



UPPSALA  
UNIVERSITET

UPTEC F 14029

Examensarbete 30 hp  
Juni 2014

# Parallel Graph Coloring

Parallel graph coloring on multi-core CPUs

---

Per Normann



UPPSALA  
UNIVERSITET

Teknisk- naturvetenskaplig fakultet  
UTH-enheten

Besöksadress:  
Ångströmlaboratoriet  
Lägerhyddsvägen 1  
Hus 4, Plan 0

Postadress:  
Box 536  
751 21 Uppsala

Telefon:  
018 – 471 30 03

Telefax:  
018 – 471 30 00

Hemsida:  
<http://www.teknat.uu.se/student>

## Abstract

### Parallel graffärgning

### Parallel Graph Coloring

---

*Per Normann*

In recent times an evident trend in hardware is to opt for multi-core CPUs. This has lead to a situation where an increasing number of sequential algorithms are parallelized to fit these new multi-core environments. The greedy Multi-Coloring algorithm is a strictly sequential algorithm that is used in a wide range of applications. The application in focus is on decomposition by graph coloring for preconditioning techniques suitable for iterative solvers like the Krylov Subspace and Multigrid methods. In order to perform all phases of these iterative solvers in parallel the graph analysis phase needs to be parallelized. Albeit many attempts have been made to parallelize graph coloring non of these attempts have successfully put the greedy Multi-Coloring algorithm into obsolescence.

In this work techniques for parallel graph coloring are proposed and studied. Quantitative results, which represent the number of colors, confirm that the best new algorithm, the Normann algorithm, is performing on the same level as the greedy Multi-Coloring algorithm. Furthermore, in multi-core environments the parallel Normann algorithm fully outperforms the classical greedy Multi-Coloring algorithm for all large test matrices.

With the features of the Normann algorithm quantified and presented in this work it is now possible to perform all phases of iterative solvers like Krylov Subspace and Multigrid methods in parallel.

Handledare: Dimitar Lukarski  
Ämnesgranskare: Jarmo Rantakokko  
Examinator: Tomas Nyberg  
ISSN: 1401-5757, UPTEC F 14029

## Table of Content

1 Introduction.....	5
1.1 Purpose.....	5
1.2 Sparse Matrices.....	6
1.3 Sparse Format.....	6
1.4 Numbering.....	7
1.5 Graph Coloring.....	8
2 Parallel Graph Coloring.....	15
2.1 General Aspects in Parallel Graph Coloring.....	15
2.2 Local Parallelism.....	15
2.3 Global Parallelism.....	16
2.4 Luby Jones.....	19
2.5 New Methods.....	20
3 Implementation and Test Configuration.....	27
3.1 Benchmark Set-up.....	27
3.2 Sequential Implementations.....	28
3.3 Parallel Implementations.....	29
3.4 Verification.....	30
4 Results.....	31
4.1 Overview.....	31
4.2 Color Quality.....	32
4.3 Color Consistency.....	33
4.4 Execution Time.....	35
4.5 Parallel Speed-up.....	36
4.6 Parallelism.....	37
4.7 Assessment of Conflict Resolution.....	37
4.8 The Maximum Degree Criterion.....	38
5 Summary.....	41
5.1 Benchmarks.....	41
5.2 High Quality Parallel Graph Coloring.....	42
5.3 Conclusions.....	42
5.4 Future Work.....	42
6 Acknowledgments.....	43
7 References.....	45



# 1 Introduction

## 1.1 Purpose

The primary goal of this project is to study parallelism in graph coloring algorithms. A secondary goal is to develop new parallel graph coloring algorithms that can be benchmarked against state of the art graph coloring methods.

### 1.1.1 Main Application

In numerical analyzing and modeling software Partial Differential Equations (PDE) describing the modeled system are solved. Generally PDE solvers produces linear systems of equations that have to be solved to obtain a result. The two predominant numerical solvers of linear systems are the Krylov Subspace and Multigrid methods.

The Krylov Subspace and Multigrid methods are iterative solvers based on preconditioning and smoothing techniques [1][2][3]. Preconditioning like Incomplete LU (ILU) factorization and smoothing techniques like Gauss-Seidel are used to accelerate Krylov Subspace and Multigrid methods. Decomposition by graph coloring is needed in the preconditioning phase in order to solve the preconditioned system in parallel [2]. All phases of iterative solvers but the graph analysis are generally parallelized. In order to obtain parallelism throughout the solution phase of iterative solvers the graph analysis also needs to be parallelized. Graph coloring is however hard to parallelize due to the difficulty in separating vertex data without creating conflicting colorings. Graph coloring is also NP-complete or NP-hard depending on how a graph is traversed. These features makes it hard to obtain sufficient parallel graph coloring with maintained quality and gained speed-up. In order to justify the parallelization of the graph coloring phase in iterative solvers a parallel graph coloring algorithm needs to outperform the sequential greedy Multi-Coloring algorithm on multi-cores. The greedy Multi-Coloring [4] algorithm is the predominant graph coloring technique in contemporary iterative solvers.

### 1.1.2 Historical Context

Graph coloring dates back the nineteenth century when map makers starts to color nations in maps [5]. In the twentieth century graph coloring is focused in the first major computer aided proof when the Four Color Theorem is proven [6]. In contemporary numerical science graph coloring is implemented in a string of different applications in addition to the usage as preconditioning tool in Krylov Subspace and Multigrid methods. Some examples of these additional applications are graph traversing, the assembly of stiffness matrices and graph partitioning.

### 1.1.3 Research Ethics

In this work the test data used is chosen because it is commonly found in contemporary work presented in the field of parallel graph coloring. This enables a fair comparison between established graph coloring algorithms and the proposed algorithms. The providers of the test matrices<sup>1</sup> are presenting these matrices as naturally occurring in simulations of various phenomena. In most cases the test data used covers a wider range of sizes, structures etc than is accustom in similar contemporary works. This choice of test data is to ensure relevant benchmarks.

All implementations are done in the same software<sup>2</sup> as an attempt to have the same test environment for all implemented graph coloring algorithms. The data of test results are derived and monitored in the same way. The verification of all results are done by the same control function.


To mitigate the effects of random elements in some of the tested algorithms the best execution time

<sup>1</sup> <http://math.nist.gov/MatrixMarket/>, <https://www.cise.ufl.edu/research/sparse/matrices/>

<sup>2</sup> Paralution 0.5.0, <http://www.paralution.com/>

## 1.2 Sparse Matrices

a



b

$$A = \begin{pmatrix} 1 & 0 & 2 & 3 & 0 \\ 0 & 4 & 0 & 5 & 0 \\ 0 & 6 & 7 & 8 & 0 \\ 0 & 0 & 9 & 10 & 11 \\ 0 & 0 & 0 & 0 & 12 \end{pmatrix}$$

The graph consists of 16 nodes labeled 1 through 16. The connections between the nodes are as follows:

- Node 1 is connected to nodes 9, 7, and 10.
- Node 4 is connected to nodes 9 and 15.
- Node 9 is connected to nodes 1 and 4.
- Node 7 is connected to nodes 1, 15, and 12.
- Node 15 is connected to nodes 4, 7, 11, and 12.
- Node 11 is connected to nodes 15 and 3.
- Node 3 is connected to nodes 11 and 8.
- Node 12 is connected to nodes 7, 15, 6, and 14.
- Node 6 is connected to nodes 12 and 13.
- Node 13 is connected to nodes 6 and 8.
- Node 8 is connected to nodes 3 and 13.
- Node 10 is connected to nodes 1 and 14.
- Node 14 is connected to nodes 10, 12, 2, and 5.
- Node 2 is connected to nodes 12, 14, and 16.
- Node 16 is connected to nodes 2 and 5.
- Node 5 is connected to nodes 10, 14, and 16.

---

3 Gullviva at the Division of Scientific Computing at Uppsala University

## 1.3 Sparse Format

In most cases it is beneficial to compress the data of a sparse matrix before any calculations are done. A comprehensive review of different compression schemes can be found in [7].

### 1.3.1 Coordinate Format

One way of storing a sparse matrix is the Coordinate Format [8]. All non zero elements of a sparse matrix are stored in an array `val`. To keep track of the position of each element two additional integer arrays, `col` and `row`, holds the coordinates for each corresponding element. If the sparse matrix in Figure 1.1.b. is stored using this scheme the result is as follows:

```
val = (1 2 3 4 5 6 7 8 9 10 11 12)
row = (1 1 1 2 2 3 3 3 4 4 4 5)
col = (1 3 4 2 4 2 3 4 3 4 5 5)
```

### 1.3.2 Compressed Sparse Row

In [8] it is stated that the most common way of storing sparse matrices in a compressed order is the Compressed Sparse Row pattern. In the Compressed Sparse Row compression all non zero elements are stored in a value array `val`. The positioning is done by two additional integer arrays. The row positioning array `ptr` points out the column placement of each element by pointing at the first indices of each row in the column array `col`. The following data is a Compressed Sparse Row compression of the sparse matrix in Figure 1.1.b.

```
val = (1 2 3 4 5 6 7 8 9 10 11 12)
col = (1 3 4 2 4 3 4 5 2 4 5 5)
ptr = (1 4 6 9 12 13)
```

## 1.4 Numbering

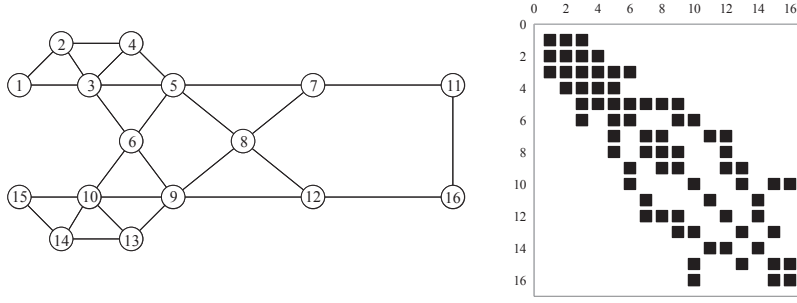
The vertices in the wrench like mesh visualized in Figure 1.2 are randomly numbered. In contemporary numerical differential equation solvers the meshes are ordered by using refined schemes. Ordered in the sense of how vertices are numbered in a general mesh. A goal of an ordering of vertices in a mesh is often to concentrate elements in the resulting matrix along the diagonal. In the extensive works of [9] it is showed that calculations on sparse matrices are significantly faster if the non zero elements are clustered along the diagonal.

### 1.4.1 Lexicographical Numbering

Lexicographical Numbering is based on letters in alphabetical order. In numerical analysis this pattern is mimicked but often with the difference that integers are used instead of letters. The two lexicographical ordering schemes Breadth First Search and Cuthill Mckee Numbering are commonly used to number meshes. In these ordering schemes vertices of a matrix are numbered with integers.

### 1.4.2 Breadth First Search

The Breadth First Search pattern in [8] is based on Lexicographical Numbering. The basic idea of Breadth First Search numbering is to start at the vertex with the lowest number of adjacent vertices and then to visit vertices one by one until all vertices are numbered. The order in which vertices are visited are governed by a pending queue. All vertices that are adjacent to the current vertex are put in a pending queue. When a vertex has been numbered the first vertex in the pending queue is next to be visited and numbered. By following the Breadth First Search scheme the numbering of the vertices in the wrench like object are as in Figure 1.3.

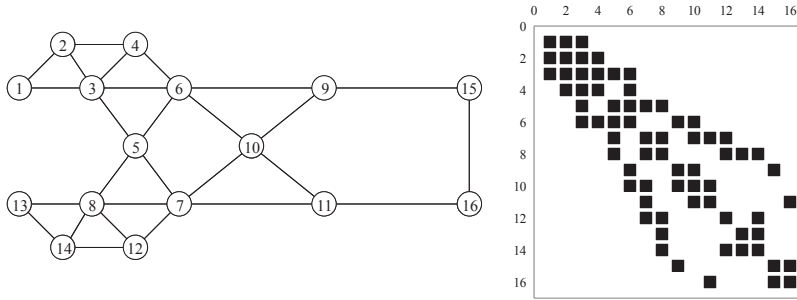


**Figure 1.3** Breadth First Search numbering on the wrench mesh. A connectivity matrix derived by numbering the wrench mesh with the Breadth First Search scheme.

The connectivity matrix for this mesh can be seen in Figure 1.3. The matrix is clearly more diagonally dense than that of the mesh with randomly numbered vertices in Figure 1.1 and hence more suitable for efficient sparse matrix calculation algorithms.

### 1.4.3 Cuthill Mckee Numbering

The Cuthill Mckee Numbering differs from the Breadth First Search ordering in one aspect. In the Cuthill Mckee Numbering the number of adjacent vertices of each currently handled vertex is taken in consideration [8]. In Cuthill Mckee Numbering adjacent vertices are placed in the pending queue depending on their number of adjacent vertices. Vertices with a low number of adjacent vertices are placed first. In Figure 1.4 the resulting mesh of the Cuthill Mckee Numbering is visualized. The pattern of the vertices follows an increase of magnitude from the first element to the last. The branching of the mesh highlights a situation where the smooth increase of the magnitude of the vertex numbering is deteriorating. The connectivity matrix of the wrench mesh derived by using the Cuthill Mckee Numbering scheme is also presented in Figure 1.4.



**Figure 1.4** The wrench mesh numbered by using the Cuthill Mckee Numbering scheme. A matrix derived from the wrench mesh with Cuthill Mckee Numbering ordering.

By using an ordering algorithm it is clear that the resulting connectivity mesh is more diagonally dense than if the numbering is done randomly, see Figure 1.1 and 1.4. To get an even more diagonally behavior of a sparse matrix graph color algorithms can be implemented.

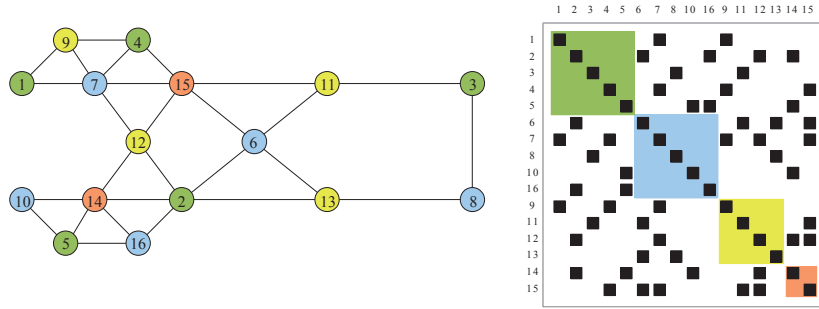
## 1.5 Graph Coloring

Graph coloring is widely spread and used in many different fields. The aim of this study is the graph analysis step in iterative solvers. The basic idea of graph coloring is to organize elements such that no interconnected elements share the same color. The heuristic approach of most contemporary graph coloring algorithms implies that a local optimal coloring leads to a sufficient global coloring. This approach is commonly known as a greedy approach.



### 1.5.1 Colored Matrices

Graph coloring algorithms can be performed on any mesh hence any matrix. A graph coloring is performed on the wrench mesh with randomly numbered vertices in Figure 1.1. The connectivity matrix can be organized such that elements with the same color are placed in the same group. This is done by changing the ordinary sequential numbering from first to last element when the sparse matrix is assembled. Instead each group of elements with the same color is placed in proximity in the sparse matrix. With this organization of the elements strictly diagonal parts are formed in Figure 1.5 rendered in the sparse connectivity matrix.



**Figure 1.5** The randomly numbered wrench mesh in Figure 1.2 is graph colored. No adjacent vertices share the same color. The sequential permutation is altered to obtain the grouping by color. Diagonal blocks in the matrix highlight the origin of the colored vertices in the mesh.

The sparse matrix in Figure 1.5 is one of an indefinite number of different color matrices possible to derive from the wrench mesh. There are a lot of fine-grained details that affect the mesh rendering and hence the resulting sparse matrix. The starting vertex, the order of which the vertices are visited in the graph coloring and the ordering scheme are used to render the numbering of the vertices all affect the resulting matrix.

### 1.5.2 The Number of Colors

One feature of a graph coloring of sparse matrices is the number of colors needed to sufficiently color a graph, or mesh. The numbers of colors needed to separate the elements in the mesh in Figure 1.5 is four. Saad concludes that a low number of color is sought [8]. This is due to the fact that a system with a sparse matrix ordered by graph coloring with a lower number of colors is generally solved faster. A higher number of colors tends to hurt the speed-up gained by graph coloring. The number of colors needed in a graph coloring is often referred to as the quality of the coloring. A low number of colors is associated with high quality coloring.

Allwright describes the concept of optimal graph colorings [4]. The usage of the term optimal is however problematic due to the fact that the graph coloring problem is NP-hard. This makes it very difficult to determine if a high quality graph coloring really is optimal in the literal sense. Generally the idea is to use the  $d+1$  criterion, where  $d$  is the maximum degree of the graph being colored. The maximum degree of a graph is the number of adjacent vertices of the vertex with the highest number of adjacent vertices.

Color quality defined by the  $d+1$  criterion is often an easy goal to achieve, e.g. the colors needed to color the nations of a world map. Some countries have a high number of adjacent nations, Russia, Brazil, China, but the number of colors is usually kept at four<sup>4</sup> in such maps. If the  $d+1$  criterion is applied in this map example a world map with 16 colors among the nations is to be considered optimal, due to the fact that China has 15 adjacent countries.

Two more relevant criteria are suggested here. Both these suggested criteria are based on the degree of a graph. The refinement of these criteria is that the average degree  $\bar{d}+1$  and the median degree  $\tilde{d}+1$  are used.

<sup>4</sup> The four color theorem, a classical mathematical problem.

$$\bar{d} = \frac{1}{n} \sum_{i=0}^n d_i \quad ; n=0,1,\dots,N \quad (1.1)$$

$$\tilde{d} = d_{n/2}$$

An additional condition has to be implemented to cover situations where multiple vertices hangs free, i.e. vertices with no adjacent vertices. The degree of each vertex have to be higher or equal to 2,  $d_i \geq 2$ . By using one of the two proposed criteria in (1.1) high extreme values of the degree is negated. Hence the predicted number of colors is more in line with results seen in colored graphs.

### 1.5.3 Multi-Coloring

Multi-Coloring is done by an iteration through the vertices of a mesh. A vertex is colored when it is visited and the color assigned is dependent on the coloring of adjacent vertices.

The original pattern of a sparse matrix is mirrored in the multi-colored version of the matrix. The non diagonal blocks in a random sparse matrix are still more or less randomly spread. On the other hand if a sparse matrix with elements clustered along the diagonal is reorganized by using a Multi-Coloring algorithm the non diagonal blocks are generally filled with clustered elements. The greedy Multi-Coloring algorithm from [8] is presented in Algorithm 1.1.

---

**Algorithm 1.1** Greedy Multi-Coloring

---

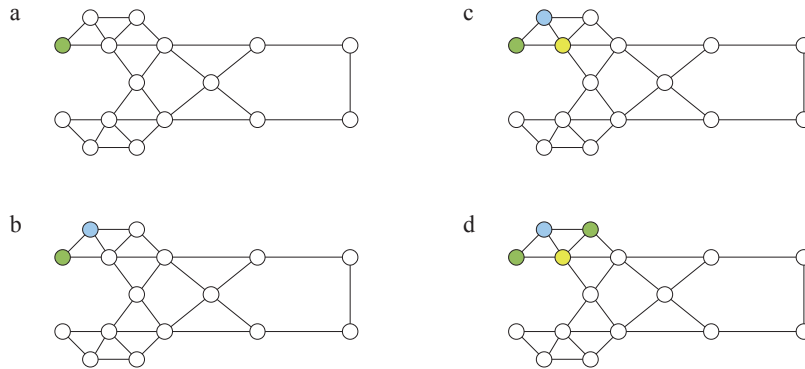
```

for  $V_i$  do
  color( $V_i$ ) = 0
end for
for  $V_i$  do
  color( $V_i$ ) = min{  $m > 0 \mid m \neq \text{color}(V_j), \forall V_j$  adjacent to  $V_i$  }
end for

```

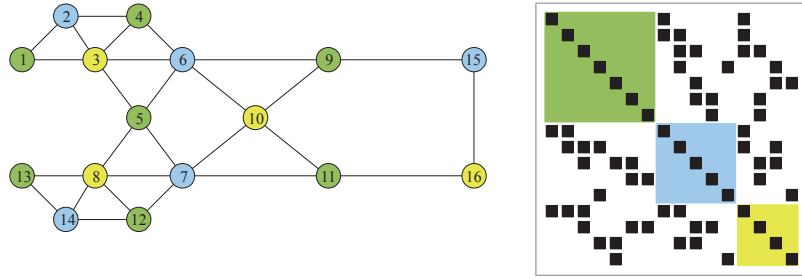
---

In Figure 1.6 greedy Multi-Coloring is performed on the wrench mesh with vertices that are numbered by the Cuthill Mckee Numbering scheme.



**Figure 1.6** The greedy Multi-Coloring algorithm is performed on the wrench mesh. a) No vertex is colored hence the first color, green, is assigned to a vertex. b) Vertex 2 is colored, the first color is present in the adjacent vertices so the second color, blue, is assigned. c) When vertex 3 is colored green and blue color is present in adjacent vertices so the third color, yellow, is assigned to vertex 3. d) The first color is not present in any adjacent, hence the vertex colored in this step is assigned green color. This procedure is followed until all vertices are colored.

The colored connectivity matrix and the colored mesh is presented in Figure 1.7. The matrix is permuted according to the colors assigned to each vertex.



**Figure 1.7** The wrench mesh ordered by Cuthill Mckee Numbering is colored by the greedy Multi-Coloring algorithm. The connectivity matrix is permuted after being colored.

#### 1.5.4 Luby Jones

In [10] an iterative way of coloring graphs is described. The basic idea is to iterate through numbered vertices and assign a color where a vertex value, each vertex is assigned a random value, forms a local minimum among adjacent vertices. For each iteration one color is assigned and each colored vertex is taken from the list of vertices up for coloring. Next iteration a new color is used to color uncolored vertices. The procedure is repeated until there are no uncolored vertices left. In this scheme the number of colors is equal to the number of iterations needed to finish the coloring.

The iterative Luby Jones graph coloring is sensitive to the numbering of the mesh. The numbering must be strictly randomized otherwise random numbers must be assigned to the vertices. If the elements are not randomly numbered the situation where the number of colors are equal or close to the number of vertices might occur. The Luby Jones algorithm is presented in Algorithm 1.2.

---

#### Algorithm 1.2 Basic Luby Jones

---

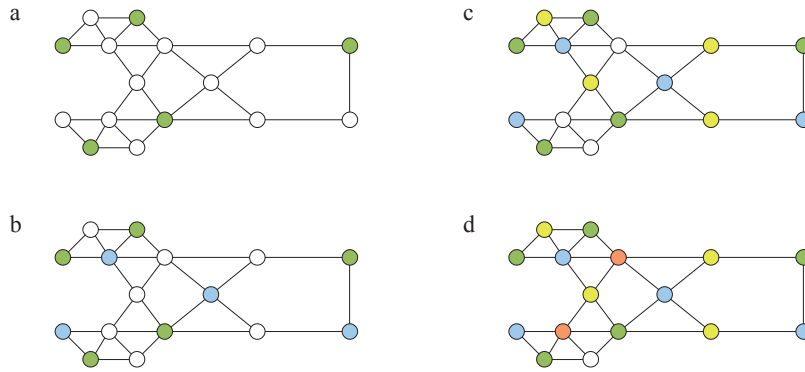
```

color = 0
 $\forall V_i$  assign random numbers
while  $\exists \text{color}(V_i) = 0$  do
  for  $V_i$  do
    for  $V_j$  adjacent to  $V_i$  do
      if  $\text{color}(V_i) = 0$  and  $\text{color}(V_j) = 0$  and  $\text{random}(V_i) > \forall \text{random}(V_j)$ 
        color( $V_i$ ) = color
      end if
    end for
  end for
  color++
end while

```

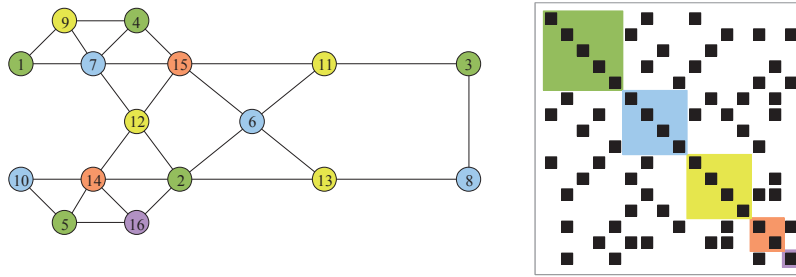
---

The Luby Jones iterative coloring algorithm is performed on the randomly numbered wrench like mesh in Figure 1.2 and presented in Figure 1.8. In every iteration local minimums are colored.



**Figure 1.8** In this version of the iterative Luby Jones coloring scheme local minimums are colored. a) In the first iteration local minimums are given green color. b) In the second iteration local minimums are given blue color. Vertices all ready colored are disregarded. d) When all vertices are colored the Luby Jones scheme is rounded up and the iterations are stopped.

When all vertices have been colored the global iteration is stopped and the matrix is permuted according to the colors assigned. In Figure 1.9 the randomly numbered wrench mesh is colored and permuted according to the coloring.



**Figure 1.9** The random numbered mesh colored by the Luby Jones algorithm. The connectivity matrix permuted according to the iterative Luby Jones coloring.

Compared to the Multi-Coloring algorithm the Luby Jones iterative algorithm generally needs a higher number of colors. The high number of colors is a drawback that occur even for well suited matrices. A compelling example is given in [10] where Castonguay and Cohen describes the Luby Jones algorithm using a mesh, a chessboard, that obviously can be sufficiently colored by two colors. The Luby Jones algorithm uses six colors to color the chessboard mesh. This behavior of the Luby Jones approach is also seen in Figure 1.9. The number of colors in the wrench mesh example above is five whereas the Multi-Coloring, Figure 1.7, algorithm uses three colors to sufficiently color the same mesh.

### 1.5.5 Distance-k Graph Coloring

The idea of Distance-k Graph Coloring is to assign colors to vertices in such a way that all vertices within the distance-k from each other has unique colors. The integer  $k$  denotes the number of edges between two vertices in a connecting path between two compared vertices. Distance-k Graph Coloring implementations utilizes ways of forming the distance between two vertices. In [11][12][13] Distance-k Graph Coloring is investigated. It is concluded that graph coloring algorithms that implement distance restriction are robust on both iterative and multi-color schemes. Although Distance-k Graph Coloring results in a high number of colors needed in a graph coloring.

A general Distance-k Graph Coloring restrain can be implemented in any graph coloring algorithm. In Algorithm 1.3 an implementation of the greedy Multi-Coloring that utilizes a Distance-k Graph

Coloring restrain is exemplified.

---

**Algorithm 1.3** Distance-k Graph Coloring
 

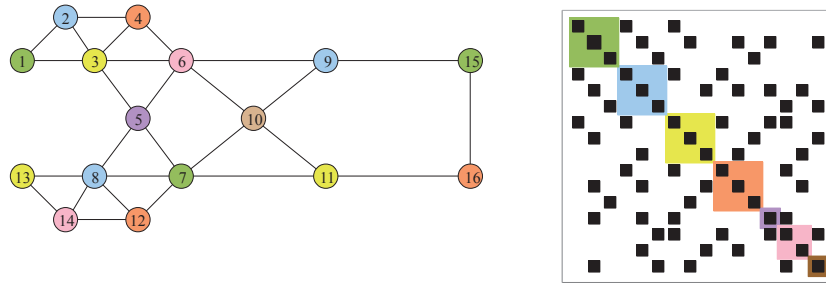
---

```

for  $V_i$  do
   $\text{color}(V_i) = 0$ 
end for
for  $V_i$  do
  if  $\text{distance}(V_i, V_j) > k$ 
     $\text{color}(V_j) = \min\{m > 0 \mid m \neq \text{color}(V_i)\}$ 
  end if
end for
  
```

---

A Distance-2 Graph Coloring is performed on the wrench like mesh and is presented in Figure 1.10, the permuted matrix is also presented in the picture.



**Figure 1.10** A Distance-2 Graph Coloring on the wrench like mesh. The connectivity matrix permuted according to the Distance-2 Graph Coloring.

Distance-k sub-sets can be derived in different ways. One way is to set-up the iterations through vertices in a way that the distance-k vertices are visited. Given the special case where  $k=2$  a distance-2 sub-set of the graph  $G$  is formed by iterating through all vertices  $g_{i,j}$ . By performing a nested iteration through all adjacent vertices to each  $g_{i,j}$  a distance-2 sub-set is formed. The iterative scheme of forming a distance-2 sub-set is outlined in Algorithm 1.4.

---

**Algorithm 1.4** Distance-2 sub-set, iterative approach
 

---

```

for  $V_i$  do
  for  $V_j$  adjacent to  $V_i$  do
    for  $V_k$  adjacent to  $V_j$  do
       $\forall V \in (V_k + V_j) \rightarrow \text{distance-2 sub-set}$ 
    end for
  end for
end all
  
```

---

Another way of forming the distance-k sub-set is to perform matrix multiplications on the graph  $G$ . A distance-k sub-set is formed by  $G * G * \dots * G^k$ . The evident drawback is that the multiplication produces a denser matrix. This is in contrast to a key feature of iterative solvers of linear systems. Iterative solvers such as the Krylov subspace method maintain the sparsity of a system throughout the solution phase. Nevertheless distance-2 sub-set is often formed by direct multiplication. To prevent cancellation of elements the absolute value of  $g_{i,j}$  is used, i.e. the distance-2 sub-set is formed by  $|G|^2$  (direct multiplication). The advantage is that the iterations through distance-2 sub-sets are transformed into a  $O(n)$  problem.

---

**Algorithm 1.5** Distance-2 sub-set, matrix multiplication
 

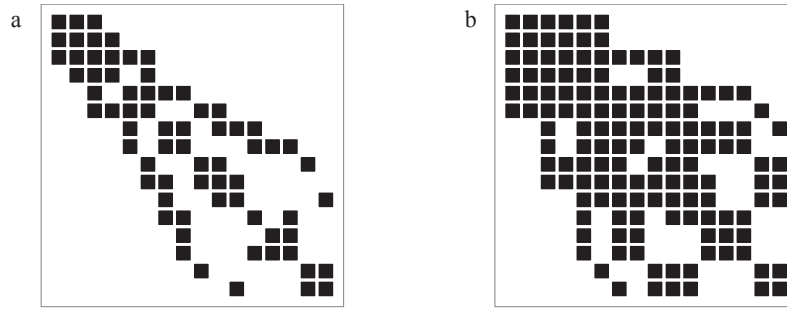
---

```

 $S = |G|^2$ 
 $\forall \text{row}(S) \rightarrow \text{distance-2 sub-set}$ 
  
```

---

A distance-2 sub-sets matrix of the wrench mesh is formed by matrix multiplication. Each row of  $|G|^2$  forms a distance-2 sub-set. A distance-2 sub-set is formed on the wrench matrix in Figure 1.11.



**Figure 1.11** a) A connectivity matrix  $M$ . b) The multiplied matrix  $M^2$ . Each row of  $M^2$  is a distance-2 sub-sets. The sparsity of  $M$  is hurt by the multiplication.

In addition to the hurt of sparsity of the matrices forming the linear system the quality of Distance-k Graph Coloring is poor. This is evident by looking at the degree of a graph. The degree is hurt when the sparsity of a graph is hurt. As mentioned in section 1.5.2 *The Number of Colors* the number of colors in a graph coloring is dependent of the degree of a graph.

## 2 Parallel Graph Coloring

The last decades a lot of work<sup>5</sup> has been put into the process of parallelizing graph coloring algorithms. The attempts of achieving sufficient parallel graph coloring implementations can be divided into two categories. One category where graphs are divided into different kinds of sub-sets where each sub-set are sequentially graph colored by a Multi-Coloring algorithm. The other category is made up by derivatives of the iterative Luby Jones approach.

Several recent articles have been published where linear speed-up is claimed for parallel graph coloring algorithms [11][14]. The performance of these parallel implementations is however poor in terms of execution time and coloring quality. None of these parallel methods have left greedy Multi-Coloring algorithm obsolete [4]. In this paper new parallel graph coloring algorithms are proposed.

The platforms in focus is multi-core CPUs. With the goal of exposing and utilizing parallelism in graph coloring without damaging the quality of the colorings rendered.

### 2.1 General Aspects in Parallel Graph Coloring

Some features have to be considered in parallel graph coloring. Due to the nature of many graph color algorithms and the data handled situations that hurt the performance easily occur. Some of these difficulties will be discussed in detail in this thesis. Here are some general aspects that have to be addressed when graph color algorithms are parallelized.

#### 2.1.1 Load Balancing

Due to the irregular nature of the data it is hard to ensure equal workload among threads. Local irregularities might lead to a situation where vertices on one thread have many interconnections that have to be handled whereas on another thread there are fewer. This situation might lead to heavier workload on some cores than others.

#### 2.1.2 Race Conditions

The vertices in a mesh are interconnected. If a graph coloring algorithm is performed in parallel, situations where several threads tries to access and write to the same vertex might occur. This means that the mesh data needs to be protected to avoid corrupt data, i.e. corrupt coloring, due to race conditions.

#### 2.1.3 Memory Access and Latency

A mesh often has interconnected vertices that are stored in different places in the memory. The poor locality might result in high latency due to slow memory access time of the mesh data. Lumsdaine covers these issues in [15]. In fact, it is stated that the calculations done in a graph coloring algorithm are so cheap that the memory latency often dominates the execution time, not the calculation speed of the thread. This is obviously not necessarily a problem confined to a parallel implementation of a graph coloring algorithm, but nevertheless an issue that have to be considered.

### 2.2 Local Parallelism

In the coloring of graphs it is possible to utilize local parallelism. When a vertex is colored the iteration through the adjacent vertices can be the target for parallelization. The efficiency of the local parallelism in the graph coloring described in this section is conditional to the number of connections of each vertex in a mesh.

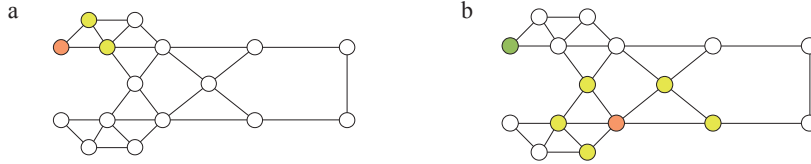
Parallelization of the iteration through adjacent vertices is done by distributing the adjacent vertices among the threads. The comparison among adjacent vertices is performed on each thread and the final

---

<sup>5</sup> The majority of the articles covering parallel graph coloring referred in this paper are published post 1990.

result derived by comparing the result from each thread. Due to the latency of thread spawning in application programming interfaces (API) like Open Multi-Processing (OpenMP) and POSIX Threads (pthread) the work on each thread needs to be considered. A high number of connections between vertices in the mesh results in a workload that is sufficient for local parallelism. An example of a mesh where local parallelism might be targeted is a mesh over friend to friend connections on social networks such as facebook.

In Figure 2.1 the local parallelism is poor but visualized as an example. The advantage with local parallelism is that it is possible to adjust to global parallelism in many of the graph coloring schemes.



**Figure 2.1** A sequential graph coloring algorithm is performed on a wrench like mesh. a) A starting vertex is chosen, marked orange. Adjacent vertices, marked yellow, are distributed among threads and compared to determine which color the currently colored orange vertex is going to get. b) A new vertex is chosen and the adjacent vertices are distributed.

In Table 2.1 a summary of the features of local parallelism in graph coloring is presented.

Table 2.1 Features of local parallelism	
Strength	Weakness
Can be implemented by compiler	Conditional scaling

## 2.3 Global Parallelism

Attempts of implementing efficient parallel graph coloring are almost exclusively based on the greedy Multi-Coloring algorithm. The iterative Luby Jones approach is an exception to this. Albeit there are many detailed differences the basic idea is to divide the sparse matrix and to perform serial graph coloring on each sub-set. A majority of these parallel implementations are based on a shared memory architecture. Some exceptions exist, in [11] Bozdag *et. al.* presents an algorithm utilizing a distributed memory.

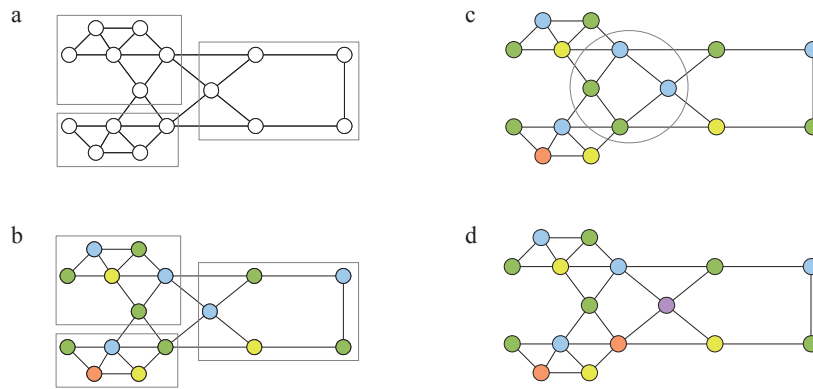
Regarding the threading on cores it is suggested that massive<sup>6</sup> multi-threading is the way to tackle the future of parallel graph coloring [15]. With a thorough review of the graph coloring algorithms on modern hardware Lumisdaine *et.al.* concludes that there is a trade-off between dense memory structures and fast computation speed.

### 2.3.1 Naive Multi-Coloring

The obvious drawback with the Naive Multi-Coloring approach is the need of an efficient way of handling conflicts in the interface between sub-sets of the divided mesh. This problem is present in Figure 2.2.

<sup>6</sup> The expression, massive multi-threading, is used by Lumisdaine *et.al* to emphasize a high number of cores where a number of threads are swapped in and out of each core to reduce memory latency.





**Figure 2.2** a) The mesh is divided and distributed among threads. b) Each sub-set is colored in parallel. c) The encircled vertices are conflicting. d) The final coloring is derived when all conflicts are resolved.

The example presented in Figure 2.2 highlights issues that might occur in parallelizations of the greedy Multi-Coloring algorithm. A risk in parallel graph colors is that the number of colors might be higher, hence the quality of parallel graph coloring is worse than sequential graph coloring. The wrench like mesh can be sufficiently colored by using 3 colors, see Figure 1.10, but in the example above the number of colors are 5. The cause of this is that the threads traverse the vertices in a way that hurt the global quality of the coloring. In Figure 2.2.c. conflicts are encircled. This kind of conflicting colorings have to be handled to get a sufficient coloring. The conflict handling might lead to a need of even more colors to achieve a sufficient coloring. In Algorithm 2.1 a mesh is divided and each sub mesh is sequentially graph colored on separate threads. When each part is colored conflicts in the interface have to be detected and recolored.

---

**Algorithm 2.1** Naive Multi-Coloring

---

```

Divide matrix  $M$ 
for  $V_i$  do in parallel
   $\text{color}(V_i) = 0$ 
end for

phase 1 – tentative coloring
for  $V_i$  do in parallel
   $\text{color}(V_i) = \min \{ m > 0 \mid m \neq \text{color}(V_j), \forall V_j \text{ adjacent to } V_i \}$ 
end for

phase 2 – detection and resolution of conflicts
for  $V_i$  do in parallel
  if  $\text{color}(V_i) = \text{color}(V_j)$ 
     $\text{color}(V_i) = \text{new color}$ 
  end if
end for

```

---

### 2.3.2 Distance-2 Graph Coloring

In recent years a lot of focus have been on Distance-k Graph Coloring algorithms in the graph coloring community, distance-2 in particular [11][12][13]. The main aim of these implementations is graph coloring algorithm that scales on multi-core hardware. Most of these contemporary Distance-2 Graph Coloring schemes are preformed in three phases; in the first phase maximal independent sets are formed, in phase 2 tentative coloring is performed on each maximal independent set, in the last phase conflicts along the interfaces are resolved. There are fine-grained differences between different implementations but the over all scheme is the same in all distance-k graph colorings. In short, Distance-k Graph Coloring is based on a Multi-Coloring algorithm with the addition that a distance constraint is applied.

The results presented by Gebremhin *et. al.* clearly shows that Distance-k Graph Coloring algorithms are not leaving the greedy Multi-Coloring algorithm obsolete [11]. Albeit the distance-k algorithms being parallel implementations and the greedy Multi-Coloring algorithm is sequential. There are several reasons for the weak performance of parallel Distance-k Graph Coloring algorithms. The, perhaps, most obvious argument against Distance-2 Graph Coloring is the hurt of the sparsity of the matrices representing the graphs. In particular if the graph coloring is used in the graph analysis step in an iterative solver of a linear system. On the implementation level extensive usage of book keeping data structures that effects the memory load and the extensive communication between cores to resolve conflicts severely hurt the performance. Another issue is the elaborate nature of these algorithms itself. This complex nature of the Distance-2 Graph Coloring is in contrast to the simplicity of other known algorithms such as the greedy Multi-Coloring and the Luby Jones algorithms.

The formation of distance-2 maximal independent sets is done by randomly creating spawning vertices for distance-2 sub-sets. The maximal independent sets are then built by adding surrounding vertices to the spawning vertex in two steps. In the first step distance-1 sub-sets are created. In the second step the distance-2 sub-set is created. This proceeds until there are no unlinked vertices left. An alternative way to form  $S=|G|^2$ , where each row in  $S$  is a maximal independent sets of distance-2. Tentative coloring is performed on the maximal independent sets sets from the previous phase of the Distance-2 Graph Coloring algorithm. The greedy Multi-Coloring algorithm is implemented on each maximal independent set. In the last phase of the Distance-2 Graph Coloring algorithm conflicts are detected and resolved. When all conflicts have been resolved a sufficient coloring of the graph is obtained. The basic scheme of parallel Distance-2 Graph Coloring algorithm is presented in Algorithm 2.2.

---

**Algorithm 2.2** Distance-2 Graph Coloring

---

*Phase 1*

Form  $S$  = distance-2 sub-set

$V \in S$

divide  $S$  and assign intervals of  $V_i$  to threads

*Phase 2 – tentative coloring*

**for**  $V_i$  **do in parallel**

**if** distance( $V_i, V_j$ )  $> k$

        color( $V_i$ ) = min{  $m > 0 \mid m \neq \text{color}(V_j)$  }

**end if**

**end for**

*Phase 3 – detection and resolution of conflicts*

**for**  $V_i$  **do in parallel**

**for**  $V_j$  adjacent to  $V_i$  **do**

**if**(color( $V_i$ ) == color( $V_j$ ))

            color( $V_i$ )

            Form list of proxy conflicts

            Send list of proxy conflicts to threads

**end if**

**end for**

**end for**

---

### 2.3.3 Summary

In table Table 2.2 the features of parallel graph coloring are summarized.

<b>Table 2.2</b> Parallel Multi-Coloring algorithms	
<b>Strength</b>	<b>Weakness</b>
<b>Naive Multi-Coloring</b>	
Low number of colors	Produces conflicts along interfaces
<b>Distance-2 Graph Coloring</b>	
Scaling has been shown [11][14]	Extensive communication
	Complex implementations
	High number of colors

## 2.4 Luby Jones

Castonguay and Cohen states that the iterative Luby Jones scheme is inherently parallel [10]. The parallelism is due to the fact that the random number of a vertex is a local maximum no matter which thread that sees it, given that the visited vertex is a maximum. Hence, no conflicts have to be solved in a parallelization of the graph coloring algorithm. This heuristic parallelism implies that if local parallelism is found it will transpire into global parallelism. In Algorithm 2.3 the Parallel Luby Jones algorithm is presented.

**Algorithm 2.3** Parallel Luby Jones

```

forall  $V_i$   $color = 0$ 
forall  $V_i$  assign random numbers
while  $\exists color(V_i) = 0$  do
  for  $V_i$  do in parallel
    for  $V_j$  adjacent to  $V_i$  do
      if  $color(V_i) = 0$  and  $color(V_j) = 0$  and  $random(V_i) > \forall random(V_j)$ 
         $color(V_i) = color$ 
      end if
    end for
  end for
   $color++$ 
end while

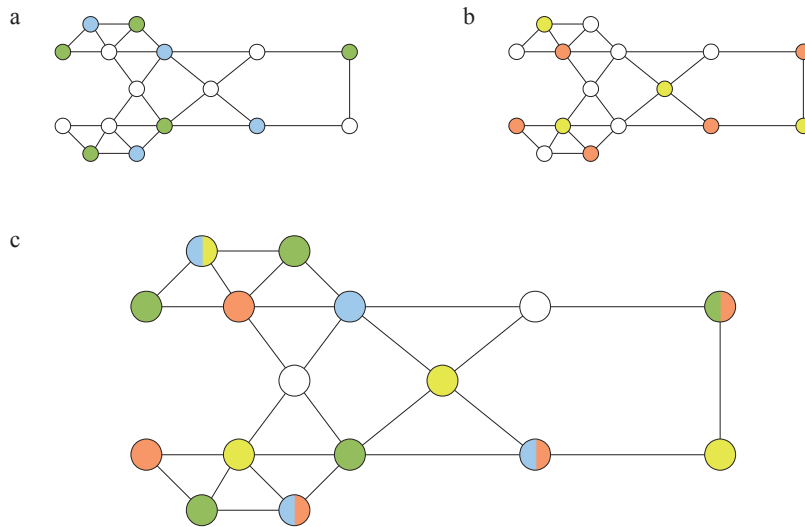
```

### 2.4.1 Max - Min Search

Depending on the hardware used it might be beneficial to opt between a search for maximums and a simultaneous search for maximums and minimums. The Max - Min Search is based on the assumption that the assigned random number of a vertex cannot form a local maximum and minimum at the same time. Even though the iterations needed to color a graph is significantly reduced it is not automatically given that the execution time is improved since the work and memory latency in each comparison is higher.

### 2.4.2 Randomized Multi-layer Coloring

To scale down the number of global sweep through all vertices multiple sets of random numbers are assigned to the vertices. Multiple colorings are done in each iteration, hence the number of uncolored vertices are significantly lowered. To get an unique color for each vertex one color is picked of the multiple colors given to each vertex. In Figure 2.3 an iterative graph coloring is presented where these two techniques are implemented.



**Figure 2.3** a,b) Multiple sets, or layers, of random numbers are used. Local maximums and minimums are colored with a set of two colors for each set of random numbers. c) The result from a and b are combined in c.

The colorings in Figure 2.3 are combined to get a coloring of the mesh. The figure also highlights a weakness in the Randomized Multi-layer Coloring. Some vertices are left uncolored. This can be handled in different ways. One way is to control the final coloring in the sense that each vertex actually is colored. Another way is to aim high and use a higher number of random sets to ensure coloring of each vertex.

### 2.4.3 Summary

In Table 2.3 a summary of the features of iterative Luby Jones graph coloring schemes are presented.

<b>Table 2.3</b> Luby Jones	
<b>Max - Min Search</b>	
<b>Strength</b>	<b>Weakness</b>
Inherently parallel	High number of colors
	Sensitive to numbering
<b>Randomized Multi-layer Coloring</b>	
<b>Strength</b>	<b>Weakness</b>
Fewer iterations [10]	High number of colors
Inherently parallel	Sensitive to numbering

## 2.5 New Methods

In contemporary graph coloring a common approach is to color interior vertices in parallel and then to implement safe coloring of the interface. Graph coloring algorithms with this set-up differs in the way which the interface is tackled and how the vertices are divided. One example of these different approaches is where the interface is colored by variations of the iterative Luby Jones algorithm [4] [15]. Another approach is presented in [13][14] where a tentative coloring with conflicts is performed on the interface. These coloring schemes implements different ways of forming sub-sets. When the vertices in each sub-set have been colored conflicts are detected and resolved. A semi-parallel version of graph coloring is discussed in [14] where the interface is sequentially colored.

In this section new parallel graph coloring algorithms are proposed. These algorithms are based on the greedy Multi-Coloring algorithm and a division of the matrix and coloring of sub-sets in parallel. The previously unseen approach in these coloring algorithms is the handling of the interface of a divided matrix. A key feature of these new algorithms are the formation of dependent and independent pairs along the interface of a the divided matrix. Independent in the sense that an internal pair of vertices are independent of any vertices located outside its own sub-set. If a pair of vertices are placed in different sub-sets, i.e. owned by different threads, these vertices forms a dependent pair. Algorithm 2.4 is used to form dependent pairs.

---

**Algorithm 2.4** Dependent pairs
 

---

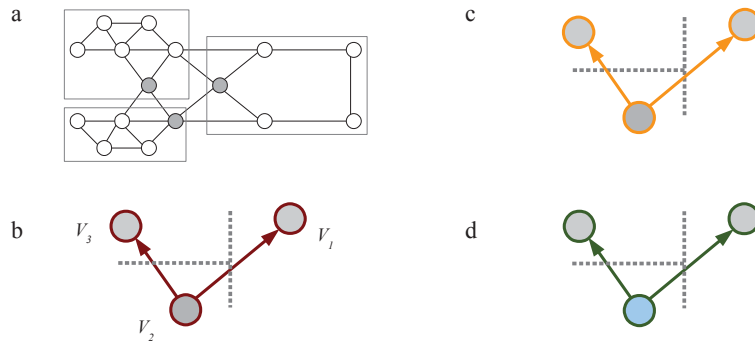
```

Divide  $\forall V$  into intervals of  $I$ 
for  $V_i$  do in parallel
  for  $V_j$  adjacent to  $V_i$  do
    if  $((V_j \text{ adjacent to } V_i) \in I)$ 
       $\rightarrow (V_j, V_i)$  independent pair
    else if  $((V_j \text{ adjacent to } V_i) \notin I)$ 
       $\rightarrow (V_j, V_i)$  dependent pair
  end for
end for
  
```

---

### 2.5.1 Parallel Multi-Coloring

The first proposed algorithm Parallel Multi-Coloring is based on features that can be utilized on dependent vertices that pose potential conflicts. Conflicts are avoided by utilizing a thread safe coloring on dependent vertices that pose possible conflicts. If a vertex forms a dependent pair with one of its adjacent vertices it is colored in a thread safe way. An example of how the thread safe coloring of critical vertices can be done is outlined in Figure 2.4.



**Figure 2.4** The colors depicts: red = lock, orange = check and green = unlock. The dotted line is the interface between the mesh parts. a) A graph is divided. b) Locks are applied on sensitive vertices around  $V_2$ . c)  $V_2$  reads colors in adjacent vertices. d) When  $V_2$  has been colored the locks on adjacent vertices are released.

In Figure 2.4.a, a vertex  $V_2$  forms dependent pairs with a vertex  $V_1$  and  $V_3$ . In Figure 2.4.b,  $V_2$  locks the two adjacent vertices  $V_1$  and  $V_3$  that are placed in proximity sub matrices. The locking is done in order to avoid race conditions which might lead to conflicting coloring. The thread owning  $V_2$  performs a local Multi-Coloring. The value of the color of adjacent vertices are checked and a color is assigned to  $V_2$  according to the colors found among adjacent vertices. In Figure 2.4.d,  $V_2$  is colored and the locks are released.

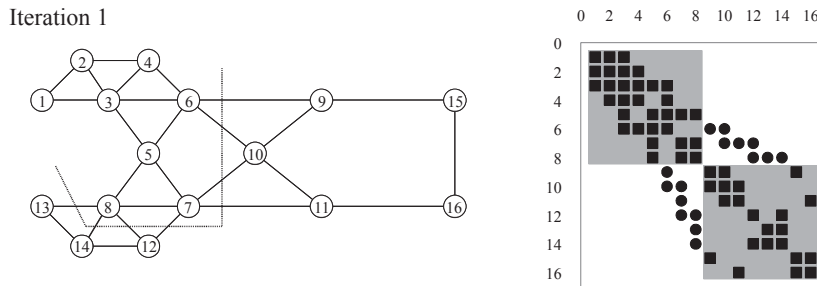
In both of the examples above conflicts are avoided by locking the dependent adjacent vertex to the vertex being colored. In Algorithm 2.5 the thread safe coloring of a divided mesh is presented.

**Algorithm 2.5** Parallel Multi-Coloring*Initiation*Divide Matrix  $M$  intervals of  $I$ **for**  $V_i$  **do in parallel**     $\text{color}(V_i) = 0$ **end for***coloring phase**a. invoking***for**  $V_i$  **do in parallel**    **for**  $V_j$  adjacent to  $V_i$  **do**        **if**  $((V_j \text{ adjacent to } V_i) \in I)$              $\text{color}(V_i) = \min\{m > 0 \mid m \neq \text{color}(V_j), \forall V_j \text{ adjacent to } V_i\}$         **else if**  $((V_j \text{ adjacent to } V_i) \notin I)$              $\text{lock}(\text{dependent } V_j \text{ adjacent to } V_i)$              $\text{dependent} = \text{true}$     **end for***b. thread safe*    **if**  $(\text{dependent})$         **for**  $V_j$  adjacent to  $V_i$  **do**             $\text{color}(V_i) = \min\{m > 0 \mid m \neq \text{color}(V_j), \forall V_j \text{ adjacent to } V_i\}$              $\text{unlock}(V_j)$         **end for**    **end for**

The parallelism of Algorithm 2.5 is dependent on the performance of the locking mechanism implemented. The number of locks needed also effect the parallelism, hence a matrix with a low bandwidth is graph colored faster than a matrix with high bandwidth. In other words, if a sub-set has many adjacent vertices in a proxy sub-set the parallelism is hurt.

**2.5.2 Normann Lukarski**

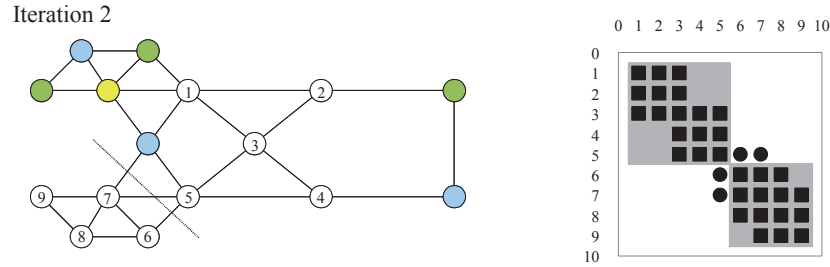
The second proposed algorithm, Normann Lukarski, is an iterative method where the interior vertices in sub-sets are trivially colored and the interface colored with features making the coloring thread safe. The formation of dependent and independent pairs of vertices are used to target the interface and interior vertices. When independent vertices are colored all threads are synchronized and the next step of the algorithm is taken so that the dependent vertices can be colored. In the following step the interface is targeted. The set-up of this iterative scheme is visualized in Figure 2.5.



**Figure 2.5** Independent pairs of vertices are depicted by squares, dependent pairs by circles. The original matrix is divided and the parts are distributed among threads. Independent vertices of each sub-set is safely colored, i.e. the vertices 1-5,15,16.

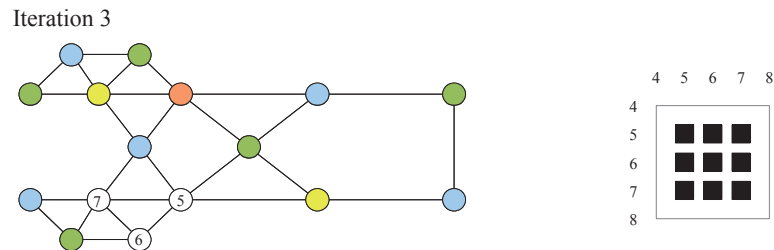
The coloring of independent vertices in each sub-set is done by an efficient sequential graph coloring algorithm, e.g. the greedy Multi-Coloring algorithm. If a coloring of independent sets is performed once on all threads the interface is left uncolored. By numbering these uncolored vertices in the interface a new matrix is formed. To ensure a diagonally dense matrix the Cuthill McKee Numbering algorithm is used to number the uncolored interface vertices. Now the vertices left uncolored in the

previous step is colored in the same manner as the internal vertices in the previous step. The coloring is done by accessing the color of the colored adjacent vertices from the former internal vertices. If this procedure is repeated a number of time all vertices will eventually be colored. The next step of the algorithm is taken in Figure 2.6.



**Figure 2.6** The internal uncolored vertices from the first step are numbered by the Cuthill Mckee Numbering algorithm to get a more diagonally dense matrix. Once again the internal vertices are safely colored, as in Figure 2.5.

When the number of uncolored vertices is small enough to cause more overhead than speed-up by the parallelization the uncolored vertices are sequentially colored. It is hard to estimate when this will occur. It is however likely to occur after a small number of iterations. A vivid example of the rate of which the interface is shrinking is a cross. In the first step the fields of the cross are colored. In the next step the arms of the cross are colored. By now there is only a small piece of the cross left, the center of the cross. A matrix being colored by the Normann Lukarski algorithm will behave similar to the exemplified cross. The third step of the parallel coloring of the wrench mesh is taken in Figure 2.7.



**Figure 2.7** At this stage the number of uncolored vertices is so low that a sequential coloring is likely to be the more efficient than a parallel coloring. If sequential coloring is to be done the vertices do not need to be numbered by the Cuthill Mckee Numbering algorithm.

There are two main concerns in this approach. The need of synchronization and numbering of the vertices in the interface. In each step of this algorithm synchronization with numbering of the uncolored vertices is needed so that the updated matrix can be divided and distributed among the threads involved. The numbering is necessary in this approach since the number of independent vertices is dependent of the diagonal density of the matrix. Without the numbering in each synchronization the number of independent vertices in each iteration shrinks rapidly and the method breaks down.

The complexity of the Cuthill Mckee Numbering algorithm is  $O(nnz)$  [16], where  $nnz$  denotes the number of non zero elements in a sparse matrix. In addition to this, the workload on each thread is more evenly spread than would be the case if the dependent vertices were not numbered. Normann Lukarski is presented in Algorithm 2.6. It is hard to make an assessment on what stop conditions to use in this scheme. In an implementation of this scheme sufficient stop conditions can be elaborated.

**Algorithm 2.6** Normann Lukarski

---

```

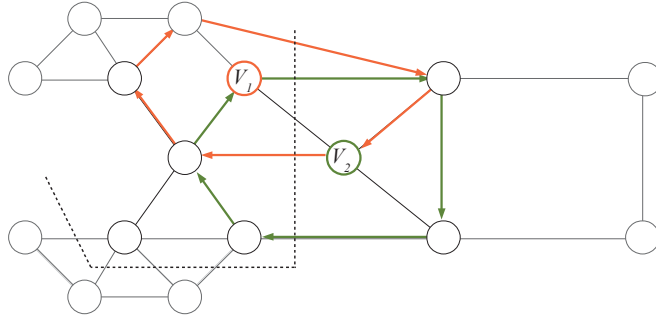
while  $\exists \text{color}(V_i) = 0$  do
  Divide Matrix  $M$  intervals of  $I$ 
  for  $V_i$  do in parallel
    for  $V_j$  adjacent to  $V_i$  do
      if  $(\forall V_j \text{ adjacent to } V_i) \in I$  and  $\text{color}(V_i)=0$ 
         $\text{color}(V_i) = \min\{ m > 0 \mid m \neq \text{color}(V_j), \forall V_j \text{ adjacent to } V_i \}$ 
      end for
    end for
    Synchronization : form refined  $M$ , i.e. number uncolored vertices
  end while

```

---

**2.5.3 Normann**

The third proposed algorithm is the Normann algorithm. In the Normann algorithm local conflicts are resolved by letting each vertex iterate through adjacent vertices multiple times. The global scheme in which each sub-set is traversed is governed by the numbering of the graph. In Figure 2.8 the basic idea of the Normann algorithm is outlined. The green and orange loops are performed a few times to lower the risk of conflict. If a conflict is present the probability that it will remain throughout the local coloring process is small. This is due to the fact that if a conflict is present at any point the conflict only needs to be resolved once to never appear again. The need of a control of the coloring is a drawback in this algorithm. Albeit the low risk of a conflict a color check is done to ensure a sufficient coloring. Another feature that might be problematic is that the number of colors is likely to fluctuate in runs of the Normann algorithm. The threads will traverse through the vertices differently due to irregularities in the reading to and writing from the memory. These irregularities are likely to cause the coloring to change between different executions.



**Figure 2.8** Two vertices  $V_1$  and  $V_2$  in proximity sub-sets are in risk of assigning conflicting colors. Each vertex iterates through adjacent vertices multiple times to lower the risk of conflicts

In Algorithm 2.7 the Normann coloring method is presented in pseudo code.

**Algorithm 2.7** Normann

---

```

Divide Matrix  $M$  intervals of  $I$ 
for  $V_i$  do in parallel
  for  $V_j$  adjacent to  $V_i$  do
    for  $V_j$  adjacent to  $V_i$  do
       $\text{color}(V_i) = \min\{ m > 0 \mid m \neq \text{color}(V_j), \forall V_j \text{ adjacent to } V_i \}$ 
    end for
  end for
end for
for  $V_i$  do in parallel
  if(conflict( $V_i$ ))
    set  $\text{color}(V_i) \neq \text{color}(V_j \text{ adjacent to } V_i)$ 
  end for
end for

```

---



### 2.5.4 Summary

The features of the parallel graph coloring algorithms purposed here are summarized in Table 2.4.

<b>Table 2.4</b> New parallel graph coloring algorithms	
<b>Strength</b>	<b>Weakness</b>
<b>Parallel Multi-Coloring</b>	
Low number of colors	Dependent on locks
<b>Normann Lukarski</b>	
Low number of colors	Synchronization needed
	Numbering needed
<b>Normann</b>	
Low number of colors	Conflict resolution sometimes needed
	The number of colors fluctuates



## 3 Implementation and Test Configuration

The main focus of this work is performance regarding parallelism, execution time and the quality of the colorings. The candidates chosen for implementation are algorithms that are deemed most promising in respect of these performance requirements. The work of implementing the graph coloring algorithms is done by implementing sequential versions of the algorithms. In the sequential implementations parallel sectors are created and then parallelized by the aid of some wisely chosen API.

### 3.1 Benchmark Set-up

#### 3.1.1 Hardware

The experimental tests are performed on a server called Gullviva at the Division of Scientific Computing at Uppsala University. The CPU of Gullviva is a AMD Opteron 6274, 2.2 GHz with 16-cores on a dual socket. The test runs are confined to 8 threads to negate the possible latency between the two sockets.

#### 3.1.2 Language and APIs

With a good support of different parallelizing libraries the C/C++ language is chosen for the implementation. If fine-grained control of the parallelization is needed the pthread library is used to parallelize the sequential implementations, otherwise the openMP library is used. In general a graph coloring algorithm implementation denotes colors by assigning integers to each vertex. In this work the colors are denoted as integers 1,2, etc. Each thread colors the assigned vertices that are stored in global Compressed Sparse Row arrays. The speed-up is monitored by comparing the execution time of one thread compared with the given number of threads. Compiler optimization is implemented throughout the tests by using the -O3 flag.

#### 3.1.3 Test Matrices

All test matrices<sup>7</sup> used in this work are freely available. The matrices are grouped by size and listed in Table 3.1 - 3.3. The size, number of non zero entries and the number of colors listed in Table 3.1 - 3.3 are taken from the specifications on the site offering the particular matrix. The degrees of the matrices are derived by a function that counts the number of adjacent vertices in a matrix. Max Degree is the highest number of adjacent vertices found among the vertices in a graph. Avg. Degree is the average degree among all vertices in a graph.

**Table 3.1** Big test matrices

Matrix	Size 10 <sup>6</sup>	Entries 10 <sup>6</sup>	Colors	Max Degree	*Avg. Degree
bone010	0.9	71.7	39	80	71.6
Flan_1565	1.6	117.4	42	80	74.0
Geo_1438	1.4	63.2	29	56	42.9
Hook_1498	1.5	60.9	30	92	39.7
ldoor	1.0	46.5	42	76	47.9
pwtk	2.2	11.5	48	179	52.4
Serena	1.4	64.5	36	248	45.4

\*Avg. Degree: The average degree among all vertices in a graph.

<sup>7</sup> <http://math.nist.gov/MatrixMarket/>, <https://www.cise.ufl.edu/research/sparse/matrices/>

**Table 3.2** Test matrices

Matrix	Size $10^6$	Entries $10^6$	Colors	Max Degree	*Avg. Degree
apache2	0.7	4.8	3	7	5.7
ecology2	1.0	5.0	2	4	4.0
G3_circuit	1.6	7.7	4	5	3.8
msdoor	0.4	20.2	42	76	47.7
offshore	0.3	4.2	12	30	15.3
StocF-1465	1.5	21.0	11	188	13.3
thermal2	1.2	8.6	7	10	6.0

\*Avg. Degree: The average degree among all vertices in a graph.

**Table 3.3** Small test matrices

Matrix	Size $10^3$	Entries $10^3$	Colors	Max Degree	Avg. Degree
nos1	0.2	1.0	2	4	3.3
nos2	0.9	4.1	2	4	3.3
nos3	0.9	15.8	10	17	15.5
nos4	0.1	0.6	4	6	4.9
nos5	0.5	5.2	9	22	10.1
nos6	0.7	3.3	2	4	3.8
nos7	0.7	4.6	2	6	5.3

\*Avg. Degree: The average degree among all vertices in a graph.

## 3.2 Sequential Implementations

### 3.2.1 Global Parallelism

In the tests carried out in this work a version of the greedy Multi-Coloring<sup>8</sup> algorithm is used. A greedy Multi-Coloring implementation is well suited for the usage as a benchmark [4]. The sought properties of low number of colors, and fast execution time are present in greedy Multi-Coloring implementation. In addition to the traits described the greedy Multi-Coloring algorithm is well known and often referred to as the best graph coloring algorithm.

### 3.2.2 Distance-2 Graph Coloring

The implementation of sequential Distance-2 Graph Coloring is based in the greedy Multi-Coloring algorithm. Distance-2 sub-sets are formed by calculating  $|G|^2$  then the sequential version of the Multi-Coloring implementation is used to color the distance-2 matrix. No implementation of creating distance-2 sub-sets by spawning sub-sets around randomized extreme points is tested in this work. This is mainly due to the poor performance of this particular algorithm [11].

### 3.2.3 Luby Jones

A key feature of the iterative Luby Jones algorithms is the use of random numbers. An implementation of the Fisher Yates shuffle algorithm [17] is used to randomize sequential numbers. These random numbers are assigned to vertices and used to create local extreme values i.e. local maximums and minimums. Another loop is used to govern the iterations. Inner loops are used to iterate through all uncolored vertices with a comparison against uncolored adjacent vertices. If a local extreme value is found the associated vertex is assigned a color.

<sup>8</sup> Paralution 0.5.0, <http://www.paralution.com/>

The colors used are incremented by one in each sweep, i.e. the first sweep uses the color 1 and so on. In the search for local maximums in the implementation set-up in this paper each vertex is first marked as a potential maximum. If an adjacent vertex shows a bigger random number the visited vertex is unmarked as a local maximum. In the case where the visited vertex is not unmarked a color is assigned. In each iteration all vertices already colored are disregarded.

The Max - Min Search is implemented in the same manner as the Max coloring scheme. The difference is that each visited vertex is also marked as a potential minimum. If no smaller adjacent vertex is found the vertex is a minimum. This coloring calls for two colors in each sweep, one for maximums and one for minimums.

One additional optimization is done in the Max - Min Search implementation. If a vertex have no adjacent vertices it is colored. The color assigned to lonely vertices is the same as the color for local maximums.

### 3.3 Parallel Implementations

The parallel graph coloring implementations in this paper are based on the formation of the division of vertices into intervals of vertices. The vertices of a sparse matrix in the Compressed Sparse Row format are divided into intervals  $I=(start, end)$  and distributed to threads. The intervals  $I$  are calculated as,

```
start = pid*n/p;
end = (pid+1)*n/p-1;
```

The thread ID  $pid$ , the numbers of threads  $p$  and the number of vertices  $n$  are all trivially retrievable. In the parallelizations where the interface is needed the vertices are checked by controlling whether a vertex is in the assigned interval or not.

#### 3.3.1 Normann

The parallelization scheme chosen for the parallel graph coloring algorithm is the Normann algorithm. It is not showed in this work that Normann is the best of the three new algorithms. Preliminary tests on Parallel Multi-Coloring showed that this method often creates deadlocks. Due to time constraints the work of implementing the Normann Lukarski algorithm is not pursued.

The core of the implementation of the Normann algorithm is the iterative repetition of the local coloring. This is done by using the same length on the local intervals as there are adjacent vertices. In other words use the same loop as the loop used to iterate through adjacent vertices. Given that the matrix is in Compressed Sparse Row format the row pointer is used for this:

```
for(int i=row_ptr[curr]; i<row_ptr[curr+1]; i++)
```

Two different control functions are developed to detect conflicts. One of these control functions detects misplaced vertices in matrices permuted by graph coloring. In detail it is done by controlling that no vertices in the permuted matrix in a diagonal block is placed outside the diagonal. The other control function checks for conflicting colors. This control function iterates through all vertices and checks for adjacent vertices with a conflicting color. The reason for implementing two different control functions is to make it possible to control the coloring on different levels of the Normann implementation. The control function which looks for conflicting pairs is implemented directly in the graph coloring function. The control function which looks for misplaced vertices are implemented outside.

Since both control functions merely reads data a parallel approach is safe. A feature of the conflict that needs to be monitored in the tests is the number of conflicts in the tentative phase of the Normann implementation. If the number of conflicts are low a parallelization of the conflict resolution will cause more overhead than speed-up.

With  $nnz$  non zero entries in a matrix and  $m$  adjacent vertices the complexity of a color check is  $O(nnz+m)$ . Without stretching to far this can be approximated to  $O(nnz)$ , due to the fact that  $m$  is much smaller than  $nnz$ .

### 3.3.2 Luby Jones

The Max - Min Search algorithm is used for the parallelization of the iterative Luby Jones scheme. The parallelization is done by a division of the vertices in a graph. Each thread colors the assigned vertices. When all threads have colored the assigned vertices the parallel Max - Min Search is done.

### 3.3.3 Parallelism

A rough estimate of the parallelism of the Normann algorithm is calculated by the usage of Amdahl's law [18],

$$S(p) = \frac{1}{1 - f_p + \frac{f_p}{p}} \quad (3.1)$$

If the speed-up  $S$  for a large number of threads  $p$  is considered the parallel fraction  $f_p$  of the algorithm can be estimated. By letting  $p$  go towards infinity  $f_p$  can be determined as,

$$\lim_{p \rightarrow \infty} S(p) = \frac{1}{1 - f_p} \quad (3.2)$$

Which can be written as,

$$f_p = 1 - \frac{1}{\lim_{p \rightarrow \infty} S(p)} \quad (3.3)$$

It should be noted that the speed-up for the Normann algorithm might not be bounded for a high number of threads. In such a case the parallelism might be better described by Gustavsson's law [18]. If that is the case the estimate used here is pessimistic. In addition to this possible low estimate the parallelism is highly dependent on the matrix being colored. In the tests done to assess the parallelism the workload is considerably high to reach the maximum parallelism of the Normann algorithm. To get a high workload big test matrices are used to clock execution times, see 3.1.3 *Test Matrices* for details.

## 3.4 Verification

All colorings are verified by a function that checks the output from the coloring function. The verification function is checking the correctness of a coloring by taking the number of non zeros, the permuted sparse matrix in Compressed Sparse Row format, the size of each color and the number of colors as parameters.

## 4 Results

In this section the performance of the tested algorithms are presented. Here it is shown that the proposed algorithm outperforms all other algorithms regarding the number of colors and execution time.

### 4.1 Overview

In Table 4.1 the performance in terms of color quality and execution time for the iterative Luby Jones, greedy Multi-Coloring, Normann and the Distance-2 Graph Coloring methods are presented. Parallel Distance-2 Graph Coloring is not implemented due to the poor quality and the way Distance-2 Graph Coloring hurts the sparsity of the matrices in a linear system.

**Table 4.1** Performance of algorithms

	<b>LJ</b>		<b>Para<sup>8</sup> LJ</b>		<b>MC</b>		<b>Para<sup>8</sup> N</b>		<b>Para Dist-2</b>	
<b>Matrix</b>	<b>Time</b>	<b>Color</b>	<b>Time</b>	<b>Color</b>	<b>Time</b>	<b>Color</b>	<b>Time</b>	<b>Color</b>	<b>Time</b>	<b>Color</b>
apache2	0.3	19	0.1	19	0.1	3	0.04	5	-	-
bone010	22.4	151	3.3	150	14.4	39	3.2	45	-	-
ecology2	0.2	17	0.1	15	0.1	2	0.02	4	-	-
Flan_1565	36.9	157	5.2	156	23.8	42	5.1	48	-	-
G3_circuit	0.4	15	0.3	14	0.2	4	0.1	4	-	-
Geo_1438	11.6	91	1.8	90	8.2	29	1.7	36	-	-
Hook_1498	11.1	93	1.8	92	8.0	30	1.7	33	-	-
ldoor	8.8	93	1.5	92	6.7	42	1.6	49	35.8*	112*
msdoor	3.8	95	0.6	93	2.9	42	0.7	45	36.9*	105*
offshore	0.4	39	0.1	37	0.3	12	0.1	12	-	-
pwtk	2.4	99	0.4	97	1.8	48	0.4	48	45.5*	180*
Serena	12.9	109	2.2	107	8.9	36	1.9	39	-	-
StocF-1465	1.6	35	0.4	34	1.4	11	0.3	12	-	-
thermal2	0.5	19	0.3	19	0.4	7	0.1	7	-	-

*The number of colors and execution times for the Luby Jones (LB), the greedy Multi-Coloring (MC), the Normann (N) and the Distance-2 Graph Coloring (Dist-2) coloring methods. Para<sup>8</sup> is the results from execution on 8 threads. \*Results from [11] used as comparison.*

Table 4.1 shows the color quality of the algorithms benchmarked in this work. By comparing with the degrees listed in Table 3.1 to 3.3 the color quality is assessed. The Distance-2 Graph Coloring achieves the  $d+1$  criterion for the *pwtk* graph and is far from this criterion for the *msdoor* and *ldoor* graphs. The iterative Luby Jones algorithms, parallel and sequential, are not rendering colorings with a lower number of colors than the  $d+1$  criterion for any graph but the *StocF-1465* graph. The differences in color count for these algorithms originates from the randomness of the numbers forming local maximums and minimums. Both the greedy Multi-Coloring and the Normann algorithms meets the  $d+1$  criterion for all tested graphs.

## 4.2 Color Quality

To assess the quality of the colorings performed by different graph coloring algorithms the number of colors is monitored, the result is presented in Table 4.2.

**Table 4.2** Quality of coloring

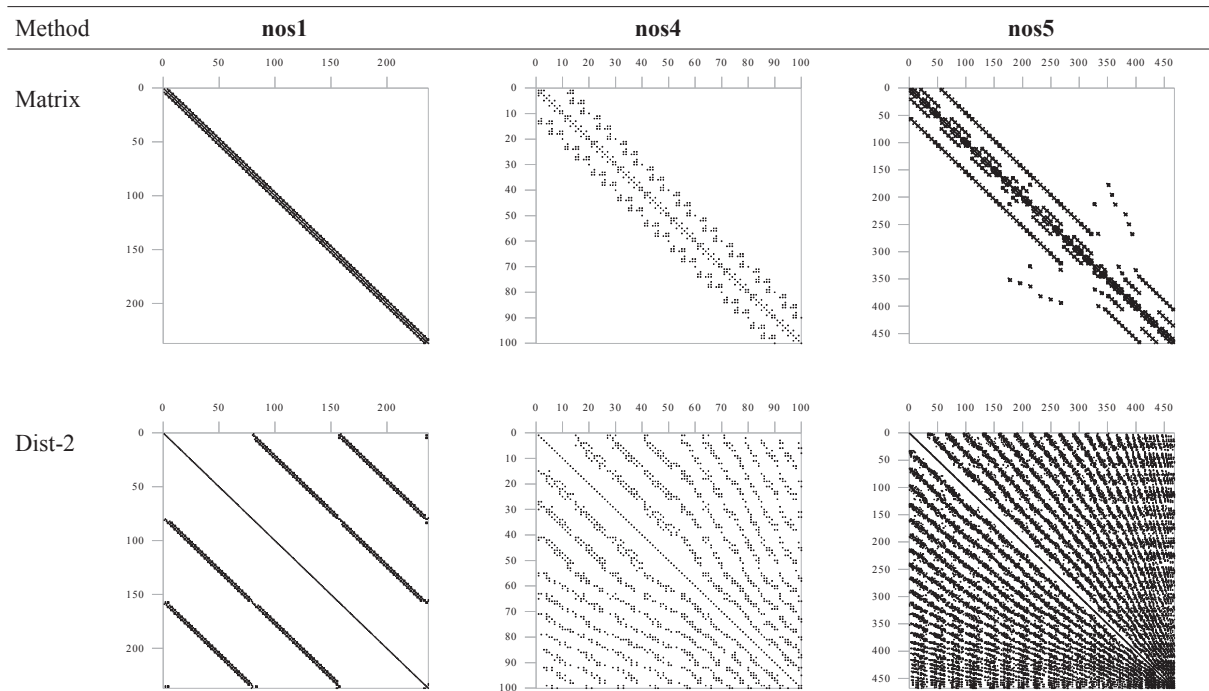
Matrix	LJ	Para <sup>8</sup> LJ	MC	Para <sup>8</sup> N	*Dist-2	**Dist-2
nos1	8	7	2	2	6	4
nos2	9	10	2	3	6	6
nos3	29	29	10	10	19	18
nos4	10	12	4	4	7	13
nos5	20	25	9	9	17	33
nos6	9	11	2	2	6	7
nos7	15	15	2	2	8	12

*The number of colors in the Luby Jones (LJ), the greedy Multi-Coloring (MC), the Normann (N) and the Distance-2 Graph Coloring (Dist-2) methods. Para<sup>8</sup> is executions on 8 threads.*

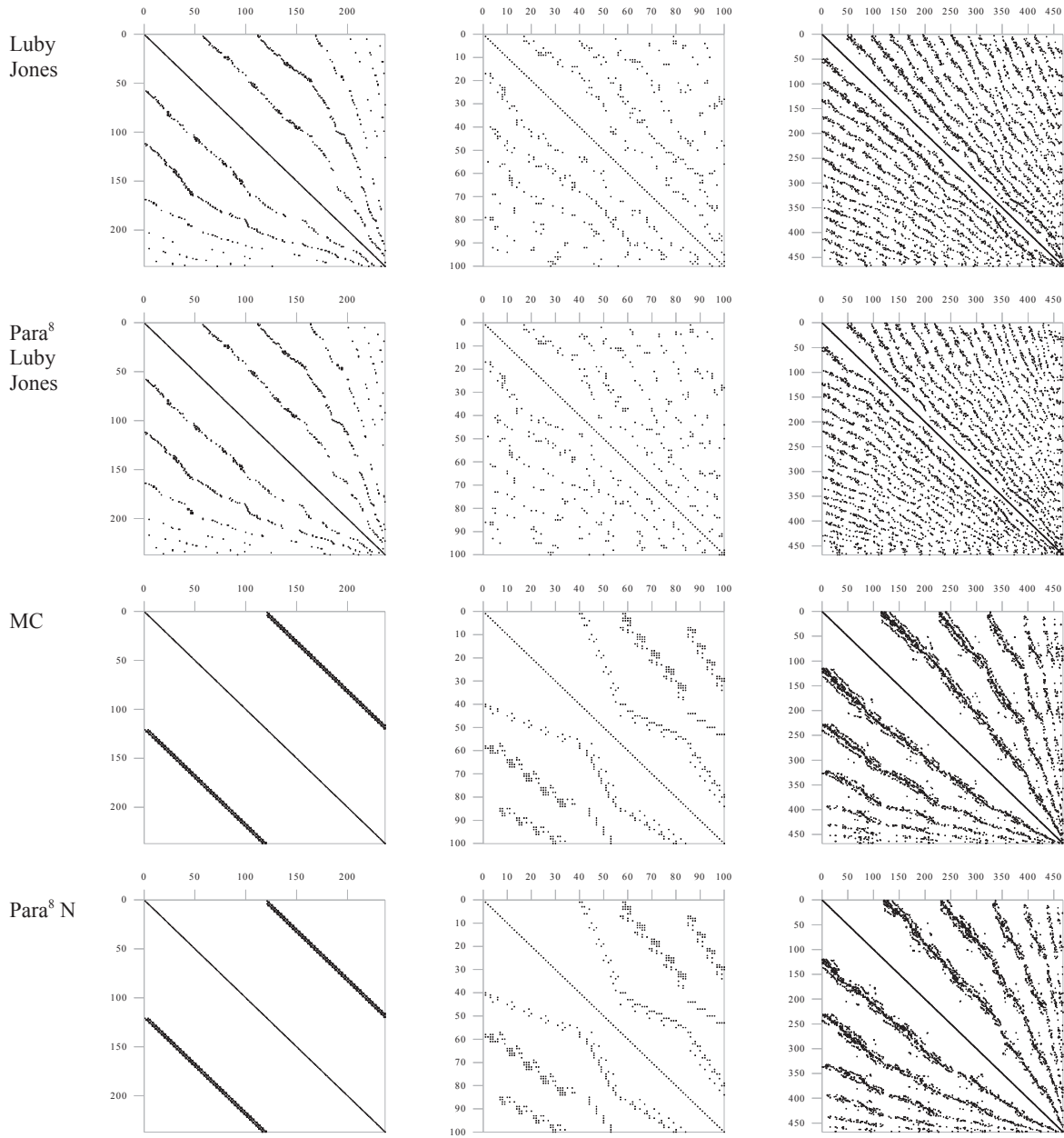
*\*Dist-2, the distance-2 sub-sets formed by iteration. \*\*Dist-2, the distance-2 sub-set formed by matrix multiplication.*

Tests are performed to visualize the quality of the coloring in different graph coloring methods. The set of test matrices used in this experiment are small and not very dense. This type of sparse matrices are chosen to make a visualization possible. Three of the test matrices in Table 4.2 are used to visualize the coloring quality of the implemented algorithms. The plots in Figure 4.1 shows the graphs colored by the algorithms benchmarked in this work.

**Figure 4.1** Visualized graph coloring





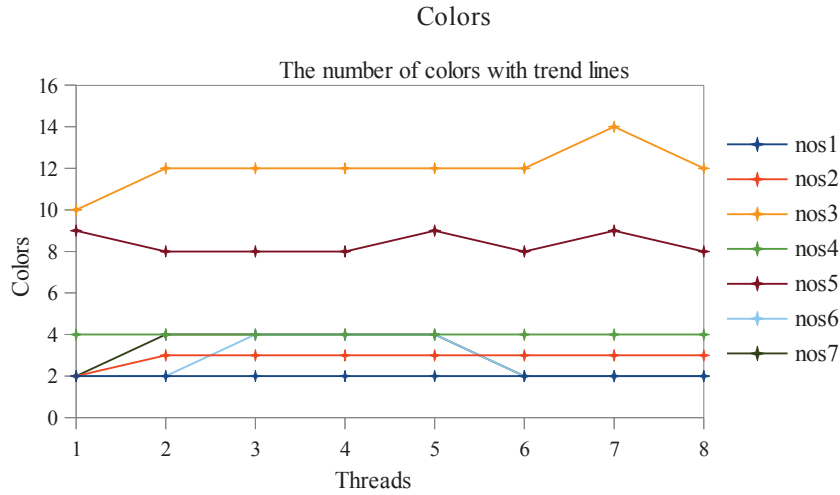


The main purpose of the plots in Figure 4.1 is to display how graph colored and permuted matrices looks. The methods presented are the Luby Jones (LJ), the greedy Multi-Coloring (MC), the Normann (N) and the Distance-2 Graph Coloring (Dist-2) methods. In addition to this the quality of the different types of graph coloring algorithms is seen. The similarities between the sequential and parallel Luby Jones algorithm are evident. The fine-grained difference between these two iterative graph coloring algorithms are due to the randomness introduced by the random numbers that are used to form local maximums and minimums. Similarities are also seen between the Multi-Coloring and the Normann algorithms. The fine-grained difference between these two algorithms are due to the way in which the graph is traversed. The local parallel colorings sometimes transpire into beneficial global colorings and sometimes not.

### 4.3 Color Consistency

Below plots with the number of colors used by Normann algorithm on different number of threads are presented. The number of colors is a key feature in graph coloring and the high quality of the

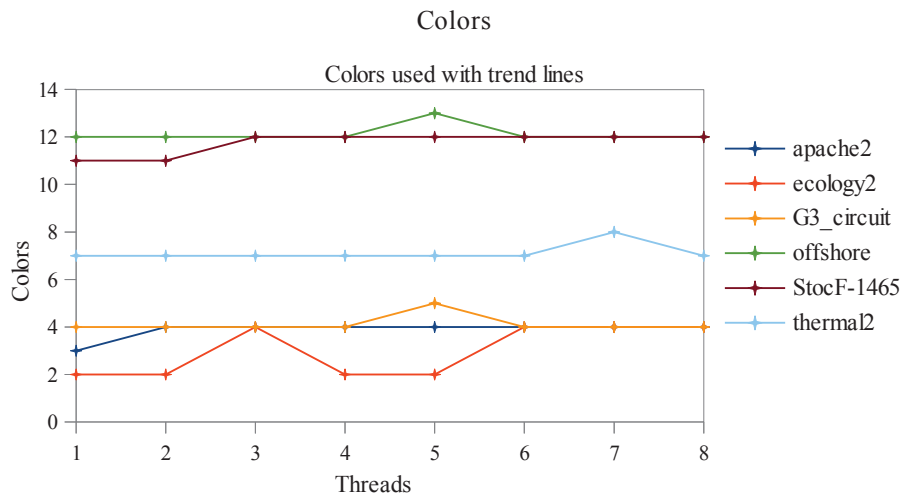
colorings rendered by the Normann algorithm is evident. The number of colors in the colorings of the small matrices are shown in Figure 4.2.



**Figure 4.2** The number of colors used by the parallel Normann algorithm when performed on test matrices.

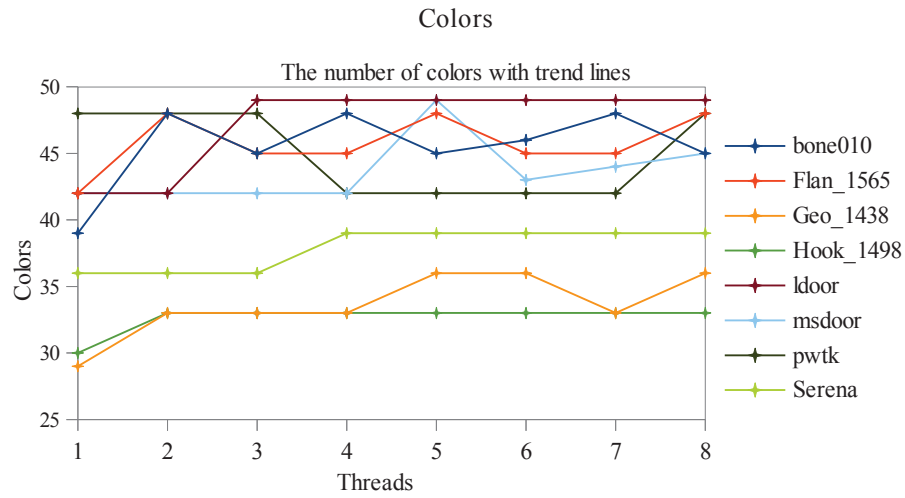
The Normann algorithm shows colorings with a slight fluctuation in the number of colors on different number of threads. This is due to the fact that the matrices are traversed differently when the coloring is done in parallel. The greedy Multi-Coloring implementation always traverse a graph in the same manner whereas the parallel Normann implementation traverse differently depending on how the threads are working in parallel. If a local coloring of a vertex transpires into a favorable coloring on a global scale the color count is lower than that of the greedy Multi-Coloring implementation. On the other hand, if a local coloring of a vertex is unfavorable the threads are coloring in a way which results in a coloring with a higher number of colors.

The set of big matrices is divided into two plots, one with graphs colored with less then 20 colors and one plot with graphs colored with more than 20 colors. In Figure 4.3 the number of colors for different number of threads for matrices sufficiently colored by less than 20 colors.



**Figure 4.3** Colors used by the Normann algorithm while performed on test graphs.

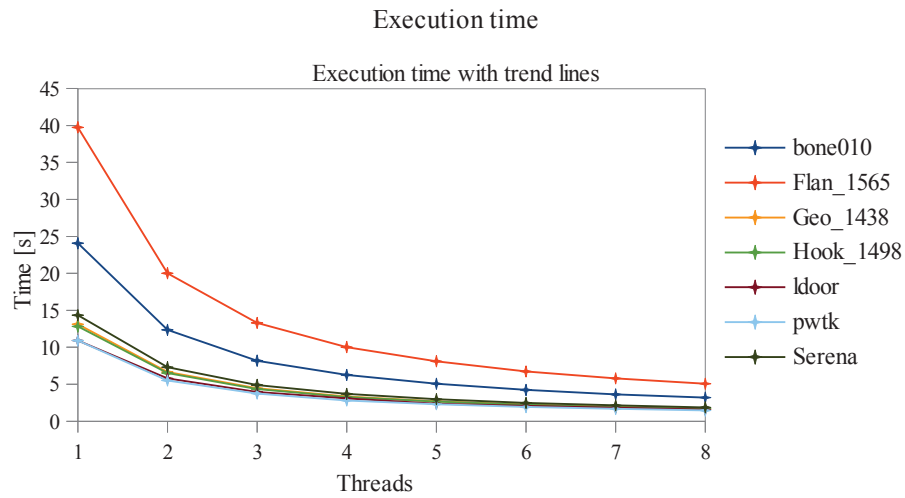
The number of colors are slightly fluctuating in Figure 4.3 as in Figure 4.2. In Figure 4.4 test matrices that were sufficiently colored by more than 20 colors are presented.



**Figure 4.4** The number of colors used by the Normann algorithm on different number of threads.

## 4.4 Execution Time

The execution time for the Normann implementation is monitored. In Figure 4.5 the execution time on big matrices is plotted.



**Figure 4.5** The execution time of the Normann algorithm.

In Figure 4.6 the execution time for a set of matrices are plotted. The execution times are cut significantly by the up scaling of multiple threads.

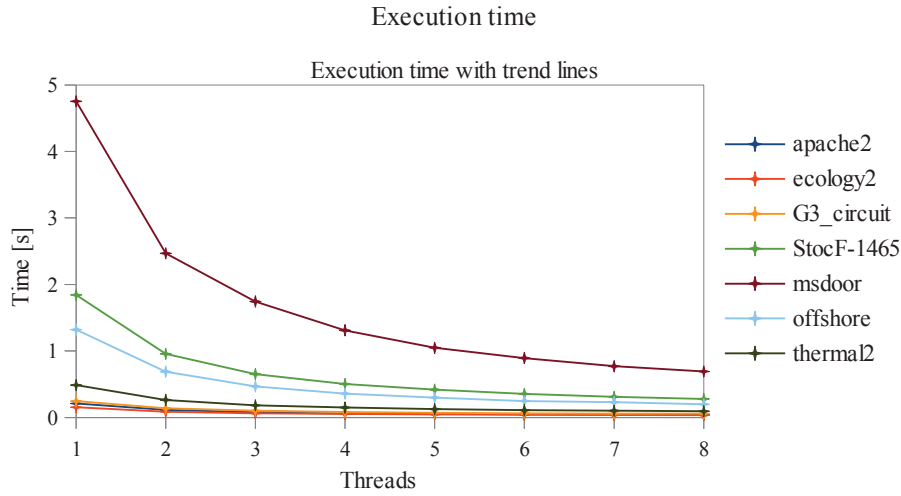


Figure 4.6 The execution time of the Normann algorithm.

## 4.5 Parallel Speed-up

In test runs the execution time is monitored. The execution time in turn is used to calculate and plot the speed-up at an increased number of threads. In Figure 4.7 the speed-up gained by increasing the number of threads while performing the Normann algorithm on bigger test matrices is seen.

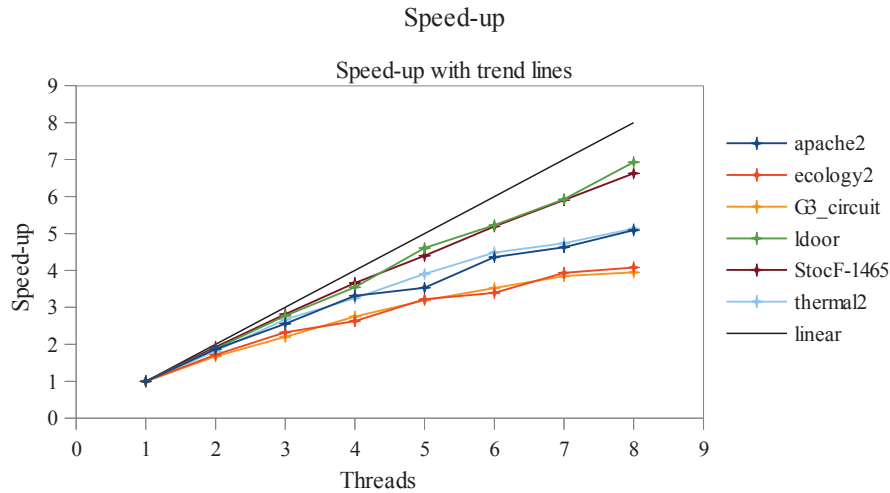
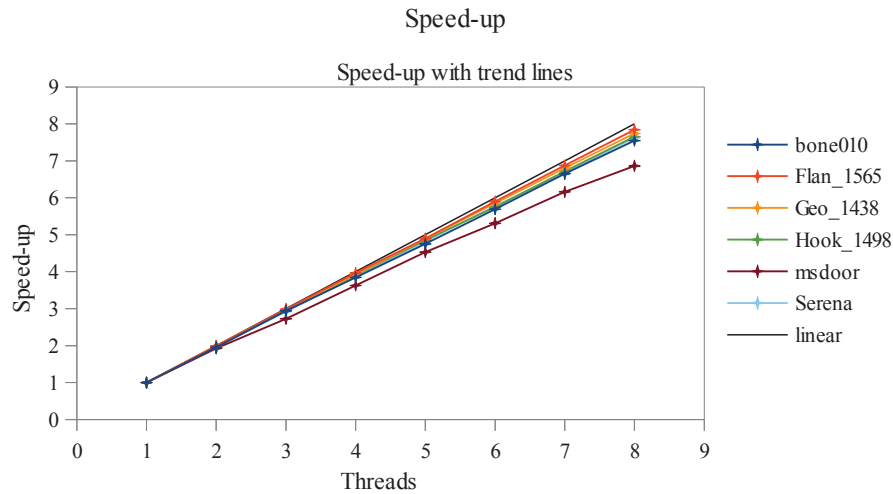


Figure 4.7 Speed-up achieved by the Normann algorithm on semi-big sparse matrices.

In Figure 4.8 the speed-up on multiple threads for the biggest matrices tested are plotted. The speed-up is dependent of the workload in the parallel graph coloring. A comparison between Figure 4.7 and Figure 4.8 shows this feature where the colorings of the bigger matrices renders more linear behavior.



**Figure 4.8** On big matrices almost linear speed-up is achieved by the Normann algorithm.

## 4.6 Parallelism

Amdahl's law is used to make a rough estimate of the parallelism of the Normann implementation. The highest number in Table 4.3 is 0.96, hence the parallel fraction of the Normann implementation is at least 0.96.

**Table 4.3** Estimate of parallel fraction

apache2	bone010	ecology2	Flan_1565	G3_circuit	Geo_1438
0.88	0.96	0.85	0.96	0.83	0.96
Hook_1498	ldoor	msdoor	Serena	StocF-1465	thremal2
0.96	0.96	0.96	0.96	0.94	0.89

## 4.7 Assessment of Conflict Resolution

Since the first phase of the Normann algorithm is based on tentative coloring a test is set-up to quantify the number of conflicts that have to be resolved. Irregularities in the memory latency on each thread affects the iteration through the assigned vertices. Sometimes this irregularity in the iteration increases the risk of two threads causing a conflict. In an attempt to quantify these conflicts a test is performed, where each graph is colored 100 times. The number of conflicts and the number of colorings where these conflicts appear in this test are presented in Table 4.4.

**Table 4.4** Conflicts in 100 colorings

Threads	10		100	
	Colorings*	Conflicts	Colorings*	Conflicts
ecology2	0	0	1	4
msdoor	13	15	44	59
ldoor	6	7	33	43
G3_circuit	5	5	21	26
pwtk	0	0	0	0
thermal2	3	3	9	10
Flan_1565	0	0	44	93
offshore	16	22	52	91
StocF-1465	0	0	23	29
apache2	0	0	6	14
bone010	0	0	86	345
Geo_1438	0	0	20	27
Hook_1498	0	0	26	48
Serena	0	0	26	35
<b>Average</b>	<b>3.3</b>	<b>4</b>	<b>30.1</b>	<b>63.4</b>

*Colorings\* - The number of colorings where conflicts occur out of 100 colorings.*

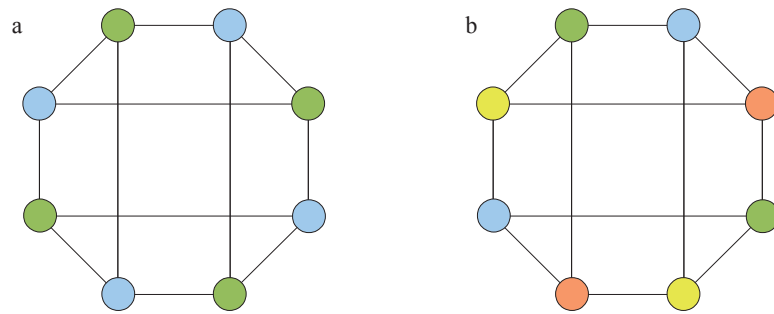
The number of conflicts increases when the number of threads increases, so does the number of colorings with conflicts. On 10 threads the average number of conflicts in 100 colorings is 4 and the average number of colorings with conflicts is 3.3. On 100 threads the number of conflicts average at roughly 63 and the number of colorings with conflicts is 30.1. The number of non zero entries in these test matrices are from 5 to 115 million, see Table 3.1 and 3.2 for details.

The number of conflicts in the coloring phase is very low compared to the non zero entries in a matrix. There is roughly one conflict for every millionth non zero entry. Hence the workload of the conflict resolution is insignificant beside the coloring phase of the Normann implementation. The tests performed confirms that the execution time for the conflict resolution is negligible.

Test data confirms that a conflict resolution is not changing the number of colors in a coloring. The reason for this phenomena is that there are free colors assigned within the interval of the present colors of the conflicting vertices. For example, a graph is sufficiently colored with 4 colors. A conflicting vertex with colors 1,2 and 4 among adjacent vertices is detected and handled. The vertex is assigned color 3 hence the number of global colors is not effected. This color assignment is ensured since the greedy Multi-Coloring algorithm used in the conflict resolution always assigns the lowest numbered colored that is still free.

## 4.8 The Maximum Degree Criterion

In Figure 4.9 an 8 vertices crown graph is colored by the greedy Multi-Coloring algorithm. In the best case scenario the graph is colored by 2 colors. In the worst case the graph is colored with 4 colors. In this example the quality of the coloring rendered is dependent on the numbering of the graph. Both these colorings meets the  $d+1$  criterion, i.e. an equal or lower number of colors than  $d+1$ , where  $d$  is the maximum degree of the graph.



**Figure 4.9** An 8 vertices crown graph colored by the Multi-Coloring algorithm. a) The crown graph is traversed in a way that is beneficial for the quality of the coloring. b) If the numbering is disadvantageous the traversing of the graph is damaging the quality of the coloring.

In this work it is suggested that the  $d+1$  criterion is problematic. If the mean value of  $d$  is used the predicted quality of a graph coloring is more in line with what should be expected from a graph coloring algorithm. Quantitative results confirm this weakness of the  $d+1$  criterion. Throughout the tests performed both the greedy Multi-Coloring and Normann algorithms are consistent below  $\bar{d}+1$  colors. One example of this is the *pwt*k matrix used as a test matrix in this study. The greedy Multi-Coloring implementation uses 48 colors on this graph and the parallel Normann implementation used 42 to 48 colors depending on the number of threads. The degrees of the *pwt*k matrix are  $d=180$  and  $\bar{d}=53.4$ .



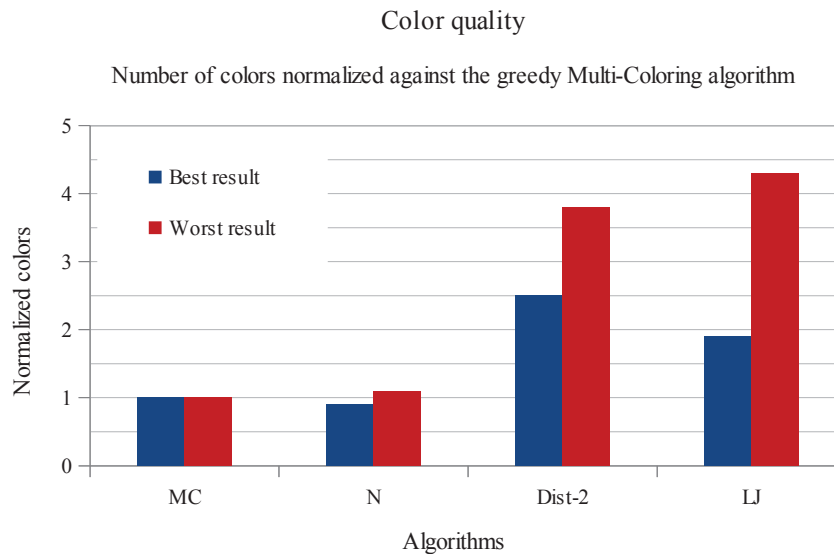


## 5 Summary

The main objective of this work is to expose parallelism in graph coloring and to benchmark known graph coloring algorithms. A secondary aim is to propose new parallel graph coloring algorithms and to benchmark these new algorithms against known graph coloring algorithms. The main implications of this work is discussed in this section.

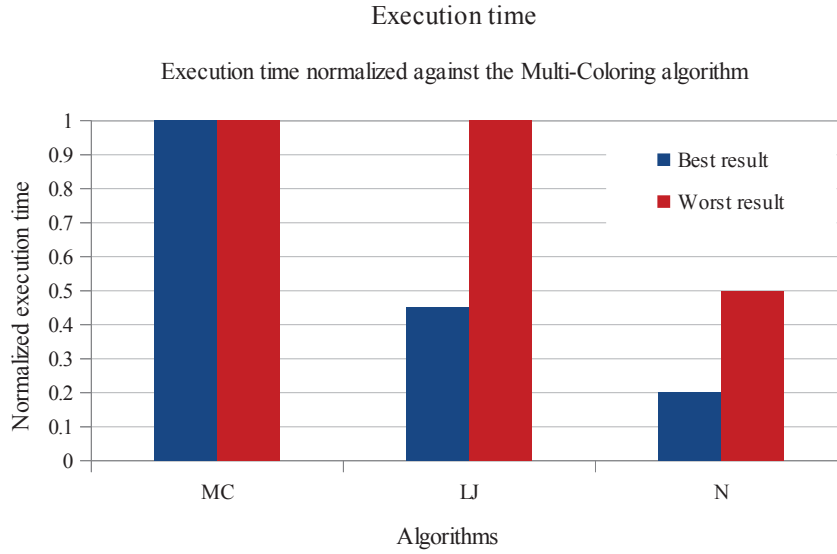
### 5.1 Benchmarks

In Figure 5.1 the color quality achieved by the Normann, the greedy Multi-Coloring, the Luby Jones and the Distance-2 Graph Coloring algorithms is presented. The number of colors presented are best and worst result normalized against the greedy Multi-Coloring algorithm.



**Figure 5.1** The number of colors used by the Normann (N), the greedy Multi-Coloring (MC), the Distance-2 Graph Coloring (Dist-2) [11] and the parallel Luby Jones (LJ) algorithms normalized against the greedy Multi-Coloring algorithm. The number of colors are from colorings of the test graph used in this work, the best and worst result compared to the result of the Multi-Coloring algorithm is presented here.

The benchmarks shows that the Normann algorithm outperforms all tested algorithms on a multi-core hardware. The execution times for the implemented graph coloring algorithms are presented in Figure 5.2. Due to the fact that the graph analysis phase is necessary in modern iterative solvers like Krylov Subspace and Multigrid methods a gained speed-up in the graph analysis phase transpires directly into an over all faster execution time for these iterative solver. The execution times presented in Figure 5.2 are from tests performed on a 8-core CPU. The Normann implementation is 2 to 5 times faster than the greedy Multi-Coloring implementation and up to 5 times faster than the parallel Luby Jones implementation. The Distance-2 Graph Coloring algorithm is not included since the results in [11] is achieved on another type of hardware.



**Figure 5.2** The execution time on a 8 core hardware. The execution times of the Normann (N), the greedy Multi-Coloring (MC), and the parallel Luby Jones (LJ) algorithms are normalized against the greedy Multi-Coloring algorithm. The values of the execution times for the Distance-2 Graph Coloring is from [11] and not used in this comparison. The values of the execution time are from colorings of the test graph used in this work, the best and worst result compared to the result of the greedy Multi-Coloring algorithm is presented here.

## 5.2 High Quality Parallel Graph Coloring

Compared to the greedy Multi-Coloring algorithm the Normann algorithm shows a slight fluctuation in the number of colors used in a coloring. This is due to the differences in how graphs are traversed in parallel by the Normann algorithm compared to the sequential traversing of the greedy Multi-Coloring algorithm. Basically, in some cases local colorings are beneficial for the global parallel coloring and some times not. Both the Normann algorithm and the greedy Multi-Coloring algorithm are consistently rendering high quality colorings.

## 5.3 Conclusions

In this work techniques for parallel graph coloring are proposed and studied. Quantitative results, which represent the number of colors, confirm that the best new algorithm, the Normann algorithm, is performing on the same level as the greedy Multi-Coloring algorithm. Furthermore, in multi-core environments the parallel Normann algorithm fully outperforms the classical greedy Multi-Coloring algorithm for all large test matrices.

With the features of the Normann algorithm quantified and presented in this work it is now possible to perform all phases of iterative solvers like Krylov Subspace and Multigrid methods in parallel.

## 5.4 Future Work

Two of the proposed parallel graph coloring algorithms presented in this thesis are not thoroughly investigated. These two algorithms have high potential, the Normann Lukarski algorithm in particular, and should be benchmarked against the Normann algorithm.

An implementation of the Normann will be available in the Paralution<sup>9</sup> software in the near future.

<sup>9</sup> <http://www.paralution.com/>

## 6 Acknowledgments

I like to thank my supervisor Dimitar Lukarski for asking me to take on this project in the first place. With this project you gave me a new playground that continues to inspire me.

A big thanks to friends and family for putting up with my dwelling on parallel graph coloring. As always, thanks mom and dad. Mom, I finally made it!



## 7 References

### Bibliography

- [1] Saad, Y. *Krylov Subspace Methods on Parallel Computers*, Cite Seer. 1989
- [2] Lukarski, D. *Parallel Sparse Linear Algebra for Multi-core and Many-core Platforms*, PhD Thesis. 2012
- [3] Heuveline, V., Lukarski, D., Weiss, J.P. *Scalable Multi-coloring Preconditioning for Multi-core CPUs and GPUs*, Lecture Notes in Computer Science Volume 6586, 2011, pp 389-397. 2011
- [4] Allwright, R. Bordawekar, P.D. Coddington, K. Dincer, C. Martin, A. *A comparison of parallel graph coloring algorithms*, Cite Seer. 1995
- [5] Kubale, M. *Contemporary Mathematics - Graph Colorings*, American Mathematical Society. 2004
- [6] van Lint, J. H.; Wilson, R. M. *A Course in Combinatorics*, Cambridge University Press. 2001
- [7] Duff, I.S., Erisman, A. M., Reid, J.K. *Direct methods for sparse matrices*, Oxford Science Publications. 1989
- [8] Saad, Y. *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA,. 2003
- [9] Watson Research Center, Et.al. *A string of publications covering handling of sparse matrices.*, Watson Research Center. 1972-1976
- [10] Castonguay, P., Cohen, J. *Efficient Graph Matching and Coloring on the GPU*, Slides to oral presentation. 2012
- [11] Bozdag, D., Catalyurek, U., Gebremedhin, A. H., Manne, F., Boman, E., Özgüner, F. *A Parallel Distance-2 Graph Colouring Algorithm for Distributed Memory computers*, Cite Seer. 2005
- [12] Catalyürek, Ü., Feo, J., Gebremedhin, A., Halappanavar, M., Pothen, P. *Graph Coloring Algorithms for Muti-core and Massively Multithreaded Architectures*, Cornell University Library pdf. 2012
- [13] Gebremedhin, A.H. Manne, F. Pothen, A. *Parallel distance-k coloring algorithms for numerical optimization*, Springer Lecture Notes in Computer Science Volume 2400, 2002, pp 912-921. 2002
- [14] Gebremedhin, A.H. Manne, F. *Scalable parallel graph coloring algorithms*, Concurrency: Practice and Experience 12 (2000) 1131–1146. 2000
- [15] Lumsdaine, A., Gregor, D., Hendrickson, B., Berry, J.W. *Challenges in parallel graph processing*, Parallel Processing Letters, Volume 17, Issue 01, March 2007. 2007
- [16] Chan, W.M., George, A. *A linear time implementation of the reverse cuthill mckee algorithm*, Springer, 1980, Volume 20, Issue 1, pp 8-14. 1980
- [17] Fisher, R.A. Yates, F. *Statistical tables for biological, agricultural and medical research*, Agricultural and Medical Research. 6th Ed. Oliver & Boyd, Edinburgh and London 1963. X, 146 P. Preis 42 s net. 1938
- [18] Gebali, G. *Algorithms and Parallel Computing*, Book, John Wiley & Sons, (2011) 29 mar. 2011