UNIVERSITY OF BERGEN

Improving Parallel Sparse Matrix-vector Multiplication

by

Torbjørn Tessem

Thesis for the degree Master of Science

December 2013

in the

Faculty of Mathematics and Natural Sciences

Department of Informatics

Acknowledgements

- I will thank my supervisor Fredrik Manne for discussions and support while writing this thesis.
- I would also like to thank my fellow students, especially Alexander Sørnes, for relevant and irrelevant discussions, making long days at the university enjoyable.
- Last but not least, I will thank my parents and family for their support throughout my studies.

Contents

Acknowledgements **iii**

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

2 Fundamentals **5**

2.1 Parallel programming . . . 6

2.1.1 Speed-up and scalability . . . 7

2.1.2 OpenMP . . . 8

2.2 Matrix representations and Sparse Matrix-vector Multiplication . . . 9

2.2.1 Coordinate representation (COO) . . . 10

2.2.2 Compressed row (CRS) and Compressed column (CCS) . . . 11

2.3 Parallel SMvM . . . 13

3 Challenges in Parallel Sparse Matrix-vector multiplication **15**

3.1 Conflicts . . . 16

3.1.1 False sharing . . . 16

3.1.2 Write conflicts . . . 16

3.2 Even work load . . . 20

3.2.1 OpenMP scheduling techniques . . . 20

3.2.2 Distribute on number of non-zero elements . . . 21

3.3 Cache efficiency . . . 21

3.4 Unfortunate structured matrices . . . 22

4 Previous Work **23**

4.1 Efficient cache use . . . 24

4.1.1 Enlarge dense blocks . . . 24

4.1.2 Restructuring and Reordering to reduce cache miss . . . 24

4.2 Avoiding write conflicts with CCS-algorithms . . . 26

4.2.1 Basic solutions . . . 26

4.2.2 CCS with colouring . . . 26

4.3 Colouring algorithms . . . 29

4.3.1 Greedy Graph Colouring . . . 30

4.3.2 Greedy distance-2 colouring . . . 30

5 Even Work Load 33
5.1 Improved distribution in CRS-algorithms 34
5.1.1 Approximation to *min-makespan* 34
5.1.2 Allocating consecutive data to threads 34
5.2 Arrowhead matrices - Can COO-representation be the fastest? 35
5.2.1 Distribution of elements 36

6 Efficient Cache Use 39
6.1 A hybrid between CRS and CCS representations 39
6.2 COO-representation with CRS-CSS-hybrid iteration 40

7 Improving CCS with colouring 43
7.1 Reducing False Sharing 43
7.1.1 Increased distance 44
7.2 Efficient parallel sections 44
7.2.1 When is the greedy colouring bad? 45
7.2.1.1 Identifying inefficient parallel steps and heavy tails 46
7.2.2 How to handle an inefficient step. 46
7.2.2.1 Basic CCS-algorithms 47
7.2.3 Perfect parallel steps 47
7.2.3.1 Random colouring 47

8 Experiments 49
8.1 Chp. 5 - Even work load 50
8.1.1 Improved distribution on CRS-algorithms 50
8.1.2 Arrowhead Matrices 52
8.2 Chp. 6 - Efficient cache use 53
8.3 Chp. 7 - Improving CCS with colouring 55
8.3.1 Replicating results 55
8.3.1.1 AP12 - Algorithms 55
8.3.2 Reducing false sharing 56
8.3.3 Efficient parallel sections 57
8.3.4 Decreased efficiency on the tail 57
8.3.4.1 Alternative handling of the tail 58
8.3.4.2 Random colouring algorithms 61

9 Conclusions 63
9.1 Even work load matters 63
9.2 The hybrid algorithm is only better on certain matrices 64
9.3 Increasing distance to reduce false sharing is not a good idea 64
9.4 It is hard to handle inefficient parallel steps 65
9.5 Future Work 65
9.5.1 Approximation algorithms on min-makespan 65
9.5.2 A hybrid algorithm using less storage 66
9.5.3 Faster and better colouring algorithms 66

Bibliography **67**

List of Figures

2.1 Illustration of a parallel program . . . 5
2.2 Matrix M, of size $4 \cdot 6$, with 8 non zero elements. . . . 9
2.3 COO representation of M (Figure 2.2) . . . 10
2.4 CCS representation of M (Figure 2.2) . . . 12
2.5 CRS representation of M (Figure 2.2) . . . 12

3.1 Two threads reading from same cache-line, leading to false sharing. . . . 15
3.2 Illustration of a write conflict . . . 17
3.3 Arrowhead matrix of size 7×7 . . . 22

4.1 Illustration of SBD matrix . . . 25
4.2 Distance-2 colouring on a bipartite graph. . . . 29

5.1 Matrix, *nlpkkt_240* . . . 33
5.2 An arrowhead matrix, *delaunay_n24* . . . 36

6.1 Illustration of the parallel hybrid SMvM algorithm . . . 40

7.1 Use of colours by COL1 on matrix HV15R . . . 45

8.1 Scaling of the hybrid algorithm on different matrices. . . . 54
8.2 Scaling on the good part and on the tail, with CCS with colouring, for *HV15R* . . . 58
8.3 Scaling on the tail, with CCS with colouring, when using different barriers on HV15R . . . 59
8.4 Run times on different solutions on the tail of HV15R . . . 60
8.5 Results on Hook1498 with greedy and random colouring . . . 62
8.6 Results on HV15R with greedy and random colouring . . . 62

List of Tables

4.1 Matrices used in AP12 28

5.1 Difference in number of elements between lightest and heaviest thread on *nlpkkt_240* 35
5.2 Difference in number of elements between lightest and heaviest thread on *delaunay_24* 37

7.1 Table that shows % of columns in the tail for different barriers. 46

8.1 Overview of the matrices used in the experiments. 50
8.2 Selected results on the CRS-algorithm with different distribution schemes. 51
8.3 Results on arrowhead matrices 52
8.4 Selected results with the hybrid algorithm 53
8.5 Replicated results from AP12 56
8.6 Selected results from CCS-col with increased distance. 57
8.7 Selected results showing speed-up on different part of the matrices 58
8.8 Selected results for alternative handling of the tail and for random colouring. 60
8.9 $\frac{n}{numColour}$ for the six matrices coloured with a random colouring 62

Chapter 1

Introduction

When solving a problem, a computer program will do a number of specified operations. Some of these have to be done in a strict order, i.e. the result of one is needed in the next. However, in many cases operations can be done independently. This introduces the idea of parallel programming.

In parallel programming [1] the problem is divided to several interconnected processors or threads who collaborate in solving the problem. This exploits that operations that are independent of each other, can be done at the same time. The intention is obviously to solve the problem faster. This is the main argument for using parallel programming. With faster programs we can solve computationally intensive problems in a reasonable time. Using more processors also gives access to more memory, thus allowing for larger data sets.

Sparse Matrix-vector Multiplication (SMvM) is a mathematical technique encountered in many programs and computations and is often heavily used. Examples included iterative methods [13], used to solve systems of linear equations, and computer graphics [15], where it is used to manipulate information on objects in space. In SMvM, values from an input vector is multiplied with values from a matrix and added into an output vector. Many of these operations can be done independently, pointing to the possibility of doing this in parallel. SMvM is a special case of general Matrix-vector Multiplication, where the matrix is sparse. A sparse matrix is defined as a matrix where the majority of the elements are equal to $zero$ or or the structure of the matrix allows us to otherwise take advantage of the $zero - elements$ [9]. SMvM on large matrices is used in many real world applications [9], thus a substantial amount of work has been done to improve parallel SMvM algorithms.

Several strategies have been tried to improve parallel SMvM. Representing the matrix in a way that takes advantage of the sparse structure is often the first step. This is typically done by only storing the non-zero elements, thus reducing the space needed to store the matrix. We still maintain information about the position of the elements by storing their coordinates (COO) or arranging them into compressed rows (CRS), columns (CCS) or blocks [12]. Standard parallel SMvM algorithms are typically designed using one of these representations, each with different positive and negative properties. This is discussed further in Chapter 2.

Further improvements include work to access the data more efficiently. The data is accessed through cache lines that each hold several elements. Loading a cache-line from main memory through the cache hierarchy has a cost, so using cache efficiently will most likely give faster programs. How to improve the efficiency of cache use is discussed in amongst others Pinar and Heath [5], Vuduc and Moon [6] and Yzelman and Bisseling [4]. Typically this is done by reordering the iterations or restructuring the matrix. Yzelman discusses this further in his thesis [3].

Other approaches to improve parallel SMvM have also been investigated. Bell and Garland [16], utilized the computational resources of the Graphic Processing Units (GPU). In Azad and Pothen [7], graph colouring is used on the matrices to avoid write conflicts when using CCS representation. These occur when two (or more) threads try to write to the same memory slot at the same time. Azad and Pothen try to reach results similar to an algorithm using CRS representation, but their research does not quite achieve this. Previous work is discussed further in Chapter 4.

The aim of the work conducted in this thesis is to develop new ideas and algorithms to speed-up parallel SMvM on a shared memory computer. In chapters 5-7 we present our main ideas. These are partly based on some of the previous done work by others and also on specific problems concerning parallel SMvM. Our first suggestion is to give a better distribution of data to the threads. To do this we use a method inspired by the min-makespan problem [1, 2]. Further we investigate distributing elements more evenly, using the COO-representation, without regard to other problems that might occur. Our test results shows some improvements compared to the standard algorithms.

We also suggest an improvement to obtain more efficient cache use by altering the iteration of the standard algorithms. This is inspired by an algorithm presented in Yzelman and Bisseling and observations on the difference in cache use of the standard parallel SMvM algorithms using CRS- and CCS-representations. The results show improvements on matrices with a structure requiring the standard algorithms to often load new cache lines. These results are similar to those presented in Yzelman and Biesseling.

In Chapter 7 we suggest possible improvements to the algorithms presented in Azad and Pothen. Their results shows CCS representation to be inferior to CRS representation. However this may not always hold. Dimensions and structure of matrices may favour CCS representation. Our suggestions for improvements include alterations to reduce false sharing. Furthermore, we tried to step away from using the colouring where we found it to create inefficient parallel steps. We also introduced using a random colouring algorithm to remove the inefficient parallel steps. The observed results to the possible improvements on the work presented in Azad and Pothen varied. The alteration of the colouring algorithm to reduce false sharing gave negative results. However, the work to handle inefficient steps alternatively gave some improvements.

In this thesis we address the research questions:

- How will a more even distribution of data affect parallel SMvM? Both with regard to improving distribution for known algorithms, as well as creating new algorithms. Furthermore, will these improvements work for general matrices or only for specific ones.
- Can we, by changing the order the elements are accessed, improve the efficiency of cache use? Even though we have to use more memory to store the matrix?
- Colouring of the columns to avoid write conflicts has already been investigated in Azad and Pothen [7]. We consider which colouring algorithms yields the best colourings for this problem. Furthermore, is using the colouring always best, and how can we identify when it is?

The remaining thesis consists of eight chapters (cp.2 - cp.9) and can be divided into four parts.

1. Chapter 2 introduces some fundamental concepts in regards to parallel programming, representing sparse matrices, and parallel Sparse Matrix-vector Multiplication. Chapter 3 presents some of the most important problems concerning implementation of parallel SMvM. Furthermore, some ideas on how to handle these problems are discussed.
2. Chapter 4 gives an overview of some of the previous work done to improve SMvM algorithms. We present results and conclusions from a selected set of papers that use different techniques to improve SMvM algorithms. One of these techniques is graph colouring and we therefore present some graph colouring algorithms.

3. Chapters 5, 6, and 7 presents some ideas on possible improvements on parallel SMvM algorithms. Theses ideas are based on both the problems presented in Part 1 as well as the work presented in Part 2. Each chapter revolves around one specific theme that addresses different issues concerning parallel SMvM.

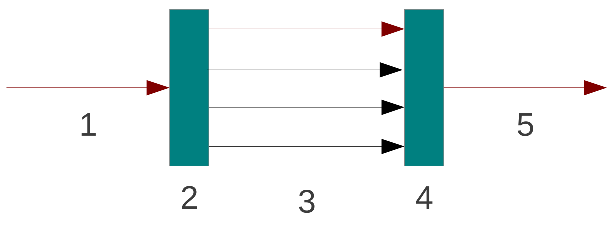
4. Chapter 8 presents test results and compares them to what we could expect from the ideas presented in chapters 5-7. Chapter 9 gives some conclusions based on the observed results and other previous work. Furthermore, we present some topics that may be studied further.

Chapter 2

Fundamentals

This chapter presents some fundamental aspects of the thesis. It introduces parallel programming in general and especially shared memory parallel programming. OpenMP, an API for Shared Memory Parallel Programming is discussed, with focus on the elements used in the thesis.

It also presents different ways of representing sparse matrices. For the different matrix-representations, basic matrix-vector multiplication algorithms are shown. It is also briefly discussed how to implement these algorithms in parallel.



A parallel program can work in the following way:

1. *Sequential part.*
2. *Divide work, and create new threads.*
3. *Parallel part (with four threads).*
4. *Collect results and terminate extra threads.*
5. *Sequential part.*

FIGURE 2.1: Illustration of a parallel program

2.1 Parallel programming

Solving computational problems using multiple tightly connected processors is known as parallel computing [1]. The positive effects of parallel computing are obvious. Using several processors allows problems to be divided on several threads and thus solved faster. In some instances parallel computing also gives access to more memory, giving the possibility to solve larger problems. The variable p usually represent the number of processors. An illustration of a parallel program, with $p = 4$, is shown in Figure 2.1.

However, parallel computing also offer some challenges. In addition to solving the problem at hand, the extra tasks of dividing and collecting the work are added. Furthermore, there can be an extra cost in creating new threads. This means that if the total work load is small, a parallel step might not be justifiable. The different processors also need to communicate, so that they are synchronized and the problems are solved correctly. In addition to this, the execution of a parallel program can be non-deterministic, thus making them hard to debug.

When a thread needs to access a memory slot, most systems lets the thread load a whole cache line into its local memory. A cache line is a part of memory consisting of several (depending on the system) memory slots. This is done to make the programs more efficient, as consecutive elements needed often are stored in memory slots close to each other. However, this can create problems for some parallel programs, as two threads might need different elements on the same cache line. Thus trying to load and update the same cache line. This is called false sharing.

In Shared memory parallel programming (SMPP) the memory is visible to all processors. This allows for both thread-private and shared variables. Communication between the threads can be done through the shared memory. The shared memory also allows the threads to work on a common, shared, solution, thus a separate collecting step may not be needed. However, if data is shared unintentionally, race conditions may occur. An example of this is write conflicts, where threads write to the same memory slot at the same time, possibly overwriting each each other.

Parallel programming can also be done with distributed memory (DMPP). In this case, each thread holds its own private storage. Thus, each thread can work without having to take regard to the other threads, and deal with issues like write conflicts and false sharing. However, to communicate, threads have to send and receive data chunks to and from each other, giving an extra cost. A DMPP program is usually made of alternating computation and communication steps, giving total time use $t_{tot} = t_{comp} + t_{comm}$. To achieve good results it is important to keep t_{comm} lower than t_{comp}. A good example where DMPP is used efficiently is matrix-multiplication with Cannon's Algorithm [1, 18].

Both SMPP and DMPP have their advantages, and are useful for different problems. In this thesis we will work with parallel SMvM, where SMPP is the preferred choice.

2.1.1 Speed-up and scalability

To justify a parallel program, it must show *speed-up* compared to the corresponding sequential program [1]. The *speed-up factor*, $\mathrm{S}(p)$, is defined as how the time used by a parallel program t_p, compares to the time from (the best) sequential one, t_s, giving the formula: $\mathrm{S}(p) = \frac{t_s}{t_p}$.
The maximum possible speed-up is usually bounded to p. This is called *linear* speed-up and means that $\frac{t_s}{p} = t_p$. It might happen that $\mathrm{S}(p) > p$. We call this *Superlinear* speed-up. Superlinear speed-up can happen hen the architecture or algorithm favours a parallel program, for instance if cache is handled more efficiently in parallel. However, it can also occur when comparing to a non-optimal sequential program.

The speed-up is also restricted by overhead in the parallel program. This can for instance be communication costs or a distribution of work that leaves threads idle. Parallel programs usually also have a sequential part. Obviously the sequential part has no speed-up, thus the speed-up is restricted to the fraction of the program that can be executed in parallel, denoted by f. Even with an infinite number of threads the speed-up cannot be higher than $\lim_{p\to\infty}\mathrm{S}(p) = \frac{1}{f}$. This was used as an argument for sequential computation by Amdahl in 1967 [14], and is known as Amdahl's law. The *efficiency* of a parallel program is measured as $\frac{t_s}{t_p \times p}$, i.e. how much resources the parallel program uses compared to its corresponding sequential program.

Scalability is defined in how a program handles different system and data sizes. Generally a larger system, i.e. using more processors, is better than a smaller one. However, more processors also requires, amongst other things, more communication, making the program less efficient. Measuring how a program reacts when applying more threads on the same problem size is called *strong* scaling. This is opposed to *weak* scaling, where one investigated how the program reacts when using both the work and the number of threads grows proportionally. Thus keeping the amount of work per processor constant.

2.1.2 OpenMP

OpenMP [10] [11] is an API used for SMPP, that works with C/C++ and Fortran. The functions of OpenMP are created by constructs. Most constructs are compiler directives, and usually apply to a structured block of code. In C/C++ constructs are written on the form *#pragma omp* $<$ *construct* $>$ [$<$ *clause* $>$].

Programs using OpenMP operate in a fork-join structure, where a master thread spawns new threads to create parallel sections, and joins them once the parallel section is over. To create a parallel section the *parallel* construct is used. The number of threads it creates is either decided by a runtime function or a clause. A parallel section in OpenMP usually works with any amount of threads. Thus a programs using OpenMP can function both sequentially and in parallel.

OpenMP also has a parallel for-loop construct that can be used either within parallel sections or to create a new parallel section. A parallel loop divides the iteration of the loop among the different threads. When executing a loop in parallel, it is important that the iterations of the loop can be executed independently of each other. This is because a parallel loop may execute the iterations in a non deterministically order. A clause on the parallel loop construct decides the parallel scheduling of the iterations. This can have a huge influence on the performance of the program. The types of scheduling options for the parallel for loop are:

- *Static*, This is the default setting. It allocates an (almost) equal sized, consecutive, chunk of the iterations from loop to each thread.
- *Dynamic*, The loop is divided into equally sized chunks and which are put into a queue. Each thread will take a chunk of iterations from the queue and execute When it is finished with this it goes back to the queue and gets the next available chunk. This continues until all the work is done.
- *Guided*, This partitions the iterations of the loop into chunks of decreasing size. These are then put into a queue, but with the largest chunks first. The distribution of chunks to threads is done in the same way as for the dynamic scheduling.

In nested loops, deciding which loop to parallelize is also important. By doing the outer one in parallel, only one parallel section is created. Thus each thread runs the inner loop individually, in parallel. In such an instance it is important that the inner loop can be executed simultaneously by multiple threads without causing unwanted side effects. If one parallelizes the inner loop, a new parallel step is created for each iteration. Thus, each instance of the inner loop, is done as a normal parallel iteration.

$$\begin{bmatrix} 1 & 0 & 2 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 & 0 \\ 0 & 0 & 6 & 0 & 0 & 7 \\ 0 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

FIGURE 2.2: Matrix M, of size $4 \cdot 6$, with 8 non zero elements.

The threads can communicate with each other through the shared variables. there are several constructs for synchronization. The *critical* construct makes sure that the following block is only executed by one thread at a time. The *atomic* construct is similar, but it only applies to the update of the memory slot accessed in the next line of code. The *barrier* construct forces all the threads to wait until the last one has reached it.

2.2 Matrix representations and Sparse Matrix-vector Multiplication

The main focus of this thesis is on parallel Sparse Matrix-Vector Multiplication (SMvM). SMvM is a variant of matrix-vector multiplication where the matrix is *sparse*. A sparse matrix is defined as a matrix, of size $n \times m$, that has nnz non-zero elements, where $n \times m \gg nnz$. This fact can be used to store the matrix more efficiently.

The standard method of representing a matrix is using a two dimensional array, where each position i, j in the array holds a value $v_{i,j}$. This representation is intuitive and easy to work with. Matrix-vector multiplication, shown in ALG 1, is given a matrix M and an input vector, the result is stored in an output vector. Matrix-vector multiplication is done by iterating over the matrix, for every element in the matrix the value $v_{i,j}$ is multiplied with the value on index i from the input vector and added onto the value on index j on the output vector.

Standard matrix representation uses space to represent every value, so for a matrix of size $n \times m$, it will use $O(n \times m)$ space, regardless if many elements are equal or uninteresting with respect to the problem at hand. For sparse matrices a large majority of the elements are zero and for many problems, including Matrix-Vector Multiplication, the zero elements in a matrix are uninteresting, so to store them is a waste of memory. Figure 2.2 shows an example matrix M using the standard representation.

Algorithm 1: Matrix-vector multiplication

```
Data: $M$ a matrix of size $m \times n$, $v$ an input vector of length $n$
Result: $w$, an output vector of length $m$

 set all indices in $w$ to 0
 for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $m$ do
       $w[j] \leftarrow w[j] + M[i, j] \times v[i]$
    end for
 end for
```

$$\begin{bmatrix} 1, & 2, & 3, & 4, & 5, & 6, & 7, & 8 \\ 0, & 0, & 0, & 1, & 1, & 2, & 2, & 3 \\ 0, & 2, & 4, & 1, & 4, & 2, & 5, & 3 \end{bmatrix}$$

FIGURE 2.3: COO representation of M (Figure 2.2)
The 1st row represent non-zero elements, the 2nd and 3rd rows represent coordinates.

2.2.1 Coordinate representation (COO)

A more efficient way to represent sparse matrices is the Coordinate Representation (COO) or triplet scheme [3, 17]. This representation only stores the non-zero elements. The original matrix is represented by a $3 \times nnz$ matrix (nnz is the number of non-zero elements), where one row holds the value of the elements and the two others hold the corresponding coordinates. An example using the COO-representation is shown in Figure 2.3, where the second row holds row coordinates and the third holds the column coordinates. The space used here is only dependent on nnz, giving a total space use of $O(nnz)$, which for sparse matrices is (by definition) much smaller than $n \times m$.

The COO-representation does not give the easy look-up of the standard matrix, but for some problems this is not important. Algorithms with COO-representation can iterate over the nnz elements, and for each element it can do the needed work. Since the element stores its coordinates independently, the elements are easy to sort, restructure and allocate to a thread.

Sparse Matrix-vector multiplication (SMvM) using the COO-representation (ALG 2) is done by iterating over the nnz non-zero elements. The value found on position $M[i, 0]$ is multiplied with the value from the input vector at the position given by $M[i, 2]$ and added onto the output vector on the position given by $M[i, 1]$.

Algorithm 2: SMvM algorithm using COO-representation

```
Data: $M$ a sparse matrix size $m \times n$ stored in COO format, $v$ an input vector of
      length $n$
Result: $w$, an output vector of length $m$

  set all indecies in $w$ to 0
  for $i \leftarrow 1$ to $nnz$ do
    $w[M[i,1]] \leftarrow w[M[i,1]] + M[i,0] \times v[M[i,2]]$
  end for
```

2.2.2 Compressed row (CRS) and Compressed column (CCS)

Two even less space demanding representations of a sparse matrix are Compressed Row Storage (CRS) and Compressed Column Storage (CCS) [3, 12]. Similar to COO, only the non-zero elements are stored, still using $O(nnz)$ space. However CCS and CRS reduces the space used to store the coordinates. By sorting the elements in the COO-representation on either row (CRS) or column indices (CCS), one of the coordinate vectors can be reduced to size n or m, respectively. This gives a total space usage of $2 \times nnz + m$ (or n).

In the CCS representation, the numerical values are stored in an array A, sorted on which columns the elements belong to. An array, JA, that holds the corresponding row for each element. These are both of length nnz. The third array IA, is only of size $m+1$. It contains information specifying at what positions in A and JA each column is located. Elements belonging to column i starts at position $IA[i]$ and ends at the position before $IA[i+1]$. Because all elements must belong to a column, we know that $IA[0] = 0$ and $IA[m] = nnz$. An example using CCS representation is shown in Figure 2.4. For CRS, the numbers are sorted by rows, A still holds the elements, JA the corresponding column, and IA holds the indices where each row starts and stops. An example using CCS representation is shown in Figure 2.5.

For most instances CRS and CCS representations will use less space than the COO representation. In fact, the size of n or m can be so small compared to nnz that a matrix using CCS or CRS representation may only use about 2/3 of the space it would have used with a COO representation. The CRS and CCS representations are also usually more efficient in terms of execution speed since more data movements will also affect the amount of time used [3].

The SMvM algorithms (ALG 3, ALG 4) using CRS or CCS-representation iterates over either the rows or the columns. For each row or column, the algorithms iterates over the elements of that row or column and multiplies it with the corresponding value from

$$\begin{aligned}
\textbf{A:}\ & \begin{pmatrix}1, & 4, & 2, & 6, & 8, & 3, & 5, & 7\end{pmatrix}\\
\textbf{JA :}\ & \begin{pmatrix}0, & 1, & 0, & 2, & 3, & 0, & 1, & 2\end{pmatrix}\\
\textbf{IA :}\ & \begin{pmatrix}0, & 1, & 2, & 4, & 5, & 7, & 9\end{pmatrix}
\end{aligned}$$

FIGURE 2.4: CCS representation of M (Figure 2.2)

$$\begin{aligned}
\textbf{A:}\ & \begin{pmatrix}1, & 2, & 3, & 4, & 5, & 6, & 7, & 8\end{pmatrix}\\
\textbf{JA :}\ & \begin{pmatrix}0, & 2, & 4, & 1, & 4, & 2, & 5, & 3\end{pmatrix}\\
\textbf{IA :}\ & \begin{pmatrix}0, & 3, & 5, & 7, & 8\end{pmatrix}
\end{aligned}$$

FIGURE 2.5: CRS representation of M (Figure 2.2)

the input vector and adds the result into its correct place in the output vector.

```
Algorithm 3: SMvM algorithm using CRS-representation
Data: $M$ a sparse matrix of size $m \times n$ stored in CRS format, $v$ an input vector of
        length $n$
Result: $w$ the result vector of length $m$

  set all indices in $w$ to 0
  for $i \leftarrow 1$ to $m$ do
    for $j \leftarrow IA[i]$ to $IA[i+1]$ do
      $w[i] \leftarrow w[i] + A[j] \times v[JA[j]]$
    end for
  end for
```

```
Algorithm 4: SMvM algorithm using CCS-representation
Data: $M$ a sparse matrix of size $m \times n$ stored in CCS format, $v$ an input vector of
        length $n$
Result: $w$ the result vector of length $m$

  set all indices in $w$ to 0
  for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow IA[i]$ to $IA[i+1]$ do
      $w[JA[j]] \leftarrow w[JA[j]] + A[j] \times v[i]$
    end for
  end for
```

2.3 Parallel SMvM

Parallel SMvM algorithms, using SMPP, have to divide the work between the threads. For most implementations this is done with a parallel loop. When using CRS–representation, a parallel algorithm can be constructed by simply doing the iteration over the rows in parallel, as shown in ALG 5. Since the result for each row is stored into a unique position of the output vector there will be no write conflicts.

For CCS- and COO representations however, parallelization is not so easy. Simply iterating over the columns (CCS) or over the elements (COO) in parallel can lead to problems. Since several threads might want to write into the same location, write conflicts may occur. How to handle write conflicts and other challenges for parallel SMvM algorithms are covered in the next chapter.

Algorithm 5: Parallel SMvM using CRS-representation

```
Data: $M$ a sparse matrix of size $m \times n$ stored in CRS format, $v$ an input vector of
        length $n$
Result: $w$ the result vector of length $m$

  set all indices in $w$ to 0
  for $i \leftarrow 1$ to $m$ do in parallel
    $t \leftarrow 0$ //local temporary variable $t$
    for $j \leftarrow IA[i]$ to $IA[i+1]$ do
      $t \leftarrow t + A[j] \times v[JA[\cancel{i}]]$ j
    end for
    $w[i] \leftarrow t$
  end for
```

Chapter 3

Challenges in Parallel Sparse Matrix-vector multiplication

There are a number of challenges that must be solved when designing efficient parallel SMvM algorithms. Not doing so may lead to inefficient computations or even errors. Some of these are specific to parallel algorithms, while others are also relevant to sequential SMvM. In this chapter, some of the most important challenges for parallel SMvM are presented. Furthermore, we discuss different solutions and some of the consequences of using these.

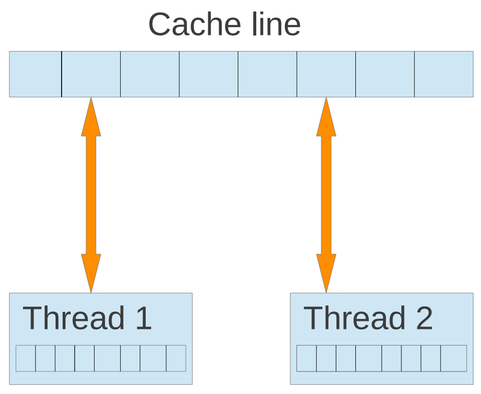


Figure 3.1: Two threads reading from same cache-line, leading to false sharing.

3.1 Conflicts

In shared memory parallel programs, different threads will from time to time try to access the same memory simultaneously. This can lead to conflicts between the threads. For most matrix representations, a parallel SMvM algorithm divides the elements of the matrix strictly between the threads, giving no (apparent) conflicts on the matrix itself. However, depending on the representation, threads will access the same memory slots on either the input vector, output vector, or both.

3.1.1 False sharing

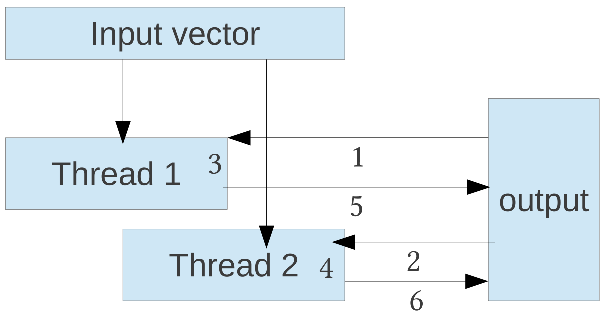
When a processor requests access to a memory slot, it will, on most systems, load the whole cache line specified memory slot is on. This is justified by the fact that the next memory slot needed is most likely nearby the current one and therefore the number of load operations can be reduced. In a sequential program this gives few, or no problems, and will most likely make the program more efficient. In a parallel program, however, false sharing occurs.

False sharing happens when two (or more) threads asks to access different memory slots that are on the same cache line, as shown in Figure 3.1. When thread 1 updates one memory slot, the cache line stored in local cache of thread 2 will have to be updated or invalidated. Even though thread 2 may never access the memory slot changed by thread 1. The same effect can happen when thread 2 updates a memory slot, in which case the cache line loaded by thread 1 must be updated or invalidated. This can go back and forth and may have huge negative effects on performance [1].

In SMvM, false sharing occurs when threads are writing to the output vector. How frequently this happens is dependent on the structure of the matrix, the matrix representation, and the distribution of work. Since the amount of false sharing is dependent on so many factors, it is hard to eliminate. However, we can use techniques that tries to keep the threads from accessing memory close by each other, thus reducing it.

3.1.2 Write conflicts

For algorithms using the CCS and COO representations false sharing is not the only type of conflict we have to deal with. These algorithms also have to handle write conflicts. These are a much more severe problem as they can lead to non-deterministic execution and incorrect solutions.



Write conflicts can occur when threads behave in the following way:

1. *Thread 1 reads the original value, **org**, from output vector.*
2. *Thread 2 reads the original value, **org**, from output vector.*
3. *Thread 1 adds its value to the original value, creating a new output value, **out1**.*
4. *Thread 2 adds its value to the original value, creating a new output value, **out2**.*
5. *Thread 1 overwrites the original value in output vector with **out1**.*
6. *Thread 2 overwrites **out1** in output vector with **out2**.*

FIGURE 3.2: Illustration of a write conflict

A write conflict occurs when two (or more) threads try to write to the same memory slot at the same time. When a thread shall write new information into a memory slot, as in the output vector in SMvM, this usually happens in three steps. It copies the value from memory, next it manipulates the copy, and finally writes the manipulated copy back into the memory slot, thus overwriting the original value. If two threads tries to do this simultaneously, the new value created by one of the threads can be overwritten by the other, as shown in Figure 3.2. This means that write conflicts can lead to errors in the computation and furthermore may lead to incorrect and non-deterministic results overall.

With CCS-representation a parallel SMvM algorithm will distribute the matrix to the threads by giving them unique columns. This also gives the threads unique parts of the input vector to work on, however they have to share the output vector. This can lead to write conflicts.

There are a couple of ways to handle write conflicts, all with disadvantages that must be taken into account.

- One way is to have one lock on each of the elements of the output vector. A lock will only allow one thread to work on a slot at a time. Only when a thread has released the lock, can other threads write to the memory slot. The use of locks makes threads wait their turn to access specific memory slots, thus potentially creating bottlenecks. It also takes time to create, acquire, and release locks. In OpenMP locks can be implemented with an atomic operation. ALG 7 and ALG 6 are CCS- and COO-algorithms using locks on the output vector.
- Another technique is to let each thread write its partial solution into a private storage. When all the threads are finished, the partial solutions are merged together to form a final solution. This eliminates the locks and the bottlenecks they may create. However, an extra step is used to merge together the partial solutions. This requires more space, proportional to the number of threads, possibly reverting the speed-up. The use of this technique is shown in ALG 8.
- A third way of avoiding write conflicts is to schedule the threads such that they never write into the same position at the same time. This avoids using locks and does not need an extra step for merging solutions. However, it has other potential problems. To achieve such a scheduling, a preprocessing step is needed. The preprocessing of data is potentially a heavy and time consuming step, depending on the algorithms used and the structure of the matrix. The original structure of the matrix may also be altered, possibly affecting the efficiency of the algorithm.

Algorithm 6: Parallel SMvM algorithm using COO-representation

```
Data: $M$ a sparse matrix of size $m \times n$ stored in COO format, $v$ an input vector of
      length $n$
Result: $w$, an output vector of length $m$

  set all indices in $w$ to 0
  for $i \leftarrow 1$ to $nnz$ do in parallel
    $w[M[i,1]] \leftarrow w[M[i,1]] + M[i,0] \times v[M[i,2]]$ //Atomic Operation
  end for
```

Algorithm 7: Parallel SMvM using CCS-representation with locks on the write operation.

Data: M a sparse matrix of size $m \times n$ stored in CCS format, v an input vector of length n

Result: w the result vector of length m

```
for $i \leftarrow 1$to $n$ do in parallel
  $t \leftarrow v[i]$ // store value in temporary variable
  for $j \leftarrow IA[i]$ to $IA[i+1]$ do
    $w[JA[j]] \leftarrow w[JA[j]] + A[j] \times t$ //Atomic Operation
  end for
end for
```

Algorithm 8: Parallel SMvM using CCS-representation writing into private out vectors.

Data: M a sparse matrix of size $m \times n$ stored in CCS format, v an input vector of length n, wt a two dimensional vector where each thread stores its partial solution

Result: w the result vector of length m

```
$numThreads \leftarrow$ total number of threads
for $i \leftarrow 1$ to $n$ do in parallel
  $ID \leftarrow$ id of thread
  $t \leftarrow v[i]$
  for $j \leftarrow IA[i]$ to $IA[i+1] - 1$ do
    $wt[ID][i] \leftarrow wt[ID][i] + A[j] \times t$
  end for
end for
for $i \leftarrow 1$ to $n$ do in parallel
  $t \leftarrow 0$
  for $j \leftarrow 0$to$numThreads$ do
    $t \leftarrow t + wt[j][i]$
  end for
  $w \leftarrow t$v
end for
```

3.2 Even work load

To have even work load is an important aspect in parallel programming [1]. At the end of a parallel step or at a point where the threads must synchronize, all threads have to wait until the last one has finished. With an even work load, threads should finish at about the same time, reducing the time they are waiting.

For SMvM-algorithms, the distribution of work is done in one of two ways. Either as a distribution of rows or columns (depending on if CRS or CCS representation is used) or as a distribution of non-zero elements.

3.2.1 OpenMP scheduling techniques

The basic parallel SMvM algorithms using CRS or CCS representations divides the work to the threads in their parallel for-loops. The division of work is done by allocating rows or columns to the threads, and then the threads do their allocated work. The technique used in the parallel loop controls in what way the scheduling is done. In OpenMP there are three main scheduling techniques, *static*, *dynamic*, and *guided* [10][11].

static For the basic algorithms using CRS or CCS representations, this means that each thread gets (almost) the same number of rows or columns to work on. This division gives a potential problem if we want an even work load. It takes no regard to the density of the rows or columns. In other words, the structure of the matrix is very important for how good this division is.

dynamic Given that the number of rows or columns (n or m) is much greater than the number of threads (p), the structure of the matrix should not be as important for this scheduling. Threads with heavy work load will not be allocated more work until they are finished, and thus we can expect the threads to get a more even work load. The on-the-go allocating of work however may be costly.

guided As with the dynamic scheduling, we can expect a more even work load than with static scheduling. Also, like dynamic scheduling, we can expect a certain cost due to the on-the-go allocation, but it may not be as high as with dynamic scheduling, since there are fewer chunks to be allocated.

3.2.2 Distribute on number of non-zero elements

Except for the COO-algorithm, which iterates over a list of the non-zero elements, the OpenMP scheduling techniques gives no guarantee of an even work load. Dynamic scheduling, which we might suspect to give an even work load, will probably be slowed down by non-consecutive data and on-the-go allocation.

Preprocessing the data might give a better distribution. Ideally each thread should be allocated $\frac{nnz}{p}$ elements, but this might be very hard if some rows or columns are denser than others. So the preprocessing must look for another good distribution. This can for instance be made by a scheme similar to min-makespan [2].

The min-makespan problem is given n jobs, $j_1, j_2, \ldots, j_n$ of varying sizes that are to be scheduled on m identical machines. The goal is to distribute the jobs such that the machine with the heaviest work load gets as little work as possible. This problem is almost identical to what we are aiming for. Unfortunately the min-makespan problem is NP-hard [2]. However, there are several approximation algorithms for this problem, that can be efficiently used for the distribution of rows or columns to threads. This is discussed further in Chapter 5.

3.3 Cache efficiency

In most systems, when a processor needs to access an memory slot, it loads in the whole cache line into memory [1]. The load operation has a certain cost, so it is important to utilize the cache line as much as possible is important to get an effective program.

In SMvM algorithms there are three memory slots a processor needs at the same time, one from the input vector, one from the matrix and one from the output vector. This means that we can expect that three cache lines are loaded into local memory. These should be handled efficiently. The CRS-, CCS-, and COO-representations all access elements in the matrix consecutively, thus giving optimal cache efficiency. However, the structure of the matrix dictates how efficiently the cache lines from the input and output vectors are used. If the matrix has an unfortunate structure, the algorithms will “jump” back and forth in the input or output vector. This will force the algorithm to load new cache lines after only using one or a few elements.

A much used solution to make SMvM algorithms more cache efficient, is to restructure the matrix so that the cache lines are utilized more efficiently [3]. The main idea

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

FIGURE 3.3: Arrowhead matrix of size 7×7

is to make some sections of the matrix denser, while others are sparser or empty. This makes the use of cache lines loaded for the dense parts more efficient, while fewer or non cache lines are loaded for the rest of the matrix.

Another way to improve cache efficiency is to change the way the matrix is iterated. One example of this is the CRS *zig-zag* algorithm, presented in Yzelman & Bisseling [4]. Different techniques to increase cache efficiency presented and further discussed in Chapters 4 and 6.

3.4 Unfortunate structured matrices

All the other mentioned challenges for parallel SMvM are in some degree dependent on the structure of the matrix being worked on. A matrix with an unfortunate structure is a matrix that in a large degree creates one or more of the problems.

One example of a possible unfortunate matrix is a random matrix. Although a random matrix is easy to distribute evenly, it creates conflicts and is inefficient with regard to cache use. The conflicts comes from the fact that there is no structure that ensures the threads to work on memory slots with a certain distance. They are also inefficient with regard to cache use, as the probability of consecutive elements needing data from the same cache line is $\frac{1}{O(n)}$. Random matrices may be handled or restructured to reduce these problems. In Chapter 6. an algorithm that increases cache efficiency, especially on random matrices, is presented.

An other example is an *arrowhead* matrix. An arrowhead matrix, shown in Figure 3.3, has non-zero elements on the diagonal and on one row and one column. This creates problems for both CRS and CCS representations, as distributing work by rows or columns will give uneven work load. How different SMvM algorithms behave on arrowhead matrices are discussed in Chapter 5.

Chapter 4

Previous Work

Sparse matrix vector multiplication is a well studied field and much work has been done to improve the performance of SMvM-algorithms. This chapter introduces some of the previous work done in this field, mostly focusing on research on efficient cache use and how to avoid write conflicts. It also presents some work done on graph colouring algorithms, as this is used to improve CCS-algorithms.

Matrix representations using less space than the standard is an improvement to SMvM-algorithms. These include the COO, CRS and COO representations presented in Chapter 2. Blocked CRS (BCRS) is also much used. BCRS stores the matrix in small boxes, where each box is treated like a dense matrix [12]. This representation is, amongst others, used in Vuduc and Moon [6], where the matrix is divided into several matrices with different block sizes. Other representations include Compressed Diagonal Storage, Jagged Diagonal Storage and Skyline Storage [12, 17].

Most of the work done to improve parallel SMvM includes some sort of preprocessing step. This involves processing the matrix such that it is handled more efficiently by the SMvM algorithm. To justify the extra cost of a preprocessing step, it is often pointed out that SMvM is often applied iteratively using the same matrix in many computations. One example is its use in iterative methods for solving linear equations [13]. Here a number of SMvM are preformed, building an increasingly better approximate solution.

4.1 Efficient cache use

CRS representation is much used for parallel SMvM algorithms. Unlike the CCS representation it does not create write conflicts, thus the focus on improvements has been elsewhere. There has especially been preformed quite a lot work on efficient cache use.

4.1.1 Enlarge dense blocks

To use of a cache line efficiently, the work load on it should be significantly higher than the time used to load it into memory. For SMvM algorithms this means that dense sections of the matrix are more cache efficient than the sparse ones. Pinar and Heath, [5],propose a modification to the basic CRS algorithm. By adding a fourth vector, which holds the start index of contiguous sections, the algorithm knows when it should load a new block. This may reduce the number of load operations, but it adds another loop to the algorithm.

Pinar and Heath also discus the idea of reordering the matrix to increase the contiguous sections and point to previous work done on the field. As reordering the matrix optimality is known to be NP-hard, heuristics must be used for practical solutions. They observe that the problem can be seen as closely related to the Travelling Salesman Problem. Since this is a heavily studied problem, many heuristics are available, and similar ones can be used.

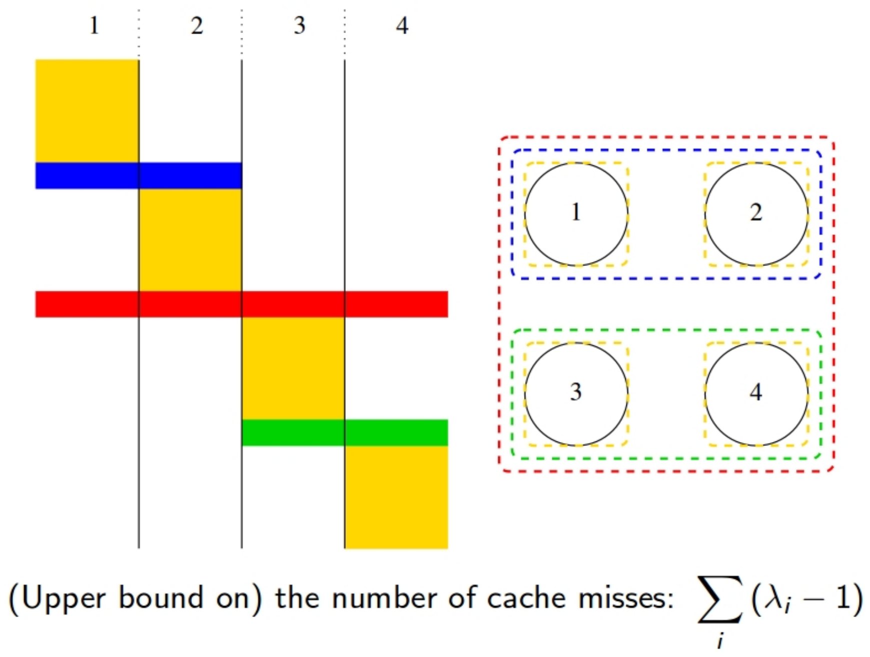
Their experiments confirm increased efficiency of exploiting the dense, contiguous blocs. The results gives improvements of up to 33% and an average of 21% improvement. These results are also shown to be better than previous work in the field.

4.1.2 Restructuring and Reordering to reduce cache miss

For sparse matrices the efficiency of the cache use is very dependent on the structure of the matrix. Therefore it is not surprising that a lot of the work done to increase the efficiency of cache use consists of partitioning or reordering the matrices.

In Yzelman and Bisseling [4], introduces a method to reduce cache misses and increase work on cache lines. This is done by permuting the matrix into a Separated Block Diagonal (SBD) form, which can give an upper bound on the number of cache misses.

To efficiently work on matrices on SBD form, Yzelman and Bisseling give a new, more


1
2
3
4
1
2
3
4
(Upper bound on) the number of cache misses: $\sum_i (\lambda_i - 1)$

The red section is N_c after the first partitioning, the blue and green lines are N_c after the second partitioning and the yellow sections are the created blocks.
Figure from [4].

FIGURE 4.1: Illustration of SBD matrix

cache efficient variant of the CRS-algorithm, called CRS-zig-zag. Instead of reading each row from left to right, CRS-zig-zag alternates, reading the first line from left to right and the next from right to left. This means that the cache line from the input vector that is already in memory at the end of a row can be used at the beginning of the next. They argue that this reduces the number of cache misses on a dense row from $O(n)$, to $O(n - L)$, where n is the length of the row and L is the number of cache lines.

Permuting the matrix to SBD form is done recursively. The matrix A is modelled as a hyper-graph $\mathcal{H} = (V, N)$, where V is the set of columns in A and N is the set of hyper-edges, each corresponding to a row in A. A partitioning of V into V_1 and V_2 is made and N is divided into three parts. The division of N is according to it having vertices in $V1$ (N_+), $V2(N_-)$ or both(N_c). This is applied recursively on V_1 and V_2. Figure 4.1 show a matrix on SBD form after two recursive steps. Optimally $p = \frac{n}{wL}$, where w is the length of a cache line and p is the number of parts created by the recursion. Yzelman and Bisseling argue that by setting $p \to \infty$ the same bound will be reached.

In the experiments p is limited to at most 400, as $p \to \infty$ or even $p = n$ would make the permutation extremely expensive. The results shows considerable speed-up for some matrices during SMvM. However, for matrices that already have a cache-friendly structure, the results show little speed-up or even slow-down.

4.2 Avoiding write conflicts with CCS-algorithms

Write conflicts is the most severe problem for CCS-algorithms and sets them at a disadvantage compared to CRS-algorithms. In the paper “Parallel sparse matrix vector multiplication” by Azad and Pothen[7], it is discussed whether it is possible to eliminate this problem by preprocessing the data.

They introduce different techniques for handling write conflicts for CCS–algorithms, compares different solutions to the run times achieved by a basic CRS-algorithm, which does not encounter write conflicts, and discusses the results and how they compare to the CRS–algorithm.

4.2.1 Basic solutions

Azad and Pothen give two basic algorithms that use the CCS-representation and one that uses CRS-representation. These algorithms are similar to CCS with locks (CCS-locking, ALG 7), CCS with private storage (CCS-private, ALG 8), and the CRS-algorithm (ALG 5). The two CCS-algorithms use different schemes to avoid write conflicts CCS-locking uses locks or atomic operations when writing into the output vector, thus only letting one thread write to a memory slot at any given time. CCS-private avoids using locks by letting each thread write into a private output vector and then adds the output vectors together in a second parallel step.

4.2.2 CCS with colouring

To let CCS-algorithms avoid handling write conflicts, Azad and Pothen introduce the idea of colouring the columns in a preprocessing step. The colouring is applied to the columns such that two columns have different colour if they have a non-zero element in the same row. All columns with the same colour can update the output vector simultaneously without causing any write conflicts. This is because the union of the columns with the same colour has at most has one element in each row, and thus at most one element for each position of the output vector.

They gives four scheme to handle the coloured columns, gradually working towards one that can compete with the running times achieved by the basic parallel CRS algorithm.

- The first algorithm iterates over the colours, and it enters a parallel section for each colour. It then iterates over all the columns in a parallel loop using an if statement to test whether or not each column has the current colour. This algorithm is inefficient as all the columns are iterated over for each colour.

- To increase efficiency, the second algorithm eliminates the full iterations for each colour. This is done by using two extra vectors, *colorColumn* storing the indices of the columns, sorted on the colours, and *colorColumnPtr* storing the starting index for each colour in *colorColumn*. The algorithm iterates over *colorColumnPtr*, entering a new parallel step for each colour. In each parallel step it iterates over the columns with the current colour. This means that each column is only accessed once. This algorithm is more efficient, but there is still a problem that the algorithm accesses the columns non-consecutively.
- The third algorithm uses *colorColumn* and permutes the matrix such that coloumns with the same colour are stored together in the matrix. The iteration over *colorColumnPtr* now lets us access the columns that are stored consecutively, increasing the efficiency.
- For the last algorithm (ALG 9) the input vector is permuted in the same way as the columns, increasing the efficiency further. This algorithm is referred to as CCS-col.

Algorithm 9: CCS + Colouring 4 (CCS-col)

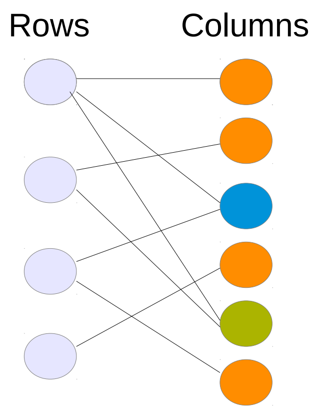
```
Data: $M$ a sparse matrix of size $m \times n$ stored in CCS format, $v$ an input vector of
      length $n$, $color$ a vector of length $n$ storing the colours of each column
Result: $w$ the result vector of length $m$

numColour ← the number of distinct colours
$M'$ ← Column permutation of $M$ such that columns with the same colour stay
together. ($M'$ is made of $A'$, $IA'$ and $JA'$, permuted versions of $A$, $IA$ and $JA$.)
$v'$ ← permutation of $v$ such that indices with the same colour are ordered
consecutively.
colorColumnPtr ← a vector of size ($numColour + 1$) where the $ith$ entry stores the
starting index of columns with colour $i$
for $col \leftarrow 1$ to $numColour$ do
  for $j \leftarrow colourColumnPtr[col]$ to $(colourColumnPtr[col+1]-1)$ do in parallel
    $t \leftarrow v'[j]$
    for i ← IA'[j] to IA'[j+1] do
      $w[JA'[j]] \leftarrow w[JA'[j]] + A[j] \times t$
    end for
  end for
end for
```

Matrix	Dimention	# Non Zero Elements
af_shell10	1.5M	52.7M
boneS10	0.9M	55.5M
cage14	1.5M	27.1M
cage15	5M	100M
random	5M	100M
Hamrle3	1.4M	5.5M
HV15R	2M	283M
nlpkkt_240	28M	775M

Table 4.1: Matrices used in AP12

Azad and Pothen give timings using the eight sparse matrices, shown in Table 4.1. The results show that CCS-col is faster than CCS-locking, even though CCS-locking sometimes scales better. The results obtained by CCS-private are closer to CCS-col when using few threads and sometimes even faster. However, when the number of threads increases above 8 or 16, CCS-private stops giving speed-up and the run times increases, giving CCS-col an advantage. The results also show that CCS-col is still not as fast as the CRS-algorithm, but the difference in performance is smaller when the matrices are denser. They argue that this is due to more potential write conflicts for the basic CCS-algorithms compared to the CRS-algorithm and CCS-col. The general performance difference between the two algorithms is explained with the fact that a restructuring of the matrix will destroy its structure and subsequently the use of consecutive information is reduced.



The graph represents matrix M, shown in Figure 2.2, colouring applied by either of ALG 10 or ALG 11.

FIGURE 4.2: Distance-2 colouring on a bipartite graph.

4.3 Colouring algorithms

Azad and Pothen introduces colouring of the columns to avoid write conflicts for SMvM with CCS-representation. However, the efficiency of CCS-col is dependent on which colouring algorithm it uses. A good colouring algorithm for parallel SMvM will use few colours, thus reducing the number of parallel steps. Moreover, with few colours we can expect there to be enough work in each colour class to justify a parallel step. The colouring of columns in a matrix can be seen as the same problem as a graph colouring. One can view the columns as vertices where there is an edge between two columns if they have an element on the same position. A colouring is legal iff two neighbouring vertices have different colour. In Azad and Pothen, the colouring is viewed as a distance-2 colouring on a bipartite graph. The columns are on one side and the rows on the other. If a column i has a non-zero element in position j, there is an edge between column vertex i and row vertex j. An example is shown in Figure 4.2.

To achieve a colouring using few colours, we can look at the *min-colouring* problem. This problem takes as input a graph and gives a colouring of the vertices, using the minimum possible amount of colours. However, *min-colouring* on general graphs is NP-hard [2]. So computing an optimal solution to either problem will take a very long time when the datasets are large. Thus a faster, but non-optimal algorithm must be used. Azad and Pothen does not mention what colouring algorithm is used, but we can assume that this is some approximation algorithm or a non-optimal heuristic.

4.3.1 Greedy Graph Colouring

One fast graph colouring algorithm is the greedy algorithm, shown in ALG 10. It starts with the first colour, iterates over all the vertices, and applies the colour to every uncoloured vertex that does not have a neighbouring vertex with this colour. The algorithm repeats the iteration, with a new colour every time, until all the vertices have been coloured.

Algorithm 10: A greedy graph colouring algorithm

```
Data: G(V,E), a graph G with vertices V and edges E
Result: colour, a vector that hold the colours of the vertices

  $currColour \leftarrow 1$
  all indices in $colour$ are set to $-1$
  while $\exists v \in V$ where $colour(v) = -1$ do
    for all $v \in V$ where $colour(v) = -1$ do
      if $\forall w \in N(v)$ has $colour(w) \neq currColour$ then
        $colour(v) \leftarrow currColour$
      end if
    end for
    $currColour \leftarrow currColour + 1$
  end while
```

4.3.2 Greedy distance-2 colouring

Another possible algorithm is to use a greedy distance-2 colouring algorithm for bipartite graphs. Such an algorithm is shown in ALG 11, and is based on a general greedy distance-2 colouring [8]. It iterates over the column vertices, and for each column vertex v, it iterates over the neighbouring row vertices. Each of the row vertices then iterates over their already coloured neighbouring column vertices, and set the colours of these to be unavailable for v. After deciding which colours are unavailable, v is coloured with the lowest available colour.

Algorithm 11: A greedy distance-2 colouring algorithm

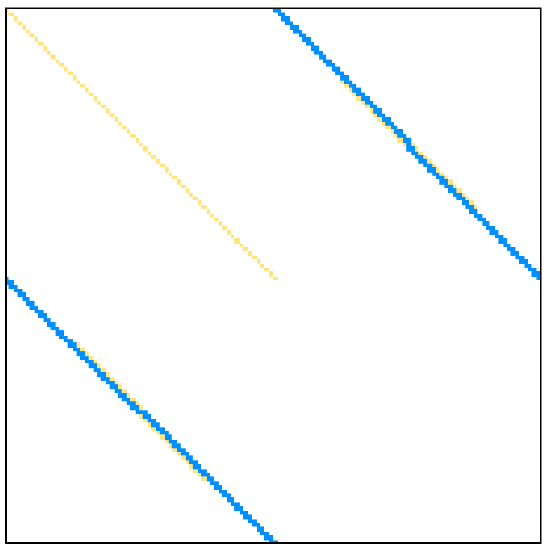
```
Data: $G(V_1, V_2, E)$, a bipartite graph with vertices $V_1$ and $V_2$ and edges $E$,
      $forbiddenColours$, a vector of size $n$
Result: $colour$, a vector that holds the colouring of $V_1$

Initialize $forbiddenColours$ with some value $a \notin V_1 \cup V_2$
for all $v \in V_1$ do
  for all $w \in N(v)$ do
    for each coloured vertex $x \in N(w)$ do
      $forbiddenColors[colour[x]] \leftarrow v$
    end for
  end for
  $colour[v] \leftarrow \min \{ c > 0 : forbiddenColors[c] \neq v \}$
end for
```

Chapter 5

Even Work Load

In this chapter the importance for parallel SMvM algorithms to have an even work load is discussed. A preprocessing technique that improves the distribution of work on the CRS-algorithm, compared to the scheduling schemes in OpenMP is introduced. Moreover, how to achieve an even work load on matrices with an unfortunate structure is discussed. From this, an argument that the COO-algorithm may be the best choice is made, even though it uses more memory.



$n = 28M \ nnz = 775M$

FIGURE 5.1: Matrix, *nlpkkt_240*

5.1 Improved distribution in CRS-algorithms

In Chapter 3, it is argued that the different scheduling techniques in OpenMP all have negative issues when distributing rows or columns in a parallel SMvM algorithm. Non of the techniques give a guarantee of an even work load. Furthermore, dynamic and guided scheduling will possibly also suffer from tthe on-the-go allocation. It is argued that a preprocessing step which allocates the rows or columns evenly could give a more even distribution.

The min-makespan problem is introduced as a similar problem to the distribution of rows or columns in parallel SMvM. However, since min-makespan is NP-hard, an approximation algorithm or heuristic must be used.

5.1.1 Approximation to *min-makespan*

In the *min-makespan* problem, we are given a set of m machines, $M_1, M_2, \ldots, M_m$ and a set of n jobs, $j_1, j_2, \ldots, j_n$. Each job has a processing time t_j. The task is to distribute the jobs to the machines such that the machine with highest total processing time is as low as possible.

The greedy approximation to *min-makespan* iterates over the jobs, and allocates each job to the machine with the current lowest work load. This is a 2-approximation [2], i.e. the approximation is proven to give a solution at most twice as large as the optimal solution. By saying that the p threads are machines $M_1, \ldots, M_p$, the rows or columns are jobs and t_j is defined by the number of non-zero elements on a row or column, it is possible to apply the greedy approximation directly. Thus we can guarantee that the heaviest thread has $c \leq 2OPT$ non-zero elements, a guarantee that non of the OpenMP techniques could give. However, using the greedy approximation algorithm will probably give each thread non-consecutive rows or columns. Thus we are still facing one of the problems of *dynamic* scheduling.

5.1.2 Allocating consecutive data to threads

What we want is an approximation algorithm or heuristic that gives a good distribution, but also gives each thread consecutive rows or columns. We know that the optimal solution to *min-makespan*, OPT, is bounded by $\frac{nnz}{p} \leq OPT \leq nnz$. This bound also holds for distribution of consecutive rows or columns. We define c highest number of non-zero elements we can allocate to a thread. Then we can do a search for a c as low as

p	CRS static	CRS dynamic	CRS guided	CRS binary search	basic COO
8	3.9M	1M	17M	100	0
16	2.7M	2.4M	9M	100	0
32	2M	1.4M	9M	200	0

TABLE 5.1: Difference in number of elements between lightest and heaviest thread on *nlpkkt_240*

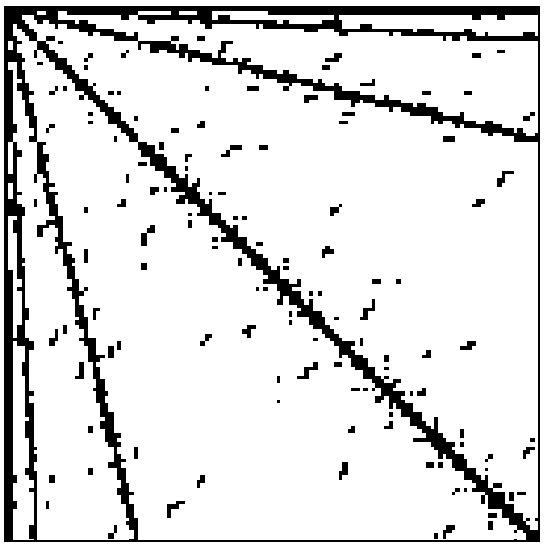
possible. A good way to do this is using binary search; we guess c, and greedily try to allocate consecutive rows or columns into p sets, such that each set has at most a total of c non zero elements. If it works, we reduce c, if not we increase c, and try again. We do this until we find the smallest possible c that gives a feasible solution.

The distribution given by the binary search for c is probably better than, or at least as good as, the distributions made by the OpenMP scheduling techniques. Furthermore, since the rows or columns are allocated consecutively and the structure remains the same, run times on SMvM algorithms may be better. Table 5.1 shows the difference in number of non-zero elements allocated to threads with different scheduling techniques on the matrix *nlpkkt_240*, from Figure 5.1. We see that the distribution created by the binary search is almost completely even (on *nlpkkt_240*), and much better than the distributions given by the OpenMP scheduling techniques.

5.2 Arrowhead matrices - Can COO-representation be the fastest?

For parallel SMvM algorithms using CRS or CCS representation, how the elements are distributed to the threads is almost always dependent on the structure of the matrix. For matrices with a unfortunate structure, for instance an arrowhead matrix like *delaunay_n24*, (Figure 5.2), this can lead to large differences in the work load between the threads. To have an even distribution one may need to split a row (for CRS) or a columns (for CCS) between two or more threads. This is a problem for CRS and CCS representations, which only stores the start index of the rows or columns to reduce memory used.

For the COO-representation, however, there is no structural problem with splitting a row or a column between threads. In fact, the parallel COO-algorithm (ALG 6), most likely, will do this. This algorithm, like the basic CRS and CCS-algorithms, divides the work using the OpenMP scheduling techniques. But unlike the CRS- or CCS-algorithm, it iterates over the elements. This leads it to ignore the rows or columns and their possible difference in density will not affect the distribution.



$n = 16.8M \; nnz = 100M$

FIGURE 5.2: An arrowhead matrix, *delaunay_n24*

Even though the parallel COO-algorithm fives an even distribution of elements, it will still run into other problems. Write conflicts will most likely occur, and the COO-representation will use up to 50% more space than CRS or CCS- representations. However, for matrices with a unfortunate structure, these problems may be small compared to dividing the work evenly.

5.2.1 Distribution of elements

The parallel COO-algorithm always distributes the elements evenly. But for matrices with an easy to handle structure, like *nlpkkt_240* (Figure 5.1), the distribution of elements can be quite good for other scheduling techniques as well (Table 5.1). Even though CRS with guided scheduling gives a difference of 17M elements between heaviest and lightest thread, this is only a little more than 2% of the total number of elements. For CRS with binary search, there is almost no difference in the work assigned to the threads.

For arrowhead matrices however, like *delaunay_n24* (Figure 5.2), the distribution of the OpenMP techniques and the binary search. Table 5.2 shows the difference between the heaviest and the lightest loaded thread for different scheduling techniques for this matrix. It shows that *static* scheduling gives a difference of over 30M for all $p \geq 2$. This represents over 30% of the total number of elements. Furthermore, for $p = 32$ (at least) one thread has approximately $10\times$ as many elements than it would have with a perfect

p	CRS static	CRS dynamic	CRS guided	CRS binary search	basic COO
1	0	0	0	0	0
2	34M	8M	4M	38	0
4	34M	20M	17M	11M	1
8	33M	20M	17M	16.7M*	1
16	33M	17M	17M	16.7M*	1
32	33M	17M	17M	16.7M*	1

*All elements allocated to six threads.

Table 5.2: Difference in number of elements between lightest and heaviest thread on *delaunay_24*

distribution. The other scheduling techniques are not much better. CRS with binary search, distributes all elements to the six first threads. This gives no justification of higher degrees of parallelism. The basic COO-algorithm, however, still distributes the work load perfectly, and we may expect it to be the best algorithm.

Chapter 6

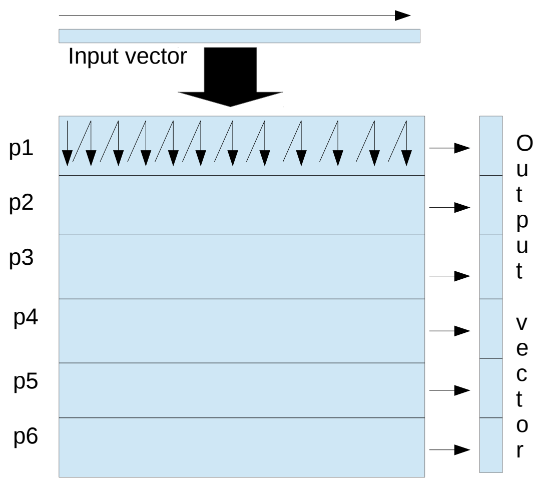
Efficient Cache Use

Much work to improve parallel SMvM with CRS-representation is done with regard to cache efficiency. Both Pinar and Heath [5] and Yzelman and Bisseling [4] introduces preprocessing techniques that restructure the matrix in order to reduce the number of cache lines loaded and do more work on them when loaded. Moreover, in Yzelman and Bisseling, the CRS-zig-zag algorithm is introduced, an algorithm that increases cache efficiency by altering the iteration over the elements in the matrix. In this chapter a new idea to increase cache efficiency is introduced. By combining the iteration and structure used by the different parallel SMvM algorithms, we discuss a possibly more cache efficient algorithm.

The basic parallel CRS-algorithm (ALG 5) iterates over the rows in parallel. For each row, it stores into one specific location on the output vector, but it might read elements from the whole input vector. This gives a potential $O(\frac{n}{w})$ cache lines to load for each row, where w is the length of the cache line. In total the program might load cache lines from the input vector $O(\frac{n}{w} \times m)$ times. The CRS-zig-zag reduces this somewhat, but the number of cache lines read from the input vector is still high [4].

6.1 A hybrid between CRS and CCS representations

The basic parallel CCS-algorithm (ALG 7) works much like the CRS-algorithm, except it iterates over columns instead of rows. Thus there is inefficient cache use on the output vector, instead of the input vector. This may also cause write conflicts. However, a hybrid representation using both elements from the CRS and the CCS-representations may remove the write conflicts and give the parallel SMvM better cache efficiency.



Each thread, $p_1 \dots p_6$, is allocated a number of rows. Then each thread sorts its allocated elements by column index, and locally does a CCS-algorithm.

FIGURE 6.1: Illustration of the parallel hybrid SMvM algorithm

By distributing the rows like in a CRS-algorithm and letting each thread run the CCS-algorithm locally, each thread only needs to load each cache line from the input vector once. Moreover, there are no write conflicts, as each thread writes into a unique part of the output vector. The hybrid algorithm may still have to load a new cache line from the output vector for each element. However, each thread will only write into a unique consecutive part of the output vector, of approximate size $\frac{n}{p}$. Thus, the probability that two consecutive elements needs the same cache line from the output vector is higher than with the CCS-algorithm. This is especially true for random matrices, where the probability that two consecutive operations writes to the same cache line on the output vector increases from $\frac{1}{O(n)}$ (with CCS) to $\frac{1}{O(\frac{n}{p})}$ (with hybrid). An illustration of the hybrid algorithm is shown in Figure 6.1.

Chapter 5 presented a CRS-algorithm where each thread is allocated a set of consecutive rows in a preprocessing step. Using this preprocessing it is possible to continue to sort the non-zero elements allocated to each thread by the columns. However, this preprocessing requires an alteration to the matrix representation, as the elements are now sorted on row indices globally and column indices locally.

6.2 COO-representation with CRS-CSS-hybrid iteration

The CCO-representation holds both the column and row coordinate for every non-zero element. Thus it is easy to sort and restructure. This means that using COO-representation will let us easily implement the hybrid algorithm.

The regular parallel SMvM algorithms using COO-representation may run into problems. However, most of these should not occur here. Since the hybrid algorithm has control over which rows that are allocated to each thread, it will mostly have to handle the same problems as the CRS-algorithm. The only problem that remains is the increased use of storage, which may be as much as 50% more.

The increased use of storage may in some cases prove the hybrid algorithm to be worse than the CRS-algorithm, since there is a higher amount of data movement [3]. However, for the efficiency on the cache lines, the hybrid algorithm will probably be better, and possibly so much that the extra storage is worth the cost. Thus, how the hybrid- and CRS-algorithms compare will depend on the structure of the matrix.

Chapter 7

Improving CCS with colouring

In Azad and Pothen [7], colouring of the columns is presented as a way to avoid write conflicts for parallel SMvM algorithms using the CCS-representation. Even though using CCS with colouring (ALG 9) yields better results than the other two CCS-algorithms (ALG 7 and ALG 8), it is still shown to be slower than the CRS-algorithm. However, as we briefly mentioned in Chapter 1, the dimension and structure of certain matrices may favour the CCS representation.

In this chapter we introduce and discuss some possible improvements to CCS algorithms with colouring. Two potential problems are addressed, false sharing and inefficient parallel steps. To handle false sharing an alteration to the colouring algorithm is presented. Furthermore its possible positive and negative effects are discussed. To handle the inefficient steps, two techniques are discussed. The first is to identify and handle them using alternative algorithms. The second uses a new colouring algorithm that (almost) guarantees efficient parallel steps.

7.1 Reducing False Sharing

When restructuring the matrix in a preprocessing step, the structure of the matrix is altered and the threads may have to access the output vector in a non consecutive way. This may lead to an increase in cache misses and an overall slow-down of the program. When a reordered matrix is used in a parallel SMvM algorithm all the threads are accessing the output vector in an unstructured way. This may increase false sharing, i.e. threads writes to the same cache line.

7.1.1 Increased distance

Propose a stricter colouring algorithm that will probably reduce false sharing. We define the *distance* between two columns to be the lowest difference of index of non zero elements. i.e. two columns with an element on the same index have $distance = 0$. The normal colouring algorithms requires a $distance \geq 1$ to give two columns the same colour.

By altering the colouring algorithm, so that the distance between columns must be larger in order for them to get the same colour, we may reduce the amount of false sharing. If the distance between the columns of the same colour increase, the probability of two threads writing to the same cache line is reduced. Furthermore, if the distance required to get the same colour is larger than the number of elements on a cache line, we can guarantee that false sharing will not occur. However, there are possible negative effects.

With increased distance, less work can be done on each loaded cache line, yielding a less cache efficient algorithm. This can be reduced by utilizing the fact that each thread can hold columns of different colours since each thread works sequentially on its allocated columns. However, this will require more preprocessing and/or more synchronization between threads. Another possible problem is that the colouring algorithm requiring increased distance will use more colours than the original colouring algorithm. More colours will increase the number of parallel steps, thus decreasing the amount of work in each each parallel step. This may make parallelization less efficient.

7.2 Efficient parallel sections

One of the fundamental ideas of parallel programming is that the total work load have to be big enough to justify a parallel step. For the basic parallel SMvM algorithms this is usually not a problem. They only have one parallel step and the total work load is often very large. For CCS with colouring, however, the work is done in many parallel steps, one for each colour.

For CCS with colouring, the work load given to each thread in each parallel step is dependent on the number of colours used. As discussed in Chapter 4, computing an optimal colouring algorithm can take a very long time, so a greedy, more time efficient approach must be used. A non-optimal colouring algorithm will use a higher amount of colours. The basic greedy colouring algorithm, COL1 (ALG 10), will for most sparse graphs be relatively efficient and give a colouring with a reasonable low number of

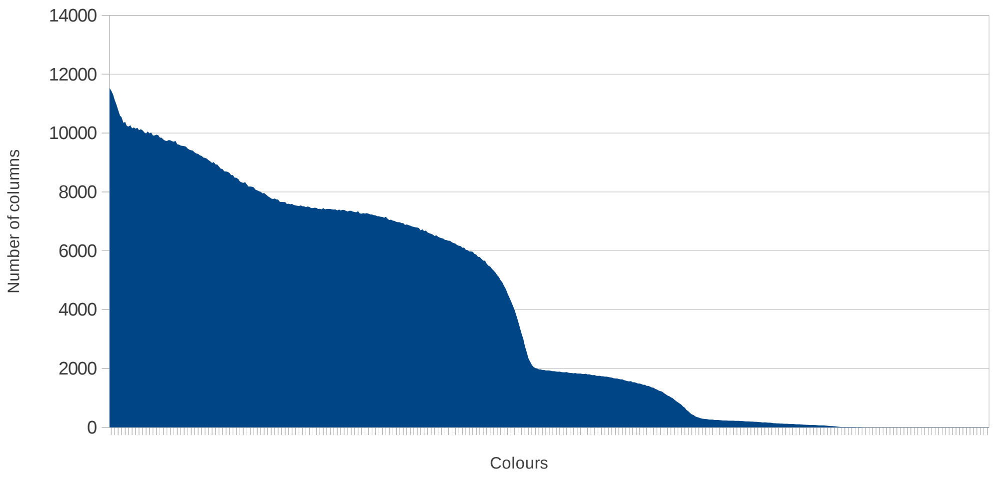


Figure 7.1: Use of colours by COL1 on matrix HV15R

colours. However COL1 will not use the colours evenly. It will most likely give a distribution where the first colours are heavily used and subsequent colours are used less. Furthermore, this can possibly result in a distribution where several colours are used only on a few columns.

7.2.1 When is the greedy colouring bad?

Figure 7.1 shows the distribution of colours on the matrix *HV15R*. This is a typical distribution of colours given by COL1. The distribution of colours is uneven and the overall tendency is that each new colour is used less than the previous. At the end there is a ***tail***, where each colour is used on only very few columns. (The part of the matrix that is not in the tail is referred to as the *good* part.) The tail can give CCS with colouring a problem, as it starts a new parallel step for each colour, even if it is only used on a couple of columns, giving possibly inefficient parallel steps.

The use of the colours on *HV15R* is certainly not optimal with regard to efficient parallel steps. For instance, the last 100 colours are only used on less than 100 columns each. The work load for the last 100 colours is too small to justify separate parallel steps, and should maybe be handled alternatively. However, the last 100 colours holds less than 0.5 % of the total columns. So an alternative solution will, in this case, most likely give little or no positive effect, except maybe reduced overhead. But if the tail contains a significant percentage of the columns, an alternative solution might pay off.

Graph / Barrier	≤ 6400	≤ 3200	≤ 1600
boneS10	15.8 %	3.1 %	< 1%
cage14	5.7 %	2.6 %	1.4 %
HV15R	20 %	8.6 %	2.9 %
Hook1498	3.2 %	1.1 %	< 1%
Flan_1565	4.2 %	1.5 %	1%
Serena	3.9 %	2 %	1.1 %

More information on the matrices is shown in Table 8.1.

Table 7.1: Table that shows % of columns in the tail for different barriers.

7.2.1.1 Identifying inefficient parallel steps and heavy tails

To identify when an alternative approach could be used, we first have to identify when a parallel step is inefficient. For the SMvM algorithms using colouring, the threads are for each colour, given a number of columns to work on. The work-load given to each thread in a parallel step is dependent on many factors, such as which algorithm is used and the input data. But if we assume that the density of non-zero elements on the columns is fairly equal, we can in general it is decided by the number of columns divided by the number of threads. If the number of columns is too small, we can assume that the parallel step does not scale well or even that parallelization is unnecessary.

"*Too small*" is a relative term, but if we know at what number of columns the parallelization becomes inefficient, it is possible to identify how many columns are in the tail. We can expect the efficiency of parallel steps to decrease gradually as the work load decreases. Therefore it is hard to find an exact barrier on where the tail starts. But by estimating a minimum amount of work needed for an efficient parallel computation, we can determine which matrices have a substantial tail. Table 7.1 shows for a set of matrices how large percentage of the columns that have colours that are used less than defined barriers.

7.2.2 How to handle an inefficient step.

An easy alternative solution is not to use extra resources on the tail and handle it sequentially. This will remove any speed-up from parallelization on the tail. Because of the restructuring of the matrix after the colouring, we know that the tail is in the last part of the matrix, thus a sequential solution for the tail is trivial to add.

7.2.2.1 Basic CCS-algorithms

The idea of removing the parallel steps for each colour in the tail and reducing it to one step can also be applied with the basic parallel CCS-algorithms. As discussed in Chapter 4, AP12 shows that CCS with locking (CCS1: ALG 7) gives substantial speed up for almost all degrees of parallelism and often scales better than CCS + colouring (ALG 9). CCS with private out-vectors (CCS2: ALG 8) also gives good speed up, but only up to 8 or 16 threads. The question is whether they give better speed-up on the tail than CCS with colouring.

CCS1 uses locking which in itself may prevent speed-up and on the tail this might be even more severe. The tail consists of the colours that are used on few columns and it often contains a significant number of the total colours. With many colour classes, we can expect a high number of non-zero elements in the same positions for these columns. This may in turn lead to write conflicts and use of the locks, potentially reducing efficiency and speed-up.

The threads in CCS2 write into private output vectors. The negative factors of this algorithm is more memory use and an extra step for adding together the private output vectors. In AP12, CCS2 gave no speed-up when applying more than 8 or 16 threads, and we can expect similar results here.

7.2.3 Perfect parallel steps

The tail may give severe problems and reduce overall performance, even when an alternative approach is used. Reducing or removing it may let us handle the whole matrix efficiently, and getting timings even closer to the CRS-algorithm. This can for instance be done by changing the colouring algorithm.

7.2.3.1 Random colouring

If each column gets a randomly chosen colour from a predetermined set of colours we can expect that each colour will be used more evenly. Each colour will be used about $\frac{n}{numColour}$ times. Depending on the barrier for the tail is higher or lower than $\frac{n}{numColour}$, either all the data is in the tail or none of it is.

An algorithm choosing colours at random may give an even use of the colours, but we also have to guarantee that neighbouring vertices are given different colours. If not,write conflicts may occur. But if we instead choose a random colour from the colours that are not used on the neighbours, we may still expect a fairly even colour distribution. A random colouring algorithm could be based on the greedy distance-2 colouring (ALG 11), but instead of choosing the smallest possible colour, a random one is chosen. This is shown in ALG 12.

The random colouring should remove the problems encountered in the tail. Thus we can expect better timings and speed-ups than with the greedy colouring. *numColour* can be chosen as the lowest possible number that yields a plausible solution. This can be decided by a binary search. However, if *numColour* is too large, we may run into problems. The iteration over the colours will be longer and may in itself prove more costly. And more seriously, using many colours will make the work load for each colour smaller, and this may reduce the efficiency of parallelism. In the worst case, the whole computation might consist of inefficient parallel steps.

Algorithm 12: A random distance-2 colouring algorithm

Data: $G(V_1, V_2, E)$, a bipartite graph with vertices V_1 and V_2 and edges E, *numColour* an integer giving the number of colours, *forbiddenColours*, a vector of size $numColour$

Result: *colour*, a vector that holds the colouring of V_1 **or** an error message telling that $numColour$ is too small

```
Algorithm:
Initialize $forbiddenColours$ with some value $a \notin V$
for all $v \in V_1$ do
    for all $w \in N(v)$ do
        for each coloured vertex $x \in N(w)$ do
            $forbiddenColors[colour[x]] \leftarrow v$
        end for
    end for
    $colour[v] \leftarrow$ random $\{ c < numColour : forbiddenColors[c] \neq v \}$
    if no such $c$ exist, return error message
end for
```

Chapter 8

Experiments

In this chapter we present and discuss the experiments based on the ideas introduced in chapters 5 - 7. Furthermore we discuss what results we expected from the previous chapters, and how they compare to the results from the actual experiments.

The experiments are done on a multi-core computer, brake.ii.uib.no, with the following specifications:

- There are 4 sockets, each with an Intel 2.0GHz E7-4850 CPU multi-core processors.
- Each processor has 32 GB memory and can run 10 threads.
- The threads can be hyper-threaded. I.e. there is hardware support for each physical thread to switch between two logical threads.

If we allow hyper-threading, this allows for a total of 80 threads. However, high memory use may restrict efficient results to at most 40 threads.

The algorithms are implemented using C/C++ with the OpenMP [11] API for parallelism. They are compiled using the g++ compiler, with the optimization flag -O3. The test matrices are, mostly, taken from the University of Florida Sparse Matrix Collection [9]. Some of the matrices used in the tests are selected since they are used in previous work, thus we can easily compare test results. Other matrices are selected for having a structure that is interesting with regard to our research. We have also used random matrices we created our self. We created the random matrices by giving nnz elements random coordinates within the decided dimension of the matrix. The dimension and nnz were decided so that they had similar size and density to the other matrices. The random placement was decided by the srand() and rand() functions in C++ [19].

Information on the matrices used is shown in Table 8.1.

To improve the reliability of the results, all tests are done 10 times. From these we chose the results with the best times. One could argue of using the average time, but the best time is shown to often be more representative [20].

For some of the algorithms, only a selection of the test results are shown. In these cases the selected results represent the overall tendencies for all the test results. When the average improvement or speed-up is mentioned, this is the average for all the test results.

Matrix	n	nnz	Matrix	n	nnz
af_shell10	1.5M	52.7M	boneS10	0.9M	55.5M
cage14	1.5M	27.1M	cage15	5M	100M
Hamrle3	1.4M	5.5M	HV15R	2M	283M
nlpkkt_240	28M	775M	delaunay_23	0.9M	50M
delaunay_24	1.7M	100M	Serena	1.4M	64M
hook1498	1.5M	59M	Flan_1565	1.6M	114M
road_usa	24M	58M	random5_100*	5M	100M
random1_50*	1M	50M			

All matrices are of size $n \times n$ and holds nnz non-zero elements, the sizes are approximate. All matrices, except for those marked*, are from the University of Florida Sparse Matrix Collection [9]. The matrices marked* are random matrices we created.

Table 8.1: Overview of the matrices used in the experiments.

8.1 Chp. 5 - Even work load

In Chapter 5. we discussed the distribution of work to the threads. We introduced a technique inspired by the min-makespan problem that should give a fairly even work load to the threads. Furthermore, we discussed whether the even distribution made by the basic COO-algorithm would yield the best results on unfortunate matrices, like arrowhead matrices.

8.1.1 Improved distribution on CRS-algorithms

In Chapter 5. we showed that distributing rows or columns by an algorithm using binary search could give a much more even distribution of the non-zero elements. This

Matrix	Distribution	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
af_shell10	static	0.248	0.142	0.071	0.036	0.020	0.015	0.022
	dynamic	0.271	0.448	0.550	0.475	0.471	0.449	0.382
	guided	0.250	0.134	0.069	0.035	0.022	0.018	0.025
	bin.search	0.250	0.138	0.071	0.036	0.019	0.015	0.017
HV15R	static	1.337	0.683	0.364	0.190	0.111	0.096	0.105
	dynamic	1.363	1.111	0.740	0.541	0.602	0.621	0.486
	guided	1.337	0.706	0.364	0.185	0.108	0.105	0.122
	bin.search	1.341	0.636	0.305	0.173	0.101	0.087	0.111
nlpkkt_240	static	3.696	1.886	0.988	0.590	0.382	0.282	0.369
	dynamic	4.113	7.185	9.625	9.099	8.692	8.266	6.934
	guided	3.711	1.893	0.977	0.547	0.332	0.307	0.398
	bin.search	3.796	1.908	1.028	0.577	0.221	0.198	0.370
random5_100	static	1.729	0.797	0.394	0.206	0.124	0.111	0.121
	dynamic	2.392	1.971	1.361	1.225	1.221	1.224	0.782
	guided	1.741	0.907	0.391	0.205	0.121	0.101	0.133
	bin.search	1.733	0.810	0.407	0.210	0.126	0.112	0.119
road_usa	static	0.710	0.524	0.307	0.203	0.154	0.133	0.119
	dynamic	1.456	3.988	4.752	5.274	8.769	11.452	7.395
	guided	0.766	0.443	0.260	0.152	0.099	0.091	0.084
	bin.search	0.721	0.470	0.347	0.213	0.212	0.207	0.209

static and *guided* are OpenMP scheduling schemes. All times are measured in seconds.

TABLE 8.2: Selected results on the CRS-algorithm with different distribution schemes.

will make the threads finish their work at about the same time. Thus we may expect better results on algorithms with this scheduling than on algorithms using the scheduling techniques in OpenMP. At least when an even work load is important for the final run-time. The dynamic and especially the guided scheduling might give an uneven distribution of non-zero elements. However, they allocate new work once each thread is finished, thus reducing the time threads are idle. If the on-the-go allocation is not too expensive, algorithms with dynamic and guided scheduling might also give good results.

Table 8.2 shows selected results for the CRS-algorithm with different scheduling techniques. It shows that the CRS-algorithm using binary search inspired by min-makespan to distribute elements indeed do get better results on some matrices. Improvements can especially be seen when using 16 or more threads. On *nlpkkt_240* it shows an approximate 30% improvement in time compared to static scheduling when using 32 threads. However, the results are often more similar, and for *road_usa*, they are worse. If we disregard *road_usa*, CRS with binary search was in average 8 % faster with $p = 16$ and 11 % faster with $p = 32$, compared to CRS with guided.

The CRS-algorithm with dynamic scheduling gave very bad results. For all the matrices

Matrix	Algorithm	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
delaunay_n23	CRS (static)	0.362	0.238	0.139	0.092	0.075	0.079
	CRS (dynamic)	0.540	1.506	1.746	1.858	3.106	3.571
	CRS (guided)	0.378	0.266	0.147	0.094	0.074	0.064
	CRS (binary search)	0.366	0.218	0.126	0.084	0.084	0.084
	COO	0.479	0.268	0.150	0.087	0.062	0.060
delaunay_n24	CRS (static)	0.728	0.422	0.351	0.265	0.241	0.235
	CRS (dynamic)	1.132	2.777	3.511	3.716	6.147	7.171
	CRS (guided)	0.742	0.439	0.249	0.197	0.148	0.122
	CRS (binary search)	0.733	0.463	0.299	0.203	0.201	0.203
	COO	0.738	0.475	0.295	0.190	0.131	0.107

TABLE 8.3: Results on arrowhead matrices

the times registered are much higher than for the other scheduling schemes. Furthermore, in some cases it actually gave worse results when going in parallel. The reason for this is probably the heavy on-the-go allocation and false sharing.

8.1.2 Arrowhead Matrices

In Chapter 5 we discussed the distribution given by different scheduling techniques on arrowhead matrices. The results in Table 8.3, shows that the COO-algorithm gives better execution times than the CRS-algorithms when applying 8 and more threads. Furthermore, adding more threads almost always gives speed-up, but the efficiency decreases when the number of threads increase. In Chapter 5 we saw that the only algorithm that consequently gave an even distribution was the basic COO-algorithm (Table 5.2), so the better execution times are not surprising. The decreasing efficiency when more threads are used is probably related to the negative issues with the COO-algorithm.

Another observation is that CRS with guided scheduling also improves results when using 16 or 32 threads. In fact, CRS with guided scheduling achieved running times only slightly worse than the COO-algorithm. They differed the most on *delaunlay_n24* with $p = 32$, where the running time of the COO-algorithm was 12 % faster.

Matrix	Algorithm	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
af_shell10	CRS	0.250	0.138	0.071	0.036	0.019	0.015	0.017
	hybrid	0.383	0.199	0.108	0.060	0.032	0.020	0.026
HV15R	CRS	1.340	0.676	0.335	0.183	0.105	0.091	0.121
	hybrid	2.058	1.051	0.525	0.270	0.150	0.108	0.130
nlpkkt_240	CRS	3.796	1.908	1.028	0.577	0.397	0.306	0.370
	hybrid	5.627	2.959	1.367	0.747	0.472	0.301	0.454
random5_100	CRS	1.733	0.810	0.407	0.210	0.126	0.112	0.119
	hybrid	2.342	0.986	0.399	0.201	0.122	0.073	0.076
random1_50	CRS	0.468	0.242	0.126	0.079	0.045	0.029	0.032
	hybrid	0.711	0.317	0.145	0.071	0.042	0.025	0.026
road_usa	CRS	0.721	0.470	0.347	0.213	0.212	0.207	0.209
	hybrid	0.706	0.427	0.271	0.164	0.167	0.143	0.156

All times are measured in seconds.

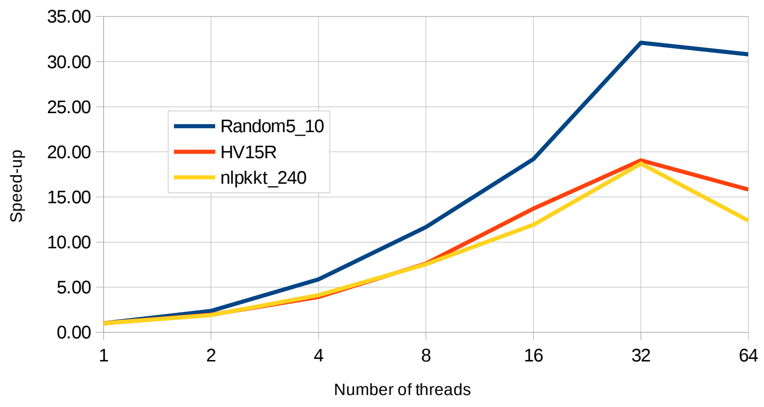
Table 8.4: Selected results with the hybrid algorithm

8.2 Chp. 6 - Efficient cache use

In Chapter 6 we argued that a hybrid algorithm using the COO-representation could be more cache efficient than algorithms using CRS- or CCS-representations. However, we also argued that the difference between the algorithms would be dependent on the structure of the matrix. In the tests, we compared the hybrid algorithm with the CRS-algorithm. Both the hybrid and the CRS-algorithm used the binary search introduced in Chapter 5 to distribute the rows.

Table 8.4 shows some results for the hybrid algorithm compared to the CRS-algorithm. Which algorithm that performs best varies between the matrices. For *af_shell10*, *HV15R* and *nlpkkt_240* the CRS-algorithm preforms better. However, the hybrid algorithm scales better. Thus the difference is smaller when the number of threads is higher. The smaller difference with higher amount of threads is probably because the hybrid algorithm has fewer cache misses on the output vector. When the section each thread writes into gets smaller, the chance of consecutive write operations on the same cache line increases.

On the random matrices, the hybrid algorithm performs better than the CRS-algorithm on 8 and more threads. On *random5_100* the hybrid algorithm is about 35 % faster on 32 threads. Furthermore, the hybrid algorithm in some cases shows *superlinear* speed-up, shown in Figure 8.1. Since the matrix is random, we can expect a high number of cache misses, which the hybrid algorithm reduces. Combined with the lower chance of cache misses when the number of threads increases, superlinear speed-up may not be surprising.



The hybrid algorithm shows *superlinear* speed-up on *random5_100*.

FIGURE 8.1: Scaling of the hybrid algorithm on different matrices.

The hybrid algorithm also preforms better on the *road_usa* matrix. In addition to the improved cache efficiency, *road_usa* is a much sparser matrix than the others, which may make the COO-representation more efficient. With n =28M and nnz =54M $= 2 \times n$, the COO-representation uses approximately $6 \times n$ space to represent the matrix, as compared to the CRS-representation, which uses approximately. $5 \times n$.

8.3 Chp. 7 - Improving CCS with colouring

In Chapter 7, possible improvements to the CCS-algorithm using colouring, presented in Azad and Pothen [7], were discussed. It was mainly concerned with identifying the possible problems CCS with colouring would meet, and how to handle them. Specifically it discussed reduction of false sharing, alternative handling of ineffective parts, and alternative colouring algorithms.

8.3.1 Replicating results

To get an idea how the new algorithms presented in this thesis actually compares to the ones presented in Azad and Pothen, we first tried to replicate their results. We did this by implementing the algorithms and running them on similar test data. Furthermore, we ran the tests on the same machine used in Azad and Pothen.

8.3.1.1 AP12 - Algorithms

Tests were done with the replicated versions of CRS (ALG 5), CCS with locking (ALG 7), CCS with private out vectors (ALG 8) and CCS with colouring (ALG 9). On CCS with colouring the greedy colouring algorithm (ALG 10) was used. For all but two of the matrices the colouring algorithm gave the same amount of colours, but for *cage15* and the random algorithm the number of colours differed. On the random matrix this is not strange, as the random matrix used in AP12 is most likely different from the one used in our test. On *cage15* however, the different colouring can point towards the use of another colouring algorithm.

The replicated results are shown in Table 8.5. They show similar results as in Azad and Pothen on the CRS-algorithm, CCS with locking and CCS with colouring, both in timing and scaling. CCS with private out vectors gave better results when using high degrees of parallelism on compared to Azad and Pothen. But like in Azad and Pothen the results gets worse when applying more than 16 threads. The generally similarity between our results and those shown in Azad and Pothen is important. It gives us a better possibility to argue for possible better results on alternative algorithms represent actual improvements.

Matrix	Algorithm	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
af_shell10	CRS	0.250	0.138	0.070	0.037	0.020	0.016	0.021
	CCS-locking	0.646	0.322	0.162	0.085	0.045	0.031	0.033
	CCS-private	0.272	0.162	0.083	0.047	0.032	0.053	0.105
	CCS-colour (50)	0.297	0.136	0.070	0.036	0.022	0.023	0.029
boneS10	CRS	0.262	0.132	0.072	0.037	0.021	0.017	0.023
	CCS-locking	0.674	0.345	0.174	0.091	0.049	0.031	0.037
	CCS-private	0.279	0.159	0.086	0.045	0.033	0.028	0.060
	CCS-colour (129)	0.287	0.149	0.079	0.046	0.036	0.040	0.045
cage14	CRS	0.138	0.087	0.047	0.024	0.015	0.010	0.008
	CCS-locking	0.343	0.197	0.193	0.105	0.062	0.033	0.024
	CCS-private	0.156	0.097	0.059	0.034	0.038	0.038	0.096
	CCS-colour (136)	0.198	0.118	0.065	0.043	0.031	0.034	0.036
cage15	CRS	0.521	0.339	0.183	0.098	0.056	0.045	0.055
	CCS-locking	1.282	0.842	0.545	0.426	0.225	0.119	0.085
	CCS-private	0.586	0.404	0.220	0.135	0.121	0.129	0.311
	CCS-colour (165)	1.272	0.715	0.306	0.162	0.106	0.086	0.100
Hamrle3	CRS	0.038	0.019	0.010	0.006	0.004	0.003	0.004
	CCS-locking	0.117	0.081	0.044	0.022	0.011	0.006	0.004
	CCS-private	0.047	0.037	0.021	0.014	0.013	0.021	0.087
	CCS-colour (8)	0.035	0.019	0.009	0.005	0.004	0.003	0.004
HV15R	CRS	1.337	0.673	0.353	0.185	0.105	0.090	0.105
	CCS-locking	3.447	1.999	1.177	0.472	0.228	0.136	0.136
	CCS-private	1.388	0.699	0.389	0.219	0.148	0.145	0.242
	CCS-colour (508)	2.286	0.954	0.561	0.332	0.231	0.192	0.211
nlpkkt_240	CRS	3.699	1.895	0.985	0.545	0.316	0.286	0.370
	CCS-locking	9.469	5.108	2.672	1.631	0.895	0.485	0.471
	CCS-private	4.089	2.280	1.230	0.839	0.730	0.925	1.345
	CCS-colour (59)	5.203	2.649	1.310	0.775	0.586	0.671	0.608
random5_100	CRS	1.738	0.781	0.376	0.195	0.122	0.116	0.120
	CCS-locking	4.593	1.956	0.844	0.431	0.285	0.173	0.152
	CCS-private	1.831	0.934	0.482	0.353	0.318	0.375	0.612
	CCS-colour (97)	1.858	0.900	0.466	0.249	0.160	0.178	0.194

Number of colours is shown in the parenthesis, all times are measured in seconds.

TABLE 8.5: Replicated results from AP12

8.3.2 Reducing false sharing

To reduce false sharing, we introduced the idea of having a stricter colouring algorithm. This algorithm demanded that the row index distance between elements of same colour should increase. It was argued that this would lead to reduced false sharing when writing to the output vector. However, several disadvantages were also discussed. The stricter colouring would probably lead to using more colours. Also there would be less consecutive data, making the program less cache efficient.

The results in Table 8.6 show that when the distance increased, the number of colours

Matrix	Distance	#colours	p = 1	p = 2	p = 4	p = 8	p = 16	p = 32	p = 64
cage14	dist = 1	136	0.198	0.118	0.065	0.043	0.031	0.034	0.036
	dist = 2	264	0.217	0.147	0.089	0.051	0.037	0.042	0.046
	dist = 3	382	0.228	0.153	0.089	0.052	0.038	0.047	0.049
	dist = 5	581	0.240	0.158	0.090	0.053	0.039	0.048	0.050
	dist = 10	953	0.255	0.168	0.094	0.056	0.038	0.050	0.055
HV15R	dist = 1	508	2.342	0.959	0.622	0.332	0.239	0.199	0.230
	dist = 2	712	2.250	0.951	0.588	0.333	0.237	0.215	0.242
	dist = 3	797	2.188	0.949	0.586	0.324	0.233	0.221	0.247
	dist = 5	1120	2.012	0.942	0.573	0.319	0.235	0.232	0.273
	dist = 10	1223	2.042	0.933	0.576	0.315	0.234	0.242	0.284
nlpkkt_240	dist = 1	59	5.403	2.799	1.441	0.926	0.636	0.702	0.788
	dist = 2	77	5.689	2.926	1.569	1.007	0.999	1.184	1.195
	dist = 3	103	6.571	3.325	1.725	1.103	1.090	1.286	1.277
	dist = 5	147	7.829	4.181	2.102	1.351	1.296	1.499	1.489
	dist = 10	239	8.895	5.045	2.564	1.607	1.565	1.819	1.733

All times are measured in seconds.

Table 8.6: Selected results from CCS-col with increased distance.

increased. However, the times achieved with increased distance generally got worse. Only on the matrix *HV15R*, they stayed about the same. From these observations we may argue that the effect from the reduction of false sharing is small compared to the effects from decreased cache efficiency and an increased number of parallel steps.

8.3.3 Efficient parallel sections

To test whether the tail indeed does scale worse than the rest of the matrix we selected matrices we believed to have heavy tails. From the measurements on tail size done with different barriers we expected the matrices *boneS10*, *cage14*, and *HV15R* (Table 7.1) from Azad and Pothen to be affected the most by the tail. The other matrices have a tail that contains at most 1% of the total columns, so we would not expect much improvement on thesefrom these. To get a bigger sample size, three other matrices were also chosen, *Serena*, *Flan_1565* and *Hook1498*.

8.3.4 Decreased efficiency on the tail

The experiments tested the time used on the tail, with different barriers, and the time used on the rest of the matrix. It was then possible to see the difference in scaling on the two parts, as well as the percentage of total time used. The results, shown in Table 8.7 and Figure 8.2, shows the speed-up on the tail compared to the rest of the matrix. They show that the speed-up on the tail is substantially smaller than on the good part, or on the matrix as a whole. The algorithm scaled worse on the tail than on the rest of

Matrix	Part	p = 1	p = 2	p = 4	p = 8	p = 16	p = 32	p = 64
HV15R	Total	1.00	2.30	3.53	6.73	9.50	11.61	10.02
	Good	1.00	2.35	3.59	6.99	9.92	12.59	11.17
	Tail	1.00	1.46	2.33	3.15	4.08	3.36	2.38
Flan_1565	Total	1.00	2.35	4.22	7.74	11.76	11.09	8.30
	Good	1.00	2.36	4.25	7.80	11.97	11.44	8.53
	Tail	1.00	1.96	3.12	4.76	5.06	3.42	2.75

barrier = 1600, shows speed-up on CCS with colouring on the different part of the matrices.

TABLE 8.7: Selected results showing speed-up on different part of the matrices

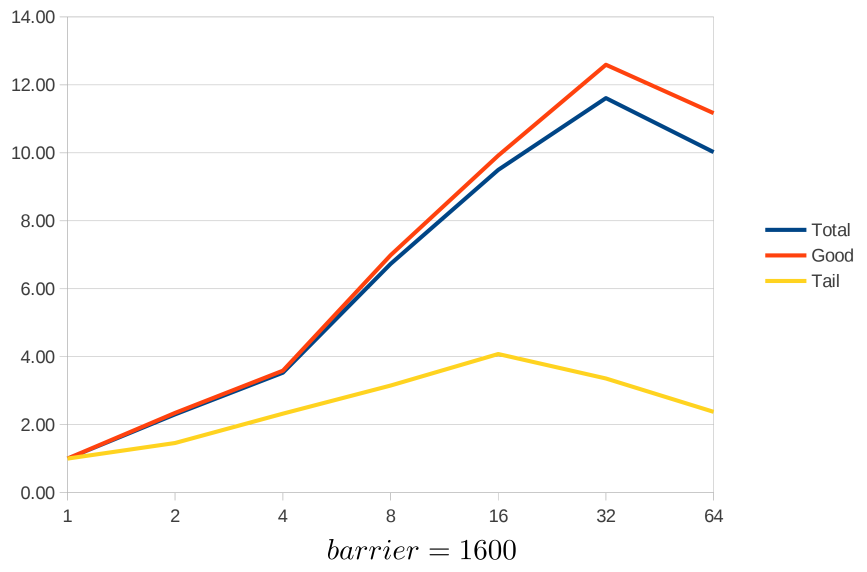


FIGURE 8.2: Scaling on the good part and on the tail, with CCS with colouring, for *HV15R*

the algorithm and especially so on many threads. These tendencies are stronger when the barrier defining the tail is smaller. Figure 8.3, show that for smaller work loads, the efficiency due to parallelism will also decrease.

8.3.4.1 Alternative handling of the tail

To handle the less efficient parallelism on the tail, we replaced the iteration over the colours on the tail with alternative algorithms. Three techniques were tried:

- A sequential CCS-algorithm.
- CCS with locking (ALG 7).
- CCS with private out-vectors (ALG 8).

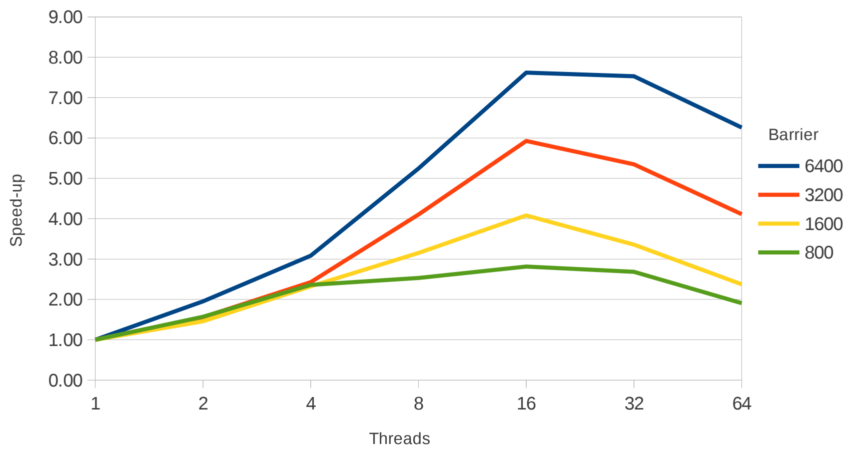
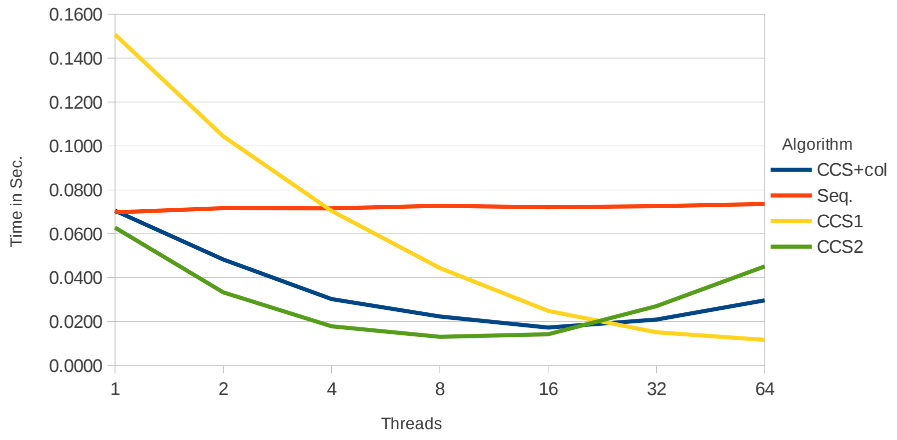


Figure 8.3: Scaling on the tail, with CCS with colouring, when using different barriers on HV15R

Selected results for the whole algorithms are shown in Table 8.8, while the results for the tail of the *HV15R* matrix can be seen in Figure 8.4. As expected by the fact that all algorithms handled the good part in the same way, no significant differences were observed here. On the tail however, the alternative solutions gave different results and we could make some observations.

- The sequential solution gave about the same results as CCS + colouring with one thread.
- CCS with locking had worse running times than CCS + colouring when tested with a few threads, but scaled much better. For 32 and 64 threads it mostly performed better than the other algorithms, but only slightly better than CCS + colouring. In the best cases it performed about 10 % better.
- CCS with private out vectors gave results close to CCS + colouring on less than threads, but gave much worse results when the number of threads increased. This is similar to the results observed in AP12, were CCS with private out vectors did not improve beyond more than 8 or 16 threads.



$barrier = 1600$

FIGURE 8.4: Run times on different solutions on the tail of HV15R

Matrix	Algorithm	p = 1	p = 2	p = 4	p = 8	p = 16	p = 32	p = 64
HV15R	Greedy col(508)	2.328	0.973	0.619	0.329	0.247	0.210	0.238
	Greedy+Lock	2.349	1.021	0.657	0.361	0.253	0.199	0.226
	Greedy+Private	2.256	1.068	0.707	0.370	0.242	0.220	0.256
	Random(700)	2.222	1.032	0.621	0.334	0.240	0.258	0.320
Serena	Greedy col(249)	0.374	0.220	0.155	0.104	0.081	0.081	0.084
	Greedy+Lock	0.386	0.249	0.166	0.106	0.086	0.075	0.074
	Greedy+Private	0.382	0.278	0.167	0.118	0.101	0.102	0.138
	Random col(300)	0.441	0.225	0.152	0.102	0.085	0.094	0.098
Cage14	Greedy col(136)	0.203	0.114	0.063	0.050	0.031	0.032	0.036
	Greedy+Lock	0.202	0.141	0.083	0.048	0.036	0.031	0.035
	Greedy+Private	0.208	0.170	0.089	0.054	0.050	0.054	0.109
	Random col(200)	0.208	0.141	0.085	0.041	0.031	0.033	0.038
BoneS10	Greedy col(129)	0.292	0.153	0.079	0.052	0.036	0.041	0.045
	Greedy+Lock	0.308	0.167	0.107	0.061	0.042	0.042	0.049
	Greedy+Private	0.291	0.191	0.098	0.059	0.047	0.062	0.102
	Random col(150)	0.276	0.141	0.088	0.050	0.040	0.051	0.066
Flan_1565	Greedy col(153)	0.761	0.320	0.178	0.104	0.066	0.068	0.097
	Greedy+Lock	0.774	0.332	0.187	0.102	0.067	0.060	0.084
	Greedy+Private	0.762	0.418	0.211	0.119	0.077	0.080	0.168
	Random col(200)	0.760	0.318	0.187	0.106	0.068	0.070	0.104
Hook1498	Greedy col(108)	0.352	0.205	0.122	0.070	0.053	0.055	0.071
	Greedy+Lock	0.355	0.210	0.113	0.073	0.054	0.055	0.067
	Greedy+Private	0.365	0.252	0.132	0.083	0.069	0.075	0.132
	Random col(140)	0.376	0.176	0.089	0.059	0.039	0.042	0.051

$barrier = 1600$, number of colours is shown in the parenthesis, all times are measured in seconds.

TABLE 8.8: Selected results for alternative handling of the tail and for random colouring.

8.3.4.2 Random colouring algorithms

We expected CCS with random colouring (colouring by ALG 12) to give better run times than CCS with greedy colouring when using a low number of colours. For high number of colours, the work load on each parallel step will be too small. Thus we expect CCS with random colouring to be less efficient and possibly have worse times than CCS with greedy colouring.

We implemented the random colouring algorithm by storing the matrix using both CCS-representation and CRS-representation. This allowed us to get an easy look-up of which columns had an element on the same row. However, the storage used was twice as big. The extra storage was only used in the preprocessing step and should thus not affect the measured run-times of the parallel SMvM. We used the srand() and rand() functions from C++ [19] to randomly choose a possible colour.

In Chapter 7 we suggested using a binary search to find an optimal number of colours for the random colouring. However, this proved infeasible. For all matrices the random colouring used much more time than the greedy colouring. This was in the scale of hours or days vs. seconds or minutes. These times discouraged running the colouring several times, like we would have to in a binary search. Thus we chose the number of colours more loosely. The number of colours used and how many columns that were coloured by each colour class is shown in Table 8.9.

The results in Table 8.8 shows that CCS with random colouring gets the biggest improvement on the matrix *Hook1498*, where each colour was used approximately 10700 times. The times used by CCS with the greedy and random colouring on this matrix is illustrated in Figure 8.5. For matrices *Flan_1565*, *BoneS10* and *Cage14* the results were similar or marginally better compared to the results obtained from CCS with greedy colouring. And for *Serena* and *HV15R*, where the average use of each colour was lower, CCS with greedy colouring showed marginally better run times (Figure 8.6). How the results for the greedy colouring and random colouring compare can be seen in direct relation to the number of columns coloured with each colour class.

The running time of the random colouring algorithm can be analysed to $\mathrm{O}(n \times avgDeg^2)$ where $avgDeg$ is the average number of non-zero elements on a column or row. This looks manageable, even if n and nnz are very large. However, there are some factors that may create the high running time. There are hidden factors in the expression that still might demand work. These are for instance choosing a possible colour and resetting the array holding the forbidden colours. The random colouring also uses twice the amount of storage compared to the greedy colouring algorithm, which might lead to a slower computation [3].

Matrix	n	$numColour$	$\frac{n}{numColour}$
boneS10	0.9M	150	6000
cage14	1.5M	200	7500
HV15R	2M	700	$\approx$ 2857
Serena	1.4M	300	$\approx$ 4666
Flan_1565	1.6M	200	7500
Hook1498	1.5M	140	$\approx$ 10714

TABLE 8.9: $\frac{n}{numColour}$ for the six matrices coloured with a random colouring

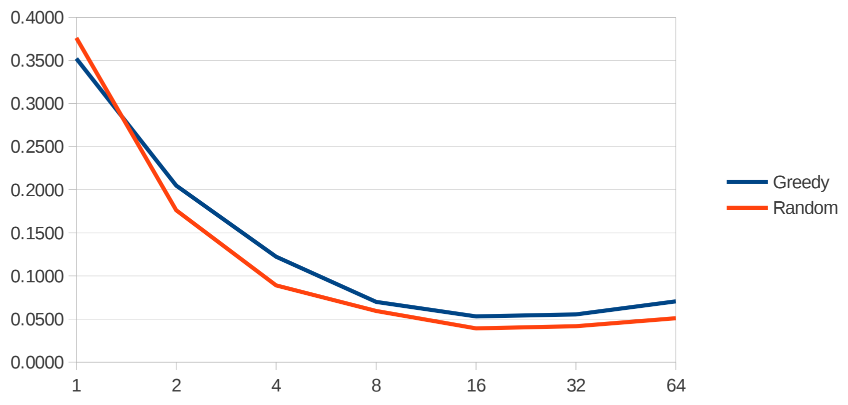


FIGURE 8.5: Results on Hook1498 with greedy and random colouring

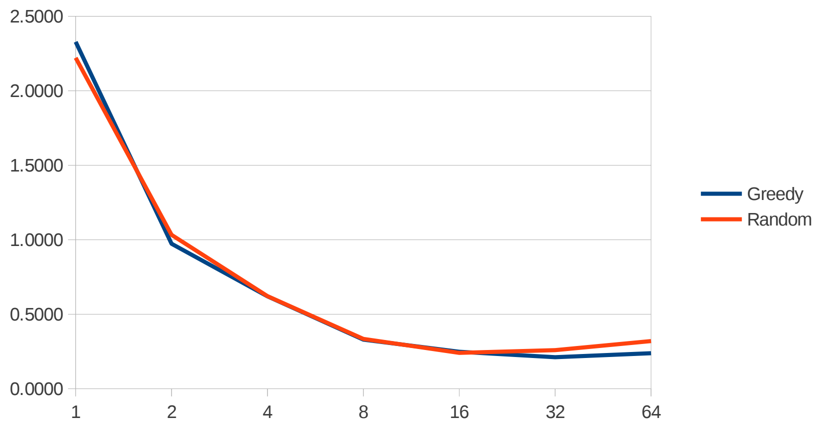


FIGURE 8.6: Results on HV15R with greedy and random colouring

Chapter 9

Conclusions

In Chapter 8 we presented and discussed the measured times and speed-ups of the ideas presented in chapters 5 - 7. In this chapter, we discuss four main conclusions that can be made from these observations. Furthermore, we discuss some ideas and issues that could be expanded further.

9.1 Even work load matters

In Chapter 5 we discussed some ideas on how to achieve even work load, and how it might affect the algorithms. We discussed a distribution decided by a binary search based on the min-makespan problem. Furthermore, we discussed handling arrowhead matrices using the basic COO-algorithm.

We observed that the CRS-algorithm with distribution inspired by min-makespan gave better times than the static and guided scheduling on some matrices (Table 8.2). The improvement was measured to be as large as 30 %. However, the running times observed were also often similar and slower than the algorithms using OpenMP scheduling techniques. Furthermore, we observed that the dynamic scheduling was inefficient, and actually negative speed-up or *slow-down*, when applying more threads. We can conclude from this that using a min-makespan inspired distribution on parallel SMvM can give positive results.

Using the COO-algorithm on arrowhead matrices sometimes gave better results than the OpenMP scheduling techniques (Table 8.3). However, the CRS algorithm with guided scheduling gave similar results. From this we can conclude that a perfect distribution of elements when working on arrowhead matrices can give large positive effects.

At least larger than the possible negative effects of using up to 50 % more memory use and handling of write conflicts.

9.2 The hybrid algorithm is only better on certain matrices

In Chapter 6 we introduced a hybrid algorithm that could be more cache efficient than the CRS or CCS algorithms, especially on random matrices. The test results showed that the hybrid algorithm performed better than the CRS algorithm on random matrices and on very sparse matrices (Table 8.4). In fact, on the random matrices it shows superlinear speed-up. On the other matrices, however, the CRS-algorithm gave the best running times.

From these observations we can draw the conclusions that the hybrid algorithm indeed is more cache efficient than the CRS-algorithm. However, the added cost of using the COO-representation reduces the improvement. Thus which algorithm is superior depends on the structure and sparsity of the matrix. This is not unlike the results achieved in Yzelman and Bisseling [4], where the results were also dependent on the structure of the matrix.

9.3 Increasing distance to reduce false sharing is not a good idea

In Chapter 7 we discussed CCS-algorithms using colouring of the columns to avoid write conflicts. To improve results by reducing false sharing, we discussed an alteration of the colouring, requiring a bigger *distance* between columns that gets the same colour. The observed results were not favourable. The tendency showed that with larger distance, the algorithm used longer running time (Table 8.6).

We discussed the possible problems of decreased efficiency on both use of cache lines and on the parallel steps. We can conclude that the negative effects from these problems are larger than the positive effects from reduced false sharing.

9.4 It is hard to handle inefficient parallel steps

Sections 7.2 of Chapter 7 discussed how to identify and improve inefficient parallel steps created by the greedy colouring algorithm. The tests showed that the parallel steps on little used colours are less efficient, and does not scale as well as the parallel steps on much used colours. To improve the inefficient steps, we tried to use alternative CCS-algorithms. We observed that replacing the inefficient steps with CCS with locking could give slightly better results when using 32 or 64 threads. Otherwise, we saw no speed-up (Table 8.8).

We also tried to remove the inefficient parallel steps by using a random colouring algorithm. The random colouring algorithm used each colour on about the same amount of columns, making each parallel step equally efficient. When the number of columns using each colour were high (on *Hook1498* about 10700) we saw speed-up with the random colouring. On matrices with fewer columns per colour, the speed-ups were smaller and on some matrices we even observed slow-down. In addition to this, the random colouring algorithm would use a very long time (hours and days), possibly removing any practical use.

From these results we can see that it is possible to improve CCS with colouring by either handling the inefficient parallel steps alternatively or removing them. However, the improvements are not necessarily very big, and may not be worth the extra cost, especially with random colouring. The results may anyway lead us to believe that some future work on improving CCS with colouring may give better results.

9.5 Future Work

The ideas and observed results and the conclusions we drew from them point towards some possible future work.

9.5.1 Approximation algorithms on min-makespan

In Chapter 5 approximation algorithms to the min-makespan problem were discussed as a way to distribute rows or columns to the threads. However, the approximation algorithms were not used as they would not allocate consecutive rows or columns. In future work it might be possible to use such algorithms to distribute rows or columns and then restructure the matrix so that the rows or columns are stored consecutively. As there are known $\frac{3}{2}$-approximations to min-makespan [2], this might give good results. Furthermore, there has been done much research on the min-makespan problem that might prove useful for achieving an even work load.

9.5.2 A hybrid algorithm using less storage

In Chapter 6 we argued that a possible negative factor of the hybrid algorithm was the increased use of storage from using COO-representation. This decreases the size of the matrices we can work on, and might even slow the program down. It should be possible to find a more space efficient representation that still allows us to run the hybrid algorithm. One idea could be to use CCS-representation locally on the threads. This would give a total storage of $2 \times nnz + n \times p$ elements, as we need one IA-array for each thread. Compared to the storage use of the COO-representation, $3 \times nnz$, which of the representations is largest depends on both the density of the matrix and the number of threads used.

9.5.3 Faster and better colouring algorithms

When using the random colouring algorithm to improve CCS with colouring we met two problems. The first was the execution speed of the algorithm. The matrices used in the tests were very large with dimension between 900.000 and 28 million and nnz between 5.5 million and 775 million (Table 8.1). However, using hours and days to colour the columns is not acceptable. If a faster algorithm that gives similar colourings can be constructed, we may be able to justify using it for a preprocessing. This can be done by either looking at other algorithms or improving the implementation (or both).

The other problem was that the random colouring algorithm used too many colours. This gives less work in each parallel step, making them less efficient. By looking at other algorithms, maybe heuristics or approximations to the min-colouring problem, we might find an algorithm that uses less colours. Thus we can expect each parallel step to be more efficient, and thus get better speed-up.

Bibliography

[1] Barry Wilkinson and Michael Allen,
Parallel Programming, techniques and applications using networked workstations and parallel computers.
Pearson Education, 2nd Edition 2005.

[2] Jon Kleinberg and Éva Tardos,
Algorithm Design
Pearson Education 2006

[3] Albert-Jan N. Yzelman,
Fast sparse matrix-vector multiplication by partitioning and reordering.
2011.

[4] Albert-Jan N. Yzelman and Rob H. Bisseling,
Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods.
SIAM Journal on Scientific Computing **31** (2009), no. 4, 3128-3154.
Slides found on: *http://people.cs.kuleuven.be/albert-jan.yzelman/slides/csc11.pdf*

[5] Ali Pinar and Michael T. Heath,
Improving Performance of Sparse Matrix-Vector Multiplication
Supercomputing '99 Proceedings of the 1999 ACM/IEEE conference on Supercomputing, Article No. 30

[6] Richard W Vuduc and Hyun-Jin Moon,
Fast sparse matrix-vector multiplication by exploiting variable block structure.
High Performance Computing and Communications. Springer Berlin Heidelberg, 2005. 807-816.

[7] Ariful Azad and Alex Pothen,
Parallel sparse matrix vector multiplication.
2012

[8] Assefaw Gebremedhin, Alex Pothen and Fredrik Manne,
What Color is your Jacobian? Graph Coloring for Computing Derivatives
SIAM Review, Vol 47, No 4 (2005), 629-705

[9] Timothy Davis and Yifan Hu,
The University of Florida Sparse Matrix Collection.
ACM Transactions on Mathematical Software (TOMS) 38.1 2011

[10] Tim Mattson, Mark Bull and Michael Wrinn,
A Hands-On Introduction to OpenMP
http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

[11] www.openmp.org

[12] Zhaojun Bai
Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide.
SIAM, 2000.

[13] Gene H. Golub and Charles F. Van Loan
Matrix Computations
Johns Hopkins Studies in Mathematical Sciences 3rd Edition, 1996

[14] Gene M. Amdahl
Validity of the single processor approach to achieving large scale computing capabilities
Proceedings of the April 18-20, 1967, spring joint computer conference
ACM, 1967

[15] Donald Hearn and M. Pauline Baker
Computer Graphics with OpenGL
Pearson Education 3rd Edition
2004

[16] Nathan Bell and Michael Garland.
Efficient sparse matrix-vector multiplication on CUDA
Vol. 20. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[17] Nathan Bell and Michael Garland
Implementing sparse matrix-vector multiplication on throughput-oriented processors
Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.
ACM, 2009.

[18] Lynn E. Cannon
A Cellular Computer To Implement The Kalman Filter Algorithm.
No. 603-Tl-0769. MONTANA STATE UNIV BOZEMAN ENGINEERING RESEARCH LABS
1969.

[19] www.cplusplus.com

[20] Magne Haveraaen and Hogne Hundvebakke,
Some Statistical Performance Estimation Techniques for Dynamic Machines
Norsk informatikkonferanse NIK'2001 176-185
Tromsø 2001