is the need of distributed memory parallelization e.g., using MPI. If one observes the time required to solve on a domain size of 240^3 this is evident, since it takes nearly 8 days to solve it on a single socket of Ivy bridge for CGMN and even more for MKL (from fitting). The ease of parallelization based on domain decomposition method for iterative methods like KACZ (CARP) [2] adds another plus point to this kind of solvers, whereas this is difficult using direct solvers. In fact the MKL version of FEAST does not currently support MPI, whereas FEAST Eigenvalue Solver library [6] which also uses PARDISO [10] provides support for MPI, but currently only exploits a second level of parallelism by solving different independent linear system across different nodes. This approach cannot be used for large problem sizes since we again run into the same problem of memory requirement as the domain size increases.

[1] $\mathcal{O}(n^{3/2})$ could be broke to 2 parts $\mathcal{O}(n)$ from 1 KACZ/CARP iteration and $\mathcal{O}(\sqrt{n})$ from the increase in number of iteration required to reach the same tolerance level

[2] an experimental version of CARP already exists in GHOST and has been tested for FEAST algorithm



(a) Runtime comparison of FEAST CARP-CG vs Intel MKL, to obtain 10 inner eigen values of Laplacian in 3D with a tolerance level of 10^{-7}. Fitting to the obtained measurements are shown with dotted lines. Note that both the axis are in log-scale



(b) Comparison of memory requirement between FEAST CARP-CG and Intel MKL. Fitting to the obtained measurements are shown with dotted lines.

Figure 8.2: FEAST: Time and Memory Footprint

9 Conclusion and Future Work

9.1 Conclusion

The Kaczmarz kernel is a linear iterative solver based on the concept of row projections. The kernel has wide application in the field of imaging, computational fluid dynamics, quantum mechanics and many others. However, parallelisation of the kernel is not straightforward due to its data dependencies. This thesis mainly focuses on effective solutions to avoid such dependencies by reordering (or partitioning) the system. We studied the effect of two such methods, namely **multicoloring** (MC) and **block multicoloring** (BMC) mainly from a performance perspective. Although the thesis focuses on the Kaczmarz kernel, this approach could also be made use for kernels with similar dependencies like sparse matrix transpose vector and symmetric sparse matrix vector multiplication.

We have introduced the basic concepts of multicoloring methods and have looked at them from a performance perspective. A close study of performance results has shown that this approach was far from optimal for the modern processors. The insights obtained from the detailed analysis of the method has become the starting point for a better approach. We have started from a concept similar to the block partition method as introduced by Kamath and Sameh for tridiagonal matrices [41] and extended it to block multicoloring for general matrices. We have further introduced splitting of the transition zones (row partitioning) to produce a large number of orthogonal blocks. This has enabled us to extract further parallelism by relaxing the constraint. Different bounds based on matrix bandwidth and number of threads at which the scheme switches from one method to the other have been established. Based on these bounds we have employed some load balancing techniques. Later on the method has been extended to a constraint-free approach by combining it with multicoloring. We have then presented a point-by-point comparison between the block multicoloring method and the multicoloring method, which has clearly shown the superiority of the block multicoloring method.

Later a detailed performance model of the BMC approach has been constructed in order to ensure the optimality of the method. A model based on the performance of sparse matrix transpose vector (SPMTV) and the Execution-Cache-Memory (ECM) model [58] has been constructed for the Kaczmarz kernel. The close agreement of the model with the measurements has provided a strong indication that the hardware resources were used in an optimal way and has led to useful optimizations. Furthermore we have studied the effect of block vectors and have given useful insights on the effect of layer conditions on the block vector size.

Finally we have presented an application of the developed parallel variant of the Kaczmarz method using the block multicoloring approach. Here a conjugate-gradient (CG) accelerated variant of Kaczmarz has been used to solve the inner linear systems appearing in the FEAST eigenvalue solver. A comparison of the method with the widely used Intel MKL version of the FEAST algorithm has demonstrated the necessity of efficient iterative methods for solving large sparse linear systems. The promising results indicate that the method could be used as a tool for solving large inner eigenvalue problems that appear commonly in quantum chemistry and physics.

9.2 Future Work

The development and further research on the method will continue. The need of CARP for distributed memory parallelisation has been well evident from the results presented in the FEAST application run. An experimental version of the CARP with KACZ_BMC as a kernel is currently available in GHOST [44]. A difficulty here is the consecutive storage of remote entries in each process, which blows up the matrix bandwidth. A workaround for such a situation has been devised by padding the local matrix with extra column entries. Further validation and testing of the CARP kernel has to be done before going to production. The recent development on distributed memory RCM [21] is interesting in this context and has to be studied.

A detailed analysis of the trade-off between the number of threads and performance for the block multicoloring method at the pre-processing stage is still missing. The extension of the block multicoloring method and especially its relation to graph theory is yet another interesting field of study. This would enable us to get more useful insights. Initial studies have shown that this approach could enable more parallel work and better performance by fine-tuning.

The incorporation of a hybrid matrix format as described in Section 7.2 in GHOST library will prove to be beneficial as one could exploit the multiple levels of parallelism efficiently within the library itself. This would in turn contribute to the development of an efficient KACZ kernel on the modern manycore and GPU architectures.

The extension of all the KACZ kernels in GHOST to support the SELL-C-σ format is also future work to be done, as well as the application of block multicoloring to other kernels with similar kinds of data dependencies, for example sparse matrix transpose vector multiplication and symmetric sparse matrix vector multiplication.

10 APPENDIX

10.1 Matrices

10.1.1 BENCH3D-x-A0

- Description: 3D Laplacian on a cubic domain of size x $\times$ x $\times$ x
- Source : Essex Project
- Nrows : x^3
- NNZ : 7*x^3
- Symmetric

10.1.2 Graphene-x-x

- Description: The graphene matrices are obtained from the standard tight-binding Hamiltonian on the honeycomb lattice with an on-site disorder term [53] and [25].
- Source : Essex Project
- Nrows : x^2
- NNZ : $\approx$ 13*Nrows
- Symmetric

10.1.3 PWTK

- Description: Pressurized Wind Tunnel - Stiffness matrix.

- Source : The University of Florida Sparse Matrix Collection[15]
- Nrows : 217,918
- NNZ : 11,524,432
- Total Bandwidth : 378,662
- Total Bandwidth(RCM) : 4,058
- Symmetric

10.1.4 Spin-24-24-24

- Description: Models the Hamilton operator (H) for the Heisenberg XXZ spin chain model with S_z-symmetry [57].

$$H = \Sigma J_{xy}({\sigma^x}_i{\sigma^x}_j + {\sigma^y}_i{\sigma^y}_j) + J_z{\sigma^z}_i{\sigma^z}_j$$

- Source : Essex Project
- Nrows : 2,704,156
- NNZ : 35,154,028
- Total Bandwidth : 417,648
- Total Bandwidth(RCM) : 124,152
- Symmetric

10.1.5 TAU

- Description: Linear problem for aerodynamic gradients calculation. Transonic inviscid flow over Onera M6 wing.
- Source : DLR (German Aerospace Center)[4]
- Nrows : 541980
- NNZ : 170361408
- Total Bandwidth : 1045418
- Unsymmetric

10.1.6 Topi-36-36-36

- Description: Modelling of Hamiltonian operator with hopping of electrons along the three-dimensional cubic lattice and with a disorder potential for Topological insulator [53] [37].
- Source : Essex Project
- Nrows : 186,624
- NNZ : 2,405,376
- Total Bandwidth : 362,886
- Total Bandwidth(RCM) : 10,414
- Symmetric

10.2 Complex Shift Algorithm

KACZ_shift: solve for x: $(A - \sigma I)x = b$

```
for $row = 0 : nrows$ do
  $scale = b[row]$
  $mval\_diag = 0$
  if $unsymmetric\ permutation$ then
    $diag\_idx = ColumnPerm[inverseRowPerm[row]]$
  else
    $diag\_idx = row$
  end if
  $norm = 0$
  $idx = 0$
  for $(idx = rowptr[row]; idx < rowptr[row + 1]$ && $col[idx]! = diag\_idx; ++idx)$ do
    $scale- = val[idx] * x[col[idx]]$ {Complex(d-z) Multiplication}
    $norm+ = val[idx] * val[idx]$
  end for
  if $idx! = rowptr[row + 1]$ then
    $mval\_diag = mval[idx]$
  end if
  $mval\_sigma = mval\_diag - \sigma$
  $scale- = (mval\_sigma) * x[col[diag\_idx]]$ {Complex(d-d) Multiplication}
  $norm+ = (mval\_sigma * mval\_sigma^H)$ {$a^H \implies$ complex conjugate of $a$}
  $scale* = omega/norm$
  for $(idx = idx + 1; idx < rowptr[row + 1]; idx++)$ do
    $scale- = val[idx] * x[col[idx]]$ {Complex(d-z) Multiplication}
    $norm+ = val[idx] * val[idx]$
  end for
  for $(idx = rowptr[row]; idx < rowptr[row + 1]; idx++)$ do
    $x[col[idx]]+ = scale * val[idx]$ {Complex(d-z) Multiplication}
  end for
  $x[col[diag\_idx]]+ = scale * (-\sigma)$ {Complex(d-d) Multiplication}
end for
```

Bibliography

[1] AVX512 - Conflict Detection Instruction. https://software.intel.com/en-us/node/534475.

[2] COLPACK. http://cscapes.cs.purdue.edu/coloringpage/.

[3] CRS format. https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_row_.28CSR.2C_CRS_or_Yale_format.29.

[4] DLR. http://www.dlr.de/dlr//en/desktopdefault.aspx/tabid-10002/.

[5] Equipping Sparse Solvers for Exascale - ESSEX. https://blogs.fau.de/essex/activities.

[6] FEAST Eigenvalue Solver library. http://www.ecs.umass.edu/~polizzi/feast/.

[7] Wikipedia - HyperTransport. https://en.wikipedia.org/wiki/HyperTransport.

[8] Intel Extended Eigensolver. https://software.intel.com/en-us/articles/introduction-to-the-intel-mkl-extended-eigensolver.

[9] NP complete. https://en.wikipedia.org/wiki/NP-completeness.

[10] PARDISO library. http://www.pardiso-project.org/.

[11] Wikipedia - QPI link. https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect.

[12] RCM example. http://de.mathworks.com/help/matlab/ref/symrcm.html.

[13] Software for Exascale Computing - SPPEXA. http://www.sppexa.de/.

[14] SpMP. https://github.com/IntelLabs/SpMP/wiki.

[15] The SuiteSparse Matrix Collection. https://www.cise.ufl.edu/research/sparse/matrices/.

[16] English translation - Angenäherte Auflösung von Systemen linearer Gleichungen. http://jasonstockmann.com/Jason_Stockmann/Welcome_files/kaczmarz_english_translation_1937.pdf.

[17] Wikipedia, OpenMP. https://en.wikipedia.org/wiki/OpenMP.

[18] Mario Arioli, Iain Duff, Joseph Noailles, and Daniel Ruiz. A Block Projection Method for Sparse Matrices. *SIAM Journal on Scientific and Statistical Computing*, 13(1):47–70, 1992. doi: 10.1137/0913003. URL http://dx.doi.org/10.1137/0913003.

[19] Mario Arioli, Iain S. Duff, Daniel Ruiz, and Miloud Sadkane. Block Lanczos Techniques for Accelerating the Block Cimmino Method. *SIAM Journal on Scientific Computing*, 16(6):1478–1511, 1995. doi: 10.1137/0916086. URL http://dx.doi.org/10.1137/0916086.

[20] ASSEFAW H. GEBREMEDHIN, DUC NGUYEN, MD. MOSTOFA ALI PATWARY,ALEX POTHEN. ColPack: Software for Graph Coloring and Related Problems in Scientific Computing. *ACM Transactions on Mathematical Software*. https://www.cs.purdue.edu/homes/agebreme/publications/colpack-toms.pdf.

[21] A. Azad, M. Jacquelin, A. Buluc, and E. G. Ng. The Reverse Cuthill-McKee Algorithm in Distributed-Memory. *ArXiv e-prints*, October 2016. URL http://adsabs.harvard.edu/abs/2016arXiv161008128A.

[22] Blaise Barney. OpenMP. `https://computing.llnl.gov/tutorials/openMP/`.

[23] Åke Björck and Tommy Elfving. Accelerated projection methods for computing pseudoinverse solutions of systems of linear equations. *BIT Numerical Mathematics*, 19(2):145–163, 1979. ISSN 1572-9125. doi: 10.1007/BF01930845. URL `http://dx.doi.org/10.1007/BF01930845`.

[24] David Callahan, Ken Kennedy, and John Cocke. Estimating Interlock and Improving Balance for Pipelined Architectures. In *ICPP*, pages 295–304. Pennsylvania State University Press, 1987. URL `http://dblp.uni-trier.de/db/conf/icpp/icpp1987.html#CallahanKC87`.

[25] A. H. Castro Neto, F. Guinea, N. M. R. Peres, K. S. Novoselov, and A. K. Geim. The electronic properties of graphene. *Rev. Mod. Phys.*, 81:109–162, Jan 2009. doi: 10.1103/RevModPhys.81.109. URL `http://link.aps.org/doi/10.1103/RevModPhys.81.109`.

[26] Yutao Zhong Chen Ding. Reuse Distance Analysis. Technical report, University of Rochester, Computer Science Department, 02 2001. URL `http://www.cs.rochester.edu/u/cding/Documents/Publications/TR741.pdf`.

[27] Dan Gordon, Rachel Gordon. AN OVERVIEW OF THE CARP-CG ALGORITHM. *Parnum*. `http://cs.haifa.ac.il/~gordon/parnum.pdf`.

[28] Dan Gordon, Rachel Gordon. CARP-CG: a robust and efficient parallel solver for linear systems, applied to strongly convection dominated PDEs. *Parallel Computing*, 2010. `http://tx.technion.ac.il/~rgordon/carp-cg.pdf`.

[29] Doruk Bozdă, Umit Catalyurek,, Assefaw H. Gebremedhin, Fredrik Manne, Erik G. Boman, Füsun Ozguner. A Parallel Distance-2 Graph Coloring Algorithm for Distributed Memory Computers. *Springer*. `http://www.cs.sandia.gov/~egboman/papers/d2-coloring-hpcc.pdf`.

[30] L. A. Drummond, Iain S. Duff, Ronan Guivarch, Daniel Ruiz, and Mohamed Zenadi. Partitioning strategies for the block Cimmino algorithm. *Journal of Engineering Mathematics*, 93(1):21–39, 2015. URL `"http://dx.doi.org/10.1007/s10665-014-9699-0"`.

[31] Tommy Elfving. Block-iterative methods for consistent and inconsistent linear equations. *Numerische Mathematik*, 35(1):1–12, 1980. ISSN 0945-3245. doi: 10.1007/BF01396365. URL `http://dx.doi.org/10.1007/BF01396365`.

[32] Martin Galgon, Lukas Krämer, Jonas Thies, Achim Basermann, and Bruno Lang. On the parallel iterative solution of linear systems arising in the FEAST algorithm for computing inner eigenvalues. *Parallel Computing*, 49:153 – 163, 2015. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/j.parco.2015.06.005. URL `http://www.sciencedirect.com/science/article/pii/S0167819115000915`.

[33] Dan Gordon and Rachel Gordon. Component-Averaged Row Projections: A Robust, Block-Parallel Scheme for Sparse Linear Systems. *SIAM Journal on Scientific Computing*, 27(3):1092–1117, 2005. doi: 10.1137/040609458. URL `http://dx.doi.org/10.1137/040609458`.

[34] Dan Gordon and Rachel Gordon. CGMN revisited: robust and efficient solution of stiff linear systems derived from elliptic partial differential equations. *ACM Trans. on Mathematical Software*, 2008.

[35] Dan Gordon and Rachel Gordon. Robust and highly scalable parallel solution of the Helmholtz equation with large wave numbers . *Journal of Computational and Applied Mathematics*, 237 (1):182 – 196, 2013. ISSN 0377-0427. doi: http://dx.doi.org/10.1016/j.cam.2012.07.024. URL `http://www.sciencedirect.com/science/article/pii/S0377042712003147`.

[36] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010. ISBN 143981192X, 9781439811924.

[37] M. Z. Hasan and C. L. Kane. *Colloquium* : Topological insulators. *Rev. Mod. Phys.*, 82:3045–3067, Nov 2010. doi: 10.1103/RevModPhys.82.3045. URL `http://link.aps.org/doi/10.1103/RevModPhys.82.3045`.

[38] Roger W. Hockney and I. J. Curington. f1/2: a parameter to characterize memory and communication bottlenecks. *Parallel Computing*, 10(3):277–286, 1989. URL http://dblp.uni-trier.de/db/journals/pc/pc10.html#HockneyC89.

[39] J. A. Scott J. K. Reid. Reducing the total bandwidth of a sparse unsymmetric matrix. Technical report, Council for the Central Laboratory of the Research Councils, 02 2005. http://www.numerical.rl.ac.uk/reports/rsRAL2005001.pdf.

[40] S. Kaczmarz. Angenäherte Auflösung von Systemen linearer Gleichungen. *Bulletin International de l'Académie Polonaise des Sciences et des Lettres*, 35:355–357, 1937.

[41] Chandrika Kamath and Ahmed Sameh. A projection method for solving nonsymmetric linear systems on multiprocessors. *Parallel Computing*, 9(3):291 – 312, 1989. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/0167-8191(89)90114-2. URL http://www.sciencedirect.com/science/article/pii/0167819189901142.

[42] James Kestyn, Vasileios Kalantzis, Eric Polizzi, and Yousef Saad. PFEAST: A High Performance Sparse Eigenvalue Solver Using Distributed-memory Linear Solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 16:1–16:12, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-4673-8815-3. URL http://dl.acm.org/citation.cfm?id=3014904.3014926.

[43] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014. doi: 10.1137/130930352. URL http://dx.doi.org/10.1137/130930352.

[44] Moritz Kreutzer, Jonas Thies, Melven Röhrig-Zöllner, Andreas Pieper, Faisal Shahzad, Martin Galgon, Achim Basermann, Holger Fehske, Georg Hager, and Gerhard Wellein. GHOST: Building Blocks for High Performance Sparse Linear Algebra on Heterogeneous Systems. *International Journal of Parallel Programming*, pages 1–27, 2016. ISSN 1573-7640. doi: 10.1007/s10766-016-0464-z. URL http://dx.doi.org/10.1007/s10766-016-0464-z.

[45] J. Liu, S. J. Wright, and S. Sridhar. An Asynchronous Parallel Randomized Kaczmarz Algorithm. *ArXiv e-prints*, January 2014.

[46] J.C. Luo. Algorithms for reducing the bandwidth and profile of a sparse matrix. *Computers and Structures*, 44(3):535–548, 1992. ISSN 0045-7949. doi: http://dx.doi.org/10.1016/0045-7949(92)90386-E. URL http://www.sciencedirect.com/science/article/pii/004579499290386E.

[47] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. URL http://www.cs.virginia.edu/stream/. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[48] Gordon E. Moore. Readings in Computer Architecture. chapter Cramming More Components Onto Integrated Circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-539-8. URL http://dl.acm.org/citation.cfm?id=333067.333074.

[49] D. Needell and J. A. Tropp. Paved with Good Intentions: Analysis of a Randomized Block Kaczmarz Method. *ArXiv e-prints*, August 2012.

[50] Paul K. Stockmeyer Norman E. Gibbs, William G. Poole. An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, 1976. ISSN 00361429. URL http://www.jstor.org/stable/2156090.

[51] Peter Knabner, Lutz Angermann. *Numerical Methods for Elliptic and Parabolic Partial Differential Equations*. Springer, 2003.

[52] Peter Oswald, Weiqi Zhou. Convergence Estimates for Kaczmarz-Type Methods. http://www.faculty.jacobs-university.de/poswald/KaczmarzPaperNew.pdf.

[53] Andreas Pieper, Moritz Kreutzer, Andreas Alvermann, Martin Galgon, Holger Fehske, Georg Hager, Bruno Lang, and Gerhard Wellein. High-performance implementation of Chebyshev filter diagonalization for interior eigenvalue computations . *Journal of Computational Physics*, 325:226 – 243, 2016. ISSN 0021-9991. doi: http://dx.doi.org/10.1016/j.jcp.2016.08.027. URL `http://www.sciencedirect.com/science/article/pii/S0021999116303837`.

[54] Eric Polizzi. A Density Matrix-based Algorithm for Solving Eigenvalue Problems. *CoRR*, abs/0901.2665, 2009. URL `http://arxiv.org/abs/0901.2665`.

[55] Thomas Röhl, Jan Eitzinger, Georg Hager, and Gerhard Wellein. *Validation of Hardware Events for Successful Performance Pattern Identification in High Performance Computing*, pages 17–28. Springer International Publishing, Cham, 2016. ISBN 978-3-319-39589-0. doi: 10.1007/978-3-319-39589-0_2. URL `http://dx.doi.org/10.1007/978-3-319-39589-0_2`.

[56] Melven Röhrig-Zöllner. Parallel solution of large sparse eigenproblems using a Block-Jacobi-Davidson method. Master's thesis, Rheinisch-Westfälischen Technischen Hochschule Aachen, Germany, 2014.

[57] Melven Röhrig-Zöllner, Jonas Thies, Moritz Kreutzer, Andreas Alvermann, Andreas Pieper, Achim Basermann, Georg Hager, Gerhard Wellein, and Holger Fehske. Increasing the Performance of the Jacobi–Davidson Method by Blocking. *SIAM Journal on Scientific Computing*, 37(6):C697–C722, 2015. doi: 10.1137/140976017. URL `http://dx.doi.org/10.1137/140976017`.

[58] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 207–216, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3559-1. doi: 10.1145/2751205.2751240. URL `http://doi.acm.org/10.1145/2751205.2751240`.

[59] G. Stewart. *Matrix Algorithms*. Society for Industrial and Applied Mathematics, 2001. doi: 10.1137/1.9780898718058. URL `http://epubs.siam.org/doi/abs/10.1137/1.9780898718058`.

[60] Thomas Strohmer and Roman Vershynin. A Randomized Kaczmarz Algorithm with Exponential Convergence. *Journal of Fourier Analysis and Applications*, 15(2):262, 2008. ISSN 1531-5851. doi: 10.1007/s00041-008-9030-4. URL `http://dx.doi.org/10.1007/s00041-008-9030-4`.

[61] P. T. P. Tang and E. Polizzi. FEAST as a Subspace Iteration Eigensolver Accelerated by Approximate Spectral Projection. *ArXiv e-prints*, February 2013.

[62] Jonas Thies, Martin Galgon, Faisal Shahzad, Andreas Alvermann, Moritz Kreutzer, Andreas Pieper, Melven Röhrig-Zöllner, Achim Basermann, Holger Fehske, Georg Hager, Bruno Lang, and Gerhard Wellein. *Towards an Exascale Enabled Sparse Solver Repository*, pages 295–316. Springer International Publishing, Cham, 2016. ISBN 978-3-319-40528-5. doi: 10.1007/978-3-319-40528-5_13. URL `http://dx.doi.org/10.1007/978-3-319-40528-5_13`.

[63] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. 2010.

[64] Gerhard Wellein. Basic Buidling Blocks for Sparse Iterative Solvers: Trends to Exascale. 22nd Workshop Talk, DFG project ESSEX.

[65] Gerhard Wellein, Georg Hager, and Markus Wittmann. Lecture notes on Programming Technique for Supercomputers. 2015. URL `http://univis.fau.de/form?__s=2&dsc=anew/module_view&mod=tech/IMMD/phleir/5&sem=2016s&anonymous=1&lvs=tech/IMMD/phleir/ptfs&ref=main&__e=136`.

[66] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL `http://doi.acm.org/10.1145/1498765.1498785`.

[67] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, 2003.