Universität Karlsruhe (TH)
Rechenzentrum
Postfach 6980
76128 Karlsruhe
Germany

Interner Berich Nr. 75/01

Implementation and comparison of three bandwidth optimizing algorithms on a distributed memory parallel computer

Projektgruppe "Numerikforschung für Supercomputer"
Rechenzentrum der Universität Karlsruhe (TH)
September 2000

Author: Detlev Zundel

Contents

Chapter 1

Introduction

Nature laughs at the difficulties of integration.

— Pierre Simon de Laplace

Many technical problems can be described by sets of differential equations. Unfortunately only very few of them can be solved explicitly. The majority however has to be handled numerically. They are usually linearized and discretized by finite differences, finite elements or boundary elements. This step transforms the original problem into a set of linear equations

$$Ax = b \tag{1.1}$$

where x is the solution vector representing the solution for discrete points in the parameter space. Due to the locality of many problems the matrix A will be a sparse matrix with only few non-zero entries coupling neighbouring points of the problem domain.

In general, the matrices generated in the discretization step usually do not have a banded structure, in which the non-zero entries are clustered around the main diagonal. Especially where the discretization meshes or grids for the problem domain are not generated in a uniform way but refined locally, sparse matrices with large bandwidths are generated.

Solving such sparse systems explicitly destroys the sparsity pattern by producing fill-in in former non-zero entries. This additional storage overhead severely limits the applicability of explicit solvers to sparse matrices.

The LINSOL [HSW99b] program package is designed to solve sparse linear systems exceeding the storage capacity of a single computer system on distributed memory parallel computers. It is in the core of the FIDISOL [Sch00], CADSOL [SS92], VECFEM [GRS94], and FDEM [SA00a] systems which transform problems stemming from different fields through a discretization step into a system of linear equations like (1.1).

LINSOL offers ten different storage schemes for sparse matrices allowing it to efficiently adapt to the pattern of the matrix in regard to additional index information overhead [WHS95].

As the fill-in prevents solving equation (1.1) explicitly, LINSOL uses fast iterative solvers giving an approximation x' to the exact solution x satisfying a given upper bound of the residual $r = Ax' - b$. The presently available fourteen generalized CG-solvers in LINSOL include PRES20 [Sch87], BICO [Sch87], AT-PRES also called CGNR [SSW87], and a selction of polyalgorithms trying to automatically select the one best suited to a given problem.

As these solvers use only matrix vector multiplications the pattern of the matrix A is of only indirect significance. It influences the storage mechanisms used internally thereby having only a slight impact on the execution times and the storage requirements.

Despite the selection of iterative solvers available, problems exist for which they fail to converge. To handle these problems an explicit LU decomposition and an incomplete LU preconditioner (ILU) is included in the LINSOL package.

Usually these explicit solvers use a reordering of the matrix such that the produced fill-in is minimized. It is however known that factorizing a matrix using indexed storage mechanisms is inefficient. Therefore the methods deployed in LINSOL use dense rectangular windows into which all non-zero entries of the matrix are unpacked. The processing of these dense windows allows for optimum performance. This requires the minimization of the bandwidth of the matrix.

A permutation matrix P is constructed resulting in PAP^{-1} having less bandwidth than A. It is well known that an algorithm for determining an optimal P in regard to the resulting bandwidth as well as minimizing the fill-in [Yan81] is NP-complete. Only heuristic approaches can therefore be used.

Chapter 2 will introduce the necessary concepts to state the problem formally. Some basics from graph theory used in the following chapters will be given.

An overview over existing heuristic approaches to reduce the bandwidth or the generated fill-in will be given in chapter 3.

Two of these heuristic algorithms and a scheme invented by Schönauer will be examined in more detail in chapter 4. Their working is illustrated by applying them to a common small test matrix.

These algorithms have been implemented in Fortran 90 as modules for the LINSOL system. Details of these implementations for distributed memory parallel computers are given in chapter 5.

The renumbering of a matrix is of course inherently recursive and can thus not be parallelized. The efforts undertaken in the parallel implementations are thus not stimulated by the strive for faster execution times but by the need to handle large problems that can only be handled with distributed memories. In this situation where a sequential algorithm works on distributed data it is of great importance to optimize the communication overhead. This lead to the design of the Schönauer single pass algorithm which is tightly coupled to its efficient implementation on distributed memory parallel computers.

Results of applying the algorithms to matrices from real problems will be given in chapter 6. A summary and possibilities for further work end this exposition in chapter 7.

Chapter 2

Definition of terms

"When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean - neither more nor less." "The question is," said Alice, "whether you can make words mean so many different things." "The question is," said Humpty Dumpty, "which is to be master - that's all."

— Lewis Carroll (Through the Looking Glass)

The reasons for doing bandwidth or related optimizations of matrices are discussed in this chapter. The problem that the algorithms strive to solve is defined formally and the definitions used in the rest of the work are introduced.

2.1 The problem domain

We will discuss systems of linear algebraic equations of the form

$$Ax = b, \qquad \text{where} \;\; A = (a_{ij}),\;\; i, j = 1, \ldots, n. \tag{2.1}$$

Matrices arising from technical problems, such as the discretization of ordinary or partial differential equations, often have only few non-zero entries. These matrices are called *sparse* matrices. Storing all a_{ij}, also called *dense storage*, would waste memory as well as computational resources. Therefore special storage schemes and operations on these have been devised.

It is not exactly clear when a matrix is considered sparse, but a definition given sometimes is that it is sparse when using such special schemes is beneficial for either the required storage or the execution times.

When A is a sparse matrix then our problem is to find a permutation matrix P such that PAP^{-1} has minimal bandwidth. Equation (2.1) then transforms to

$$(PAP^{-1})Px = Pb. \tag{2.2}$$

If all non-zero elements of A are clustered near the main diagonal the matrix is called *banded.*

Definition 1
The *bandwidth* of a general matrix A is $\beta(A) = \max_{a_{ij} \neq 0} |i - j|$.

As the presented algorithms work only for structurally symmetric matrices with a non-zero main diagonal we will subsequently assume A to be of this form if not explicitly stated otherwise and use the following notations considering only the lower left triangular part of the matrix.

Definition 2

- $f_i(A) = min\{j \mid a_{ij} \neq 0\}$. $f_i(A)$ is the index of the column with the first non-zero element of row i.
- $\beta_i(A) = i - f_i(A)$ is the *left bandwidth of row i.*
- Because of the structural symmetry, the *bandwidth of a matrix A* is then equivalent to
  $$\beta(A) = \max_{i=1,\ldots,n} \beta_i(A) \tag{2.3}$$
  and
  $$band(A) = \{(i,j) \mid 0 \leq i - j \leq \beta(A)\} \tag{2.4}$$
  contains all index pairs inside the band limited by the bandwidth of the matrix.

Closely coupled to the bandwidth are the envelope and the profile of a matrix.

Definition 3
The *envelope* of a matrix A is defined by

$$Env(A) = \{(i,j) \mid 0 \leq i - j \leq \beta_i(A)\} \tag{2.5}$$

and the *profile* as

$$Pr(A) = |Env(A)| = \sum_{i=1}^{n} \beta_i(A). \tag{2.6}$$

The sets $band(A)$ and $Env(A)$ have direct implications on possible storage schemes of sparse matrices on a computer system. If $\beta_i(A)$ is almost constant as a function of i then storing all entries in $band(A)$ will be an efficient way of handling the matrices with few index overhead. If this is not the case then storing the entries in a row included in $Env(A)$ as a consecutive vector will be efficient throughout the computation although additional index information has to be included.

Only sparse matrices with a low bandwidth or a low profile can exploit these two storage schemes optimally. This was the stimulation for the research of the bandwidth- and profile reducing algorithms discussed later.

Before introducing the graph theoretical basics we note that a matrix A is *reducible* if and only if there exists a permutation matrix P such that

$$PAP^{-1} = \begin{pmatrix} D_1 & 0 \\ F & D_2 \end{pmatrix}, \tag{2.7}$$

where D_1 and D_2 are square matrices. If A is reducible then equation (2.1) can be solved by solving two independent subproblems of smaller size.

2.2 Basic concepts from graph theory

Considerable insight often can be gained by using a graph representation of sparse matrices. To work with this representation we define a graph as follows.

Definition 4

- If V is a finite nonempty set and $E \subseteq \Big\{\{a,b\} \,|\, a \neq b \text{ and } a,b \in V\Big\}$ is a collection of unordered pairs of elements of V, then $G = (V,E)$ is a finite undirected graph, or simply a *graph*. The elements of V are called the *vertices*, or *nodes*, and the elements of E are called the *edges* of the graph.

- $G' = (V', E')$ is a *subgraph* of G if $V' \subset V$ and $E' \subset E$.

- Let $S \subset V$, the *section graph* $G(S)$ is a subgraph $(S, E(S))$ with

$$E(S) = \{\{x,y\} \in E \,|\, x \in S \wedge y \in S\}. \tag{2.8}$$

Note that loops or multiple edges are excluded from our definition. The following definitions capture local informations in a graph.

Definition 5

- The vertices $x, y \in V$ are *adjacent* if $\{x, y\} \in E$. The adjacency set of a vertex $x \in V$ is defined as

$$Adj(x) = \{y \in V \mid \{x, y\} \in E\}. \tag{2.9}$$

  This notion can be extended to sets of vertices, i.e. for $S \subset V$

$$Adj(S) = \{x \in V \setminus S \mid x \in Adj(y) \wedge y \in S\}. \tag{2.10}$$

- A vertex $x \in V$ is said to be *incident* with an edge $e \in E$ if

$$e = \{x, y\}, \quad y \in V. \tag{2.11}$$

- The *degree* of a vertex x in G is the number of its adjacent vertices in G,

$$deg(x) = |Adj(x)|, \tag{2.12}$$

  where $|\cdot|$ denotes the set cardinality.

- A set $S \subseteq E$ is *independent*, if no two vertices of S are adjacent to each other. If S is an independent set of vertices and there is no independent set of vertices S' with $|S'| > |S|$, then S is called *maximal independent set of vertices.*

By associating a matrix with its corresponding graph we can transform our original problem into the domain of graph theory.

Definition 6

Given a matrix $A = (a_{ij}),\ i, j = 1, \ldots, n$, we define the *corresponding graph* $G(A) = (V, E)$ where V has n vertices $\{v_1, v_2, \ldots, v_n\}$, and $\{v_i, v_j\} \in E$ if $a_{ij} \neq 0$ and $i \neq j$.

Note that this definition disregards the main diagonal and essentially symmetrizes the matrix, i.e. $G(A) = G(A + A^t)$ for every matrix A. Figure 2.1 shows a simplistic example.

For us it is important to note that a permutation of the rows and columns of the matrix A correspond to a renumbering of the vertices in the corresponding graph G as follows.

Definition 7

- A *numbering* g for a graph G with n vertices is defined by a bijective function $g\colon V \to \{1, 2, \ldots, n\}$.

$$A = \begin{pmatrix} x & x & x & 0 \\ x & x & 0 & x \\ x & 0 & x & x \\ 0 & x & x & x \end{pmatrix}$$

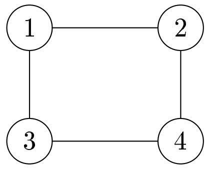

Figure 2.1: Corresponding graph of a matrix

- For a numbering g the *bandwidth of G relative to g* is

$$\beta_g(G) = \max_{\{x,y\} \in E} |g(x) - g(y)|. \tag{2.13}$$

- The minimum of $\beta_g(G)$ over all numberings is called the *bandwidth of G* and is denoted by $\beta(G)$.

The original problem of reducing the bandwidth of a matrix A thus transforms to finding a numbering g under which the graph $G(A)$ has minimal bandwidth.

In order to characterize global properties of graphs we need some additional concepts and definitions.

Definition 8

- Let $x, y \in V$. A *path* of length l from x to y is a set of vertices $\{v_1, v_2, \ldots, v_{l+1}\}$ such that $x = v_1$, $y = v_{l+1}$ and $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq l$.

- A graph is *connected* if for every $x, y \in V$, there exists a path from x to y. This corresponds to the matrix being irreducible.

- A set $S \subset V$ is a *separator* for the connected graph G if $G(V \setminus S)$ has two or more connected components.

- The *distance* $d(x, y)$ between two vertices x and y of G is the length of the shortest path between x and y.

- The *eccentricity* of a vertex is

$$e(x) = max\{d(x, y) \mid y \in V\}. \tag{2.14}$$

- The *diameter* of graph G is

$$\delta(G) = max\{e(x) \mid x \in V\}. \tag{2.15}$$

- A vertex x is *peripheral* if $e(x) = \delta(G)$.
- The term diameter will also be used for a path connecting two peripheral vertices but the distinction should be clear from the context.
- A *pseudo-diameter* is a path connecting two vertices that are at nearly maximal distance apart.
- Let $S \subset V$ and $x, y \in V \setminus S$, x is said to be *reachable* from y through S if there exists a path $\{v_1, v_2, \ldots, v_l\}$ with $x = v_1, y = v_l$ and $v_i \in S$ for $i = 2, \ldots, l-1$.

A central concept for the three algorithms presented in the next chapter is a level structure of a graph G.

Definition 9

- A *level structure* $\mathcal{L}(G)$ of a graph G is a partition of the set V into *levels* $L_1, L_2, \ldots, L_l$ such that

$$\begin{aligned} Adj(L_1) &\subseteq L_2, \\ Adj(L_i) &\subseteq (L_{i-1} \cup L_{i+1}), \qquad i = 2, \ldots, l-1, \\ Adj(L_l) &\subseteq L_{l-1}. \end{aligned} \tag{2.16}$$

- The *width of level* i is $w_i(\mathcal{L}) = |L_i|$, and the width of the level structure is

$$w(\mathcal{L}) = \max_{i=1,\ldots,l} w_i(\mathcal{L}). \tag{2.17}$$

- The number of levels l is called the *depth* of the level structure.
- To each vertex $x \in V$ there corresponds a particular level structure $\mathcal{L}_x(G)$ called the *level structure rooted at* x. Its levels are determined by

$$\begin{aligned} L_1 &= \{x\}, \\ L_i &= Adj(L_{i-1}) \setminus \bigcup_{k=1}^{i-1} L_k, \quad 1 < i \leq e(x). \end{aligned} \tag{2.18}$$

Note that each of the levels L_i $(1 < i < l)$ is a separator for the graph G and that the depth of a level structure rooted at x has the depth $e(x)$. For the matrix and the corresponding graph from figure 2.1 the level structure rooted at vertex 1 would be

$$L_1 = \{1\}, \quad L_2 = \{2, 3\}, \quad \text{and} \quad L_3 = \{4\}.$$

Chapter 3

Overview of optimizing algorithms

For since the fabric of the universe is most perfect and the work of a most wise Creator, nothing at all takes place in the universe in which some rule of maximum or minimum does not appear.

— Leonhard Euler

Different approaches to the bandwidth-, profile-, and fill-in minimizing problems are explained in this chapter. The available space does not permit a formal treatment of all of them. Where this is not possible a motivation why the algorithms work, and references for further reading are given.

3.1 Minimum degree algorithms

The popular *Minimum Degree* algorithm and its derivatives are fill-in minimizing algorithms. They try to locally minimize the fill-in generated in a Gaussian elimination scheme. Let us first consider what elimination of an unknown in a matrix means in the corresponding graph.

3.1.1 Elimination graphs and fill-in

Let us reconsider the sample matrix from figure 2.1. If we were to eliminate the first unknown, corresponding to vertex 1, in a Gaussian scheme or in a factorization process we add multiples of the first row to the rows i where $a_{i1} \neq 0$ producing zeroes in the first column. This will lead to the situation depicted in figure 3.1. New non-zero entries are generated in the places marked as (•). The corresponding new edge in the graph is shown as a dashed line.

In general, eliminating vertex x from the graph $G^{(i)} = (V^{(i)}, E^{(i)})$ yields a new graph $G^{(i+1)} = (V^{(i+1)}, E^{(i+1)})$ by removing the vertex and all its incident edges and by inserting new edges such that all neighbours of x are connected pairwise in the new graph:

$$\begin{aligned} V^{(i+1)} &= V^{(i)} \setminus \{x\}, \\ E^{(i+1)} &= \Big(E^{(i)} \cap \Big\{\{x,y\} \mid y \in Adj(x)\Big\}\Big) \\ &\quad \cup \Big\{\{y,z\} \mid y,z \in Adj(x)\Big\}. \end{aligned} \tag{3.1}$$

$$A = \begin{pmatrix} x & \bullet & x \\ \bullet & x & x \\ x & x & x \end{pmatrix}$$

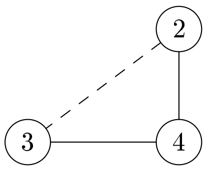

Figure 3.1: Fill-in produced by eliminating vertex 1

3.1.2 Description of the basic variant

As we have seen, an upper bound for the the fill-in produced by eliminating an edge x from the graph G is $\binom{deg(x)}{2}$. The minimum degree algorithm tries to minimize the fill-in by controlling this worst case locally by eliminating vertices with low degree first.

Using the notation of the elimination graphs $G^{(i)} = (V^{(i)}, E^{(i)})$ the algorithm can be stated as follows:

Algorithm M (Minimum degree algorithm). A numbering g is computed for the matrix A minimizing the fill-in.

M1. [Initialization] Build graph $G^{(1)}$ corresponding to the matrix A as described in definition 6 and let $i \leftarrow 1$.

M2. [Elimination] Choose a vertex $x_i \in V$ with $deg(x_i) = \min_{y \in G^{(i)}} deg(y)$. Label this vertex as i by defining $g(x_i) = i$.

M3. [Update] Determine the elimination graph $G^{(i+1)}$ as described in equation (3.1).

M4. [Finished?] If $V^{(i+1)} = \emptyset$ then stop, otherwise set $i \leftarrow i + 1$ and go to M1.

A central problem of this algorithm is that quite often there are several vertices of minimum degree to choose from in step M2. A tie breaking strategy is necessary to resolve the situation. George and Liu [GL89] have shown that the algorithm is unfortunately very sensitive to the strategy applied.

Furthermore a substantial effort of the algorithm is spent in updating the graph. An efficient representation has to be used in order to prevent the growth of the graph representation proportionally to the fill-in. Quotient graphs are usually used for this purpose.

Many variations and improvements of this scheme have been suggested over the years. For an overview of the variations see [GL89]. Most of them are aimed at reducing the execution time of the algorithms on computer systems and not at improving the ordering. One widely used variant, *Multiple Minimum Degree* [Liu85] eliminates maximum independent subsets of vertices in one step thus reducing the updates of the graph. Kumfert and Pothen [KP99] implemented an extensible object oriented library of several variants to ease research of further improvements. Appendix A contains an example of applying the symmetrical Multiple Minimum Degree algorithm to the test matrix used in chapter 4.

3.2 Spectral methods

These profile reduction methods originate not from graph theory but from linear algebra. We will only show the motivation behind them without introducing all necessary concepts formally, and without giving proofs for theorems that can be found in the referenced literature.

3.2.1 Motivation

Rather than working on the corresponding graph of a matrix, these schemes use properties of the Laplacian matrix $La(A) = (l_{ij})$ defined by

$$\begin{aligned} La(A) &= (l_{ij}), \\ l_{ij} &= \begin{cases} -1, & \text{if } i \neq j \wedge a_{ij} \neq 0; \\ -\sum\limits_{k, k \neq i} l_{ik} & \text{if } i = j. \end{cases} \end{aligned} \tag{3.2}$$

This matrix is a singular M-matrix [Meu99], with eigenvalues $\lambda_i \geq 0$, $\lambda_1 = 0$. The eigenvector u_2 corresponding to the smallest positive eigenvalue λ_2 has interesting properties however, see for example [Fie75].

If we define $row(i) = \{j \mid a_{ij} \neq 0,\ 1 \leq j \leq i\}$ the profile of A is

$$Pr(A) = \sum_{i=1}^{n} \beta_i(A) = \sum_{i=1}^{n} \max_{j \in row(i)} (i - j). \tag{3.3}$$

The work in the Cholesky factorization is proportional to

$$W = \sum_{i=1}^{n} \max_{j \in row(i)} (i - j)^2. \tag{3.4}$$

This is related to the 2-sum

$$\sigma_2^2(A) = \sum_{i=1}^{n} \sum_{j \in row(i)} (i - j)^2. \tag{3.5}$$

George and Pothen [BPS95] give a proof for the following theorem.

Theorem 1
Let p be the maximum number of off-diagonal non-zeros in a row of A (or the maximum degree in the corresponding graph G), then

$$W \leq \sigma_2^2(A) \leq pW. \tag{3.6}$$

The spectral ordering is obtained by sorting the components of u_2 in increasing order. This yields a permutation vector giving an ordering that reduces the envelope. In order to justify this, theorem 1 allows us to consider $\sigma_2^2(A)$ instead of W to minimize over all orderings. For the sake of simplicity, suppose that n is even and let $\mathcal{P}$ be the set of vectors whose components are permutations of $\{-n/2, \ldots, -1, 0, 1, \ldots, n/2\}$. Then

$$\min_{x \in \mathcal{P}} \sum_{i=1}^{n} \sum_{j \in row(i)} (x_i - x_j)^2 = \frac{1}{2} \min_{x \in \mathcal{P}} \sum_{a_{ij} \neq 0} (x_i - x_j)^2. \tag{3.7}$$

The condition on x is relaxed to obtain an easier continuous problem. We now consider the set $\mathcal{H}$ of vectors x such that $\sum x_i = 0$ and (x, x) given. This yields

$$\frac{1}{2} \min_{x \in \mathcal{H}} \sum_{a_{ij} \neq 0} (x_i - x_j)^2 = \min_{x \in \mathcal{H}} (x, La(A)x) = \lambda_2(u_2, u_2). \tag{3.8}$$

Barnard, Pothen and Simon [BPS95] have shown that the permutation vector induced by u_2 is in the Euclidean norm the closest vector in $\mathcal{P}$ to u_2.

3.2.2 Multilevel spectral algorithms

As the computation of the second eigenvector, even if it does not need to be carried out with great accuracy, is a costly operation, the algorithm above cannot compete with other profile reduction algorithms for large examples. In order to overcome this, Barnard, Pothen and Simon [BPS95] propose a *multilevel spectral algorithm* that works on a contracted graph with much fewer vertices than the original graph instead.

For this a series of coarser and coarser graphs, that retain the structure of the original graph, is constructed. At a certain stage the second eigenvector of the coarsest graph is computed and interpolated backwards to the series of graphs until the original graph is reached. This interpolated solution is the output of the algorithm.

There are many ways to define a graph contraction. The algorithm proposed by Barnard, Pothen, and Simon finds a maximal independent set of vertices which are to be the vertices of the contraction. The edges are found by growing domains from the selected vertices adding an edge when two domains intersect.

3.3 Graph partitioning algorithms

These fill-in minimizing algorithms are based on the fill-in characterization given by George [GL81]. If x_i and x_j are vertices in a graph G, one can prove that there cannot be any fill-in between x_i and x_j if, on every path from x_i to x_j in G, there is a vertex with a number greater than x_i and x_j.

For a separator S of the graph G splitting the graph into A and B, it is clear from this observation that if we first number the vertices in A, then in B, and the vertices in S last, there cannot be any fill-in between vertices in A and B. Applying this scheme recursively to A and B yields an ordering minimizing the fill-in.

3.3.1 Nested Dissection

A separator for a rectangular grid mesh is for example a vertical strip of the grid. In the second iteration a horizontal strip of the grid can be used as a separator. The nested dissection algorithm introduced by George [Geo73] alternates the directions of successive separations in this way.

3.3.2 Other partitioning schemes

For a general sparse matrix with no underlying mesh it is not so obvious to find a small separator partitioning the graph in two or more components of almost an equal number of vertices. Many ways have been devised to handle this problem.

One of the best known algorithms for graph bisection was given by Kernighan and Lin [KL70]. It is a greedy, local optimization strategy. Vertices are moved between sets in an effort to reduce the number of edges cut by the partition.

Recently Hendrickson and Leland proposed a Multilevel-KL algorithm [HL95] similar to the idea of the multilevel spectral scheme. A sequence of increasingly smaller graphs approximating the original graph is constructed and the smallest graph is partitioned. This partition is projected back through the intermediate levels. The Kernighan-Lin algorithm is invoked every few levels of projection to refine the partition.

3.4 Level structure based algorithms

Level structures can also be used for graph partitioning as we noted earlier that each inner level is a separator of the graph. They are more often used for minimizing the bandwidth however.

All three algorithms further examined here belong to this class. The basic idea is that renumbering vertices in the order they appear in a generated level structure will yield a low bandwidth.

The algorithms can be differentiated by the way they arrive at a level structure $\mathcal{L}(G)$ and how they label the vertices once they determined one. This will be discussed in depth in the next chapter.

Chapter 4

Three level structure based algorithms in detail

Science is what we understand well enough to explain to a computer. Art is everything else we do.

— Donald E. Knuth

A complete description of the Reverse Cuthill-McKee algorithm, a variation of it invented by Schönauer, and the Gibbs-Poole-Stockmeyer algorithm will be given in this chapter.

4.1 The test matrix

To illustrate their working we will applicate each of the three algorithms to the symmetric 24×24 test matrix A_t given in figure 4.1. The bandwidth of this matrix is $\beta_t = 21$.

As can be seen from the corresponding graph in figure 4.2 this matrix might be the result of a two-dimensional discretization on a 4×6 grid and is thus representative for the matrices LINSOL is used for.

Whenever the vertices in the graph are represented by circles, the integers in the circles are the numbers of the vertices. When they are represented by rectangles, the integers in the rectangles are the level numbers generated by the algorithms.

$$
\begin{bmatrix}
x & & & & & x & x & & & & & & x & x & & & & x & x & x & & x & & \\
& x & & & & & & & x & x & & & & x & x & & & x & & & & & & \\
& & x & & & & x & & & & & x & & & & & & & & & x & x & x & \\
& & & x & & & & x & & & x & & & & & x & & & x & x & & & & \\
& & & & x & & & x & & x & & & & & x & x & x & & & & & & & \\
x & & & & & x & x & & & & & & x & x & & & & x & & & & & & \\
x & & x & & & x & x & & & & & x & x & & & & & & & x & & x & & x \\
& & & x & x & & & x & & x & & & & & x & x & x & x & x & & & & & \\
& x & & & & & & & x & x & & & & & x & & & & & & & & & \\
& x & & & x & & & x & x & x & & & & x & x & & & x & x & & & & & \\
& & & x & & & & & & & x & & & & & & & & x & x & x & x & & \\
& & x & & & & x & & & & & x & x & & & & & & & & & x & & x \\
x & & & & & x & x & & & & & x & x & & & & & & & & & & & x \\
x & x & & & & x & & & & x & & & & x & & & & x & & & & & & \\
& x & & & x & & & x & x & x & & & & & x & & & & & & & & & \\
& & & x & x & & & x & & & & & & & & x & x & & x & & & & & \\
& & & & x & & & x & & & & & & & & x & x & & & & & & & \\
x & x & & & & x & & x & & x & & & & x & & & & x & x & x & & & & \\
x & & & x & & & & x & & x & x & & & & & x & & x & x & x & & & & \\
x & & & x & & & x & & & & x & & & & & & & x & x & x & x & x & & \\
& & x & & & & & & & & x & & & & & & & & & x & x & x & x & \\
x & & x & & & & x & & & & x & x & & & & & & & & x & x & x & x & \\
& & x & & & & & & & & & & & & & & & & & & x & x & x & \\
& & & & & & x & & & & & x & x & & & & & & & & & & & x
\end{bmatrix}
$$

Figure 4.1: Pattern of the test matrix A_t

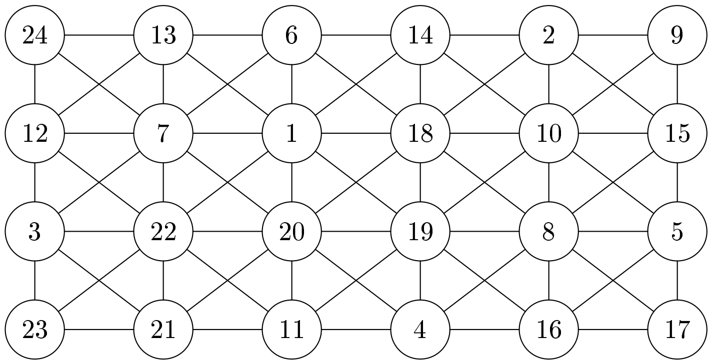

Figure 4.2: Graph of the test matrix A_t

4.2 General results for level structuring schemes

As all three algorithms work on level structures as introduced in definition 9 we can prove properties common to them. The following lemma appears in [GPS76] without a proof. We will give our own proof as the concepts are central to the following discussion.

Lemma 2
For any level structure $\mathcal{L}$ a numbering g_L assigning consecutive integers to vertices level by level, first to the vertices of L_1, then to those of L_2 and so forth, yields a bandwidth β_{g_L} satisfying

$$\beta_{g_L} \leq 2w(\mathcal{L}) - 1. \tag{4.1}$$

If the level structure $\mathcal{L}$ is in addition rooted then we also have

$$\beta_{g_L} \geq w(\mathcal{L}). \tag{4.2}$$

Proof. From the definition follows that for a level structure $\mathcal{L}$ with depth $k > 1$ the maximum in equation (2.13) is taken over sets of vertices from the same or from adjacent levels. If they are in the same level i then

$$|g(x) - g(y)| < w_i(\mathcal{L}) \leq w(\mathcal{L}). \tag{4.3}$$

Otherwise let $x \in L_i$ $(1 \leq i < k)$, and $y \in L_{i+1}$ then

$$\begin{aligned} |g(x) - g(y)| &\leq \Big(w_i(\mathcal{L}) + w_{i+1}(\mathcal{L})\Big) - 1 \\ &\leq 2\Big(\max\{w_i(\mathcal{L}), w_{i+1}(\mathcal{L})\}\Big) - 1 \\ &\leq 2w(\mathcal{L}) - 1. \end{aligned} \tag{4.4}$$

Together this yields equation (4.1).

If a level structure is rooted then for every $x \in L_i, 1 \leq i < k$ there exists a vertex $y \in L_{i+1}$ such that $\{x, y\} \in E$. Now let x be the vertex with the highest number in level L_m such that $w_m(\mathcal{L}) = w(\mathcal{L})$, then there exists a vertex $y \in L_{m-1}$ with $\{x, y\} \in E$ and $|g(x) - g(y)| \geq w_m(\mathcal{L}) = w(\mathcal{L})$. This proves equation (4.2). ∎

This result motivates the search for level structures of low width occurring in all the algorithms.

4.3 The Reverse Cuthill-McKee algorithm

The algorithm was first given by Cuthill and McKee in [CM69]. In contrast to the fill-in reducing strategies this algorithm aims at reducing the bandwidth and the profile of the matrix.

4.3.1 Description

Algorithm R (Reverse Cuthill-McKee algorithm, in the following denoted by RCM). A numbering h is computed for the irreducible matrix A minimizing the bandwidth and the profile.

R1. [Initialization] Build graph G corresponding to the matrix A as described in definition 6 and compute the values

$$d_{min} = \min_{x \in V}\{deg(x)\}, \tag{4.5}$$

$$d_{max} = \max_{x \in V}\{deg(x)\}, \tag{4.6}$$

$$d_{median} = \operatorname*{median}_{x \in V}\{deg(x)\}. \tag{4.7}$$

R2. [Generate level structure] For each vertex of low degree generate the corresponding rooted level structure and determine its width. Normally low degree here means less than or equal to d_{cand} with

$$d_{cand} = \max\Big\{\min\{(d_{max} + d_{min})/2, d_{median} - 1\}, d_{min}\Big\}, \tag{4.8}$$

although this can be controlled somewhat by parameters.

R3. [Generate numbering] For each rooted level structure of minimal width generated in step R2, define the numbering g on the graph G by consecutively assigning positive integers level by level according to the following procedure:

1. Assign the number 1 to the root vertex.
2. For each successive level, beginning with level 2, first number the vertices adjacent to the lowest numbered vertex of the preceding level, in order of increasing degree. Ties are broken arbitrarily. The remaining vertices adjacent to the next lowest numbered vertex of the preceding level are numbered next, again in order of increasing degree. Continue this process until all vertices of the current level are numbered, then proceed with the next level. The process terminates when all levels have been numbered.

R4. [Select best numbering] For each numbering g from step (R3-2) compute the corresponding bandwidth $\beta_g(G)$. Select the numbering producing the smallest bandwidth.

R5. [Reverse the numbering] Generate the reverse numbering h by defining $h(x) = n + 1 - g(x)$, for each $x \in V$.

George [Geo71] first suggested step R5 after he observed that reversing the numbering frequently reduced the profile further. As we will see it can never increase the profile and it has of course no effect on the bandwidth as

$$|h(x) - h(y)| = |n + 1 - g(x) - (n + 1 - g(y))| = |g(x) - g(y)|. \tag{4.9}$$

The Cuthill-McKee algorithm defined by (R1–R4) only, will subsequently be referred to as CM.

4.3.2 Reversing the numbering may decrease the profile

We will now prove that reversing the numbering cannot increase the profile of the resulting matrix. This result shows that the reverse Cuthill-McKee is always as good as the original algorithm so there is no reason not to use it.

Although this proof was given earlier [Meu99] it is instructive to restate it here as the applicability to the Schönauer algorithm will then become clear.

Theorem 3
Let A be an irreducible matrix and $A^{(CM)}$ be the matrix corresponding to reordering the graph of A by the Cuthill-McKee algorithm. Then for $A^{(CM)}$

$$\forall i, j,\ i \leq j : \quad f_i \leq f_j. \tag{4.10}$$

Proof. If the conclusion does not hold then there exists a column k and rows $p < l < m$ such that

$$\begin{aligned} f_p &\leq k, \\ f_l &> k, \\ f_m &\leq k. \end{aligned} \tag{4.11}$$

For the corresponding graph this means

$$\begin{aligned} a_{p,k} \neq 0 &\Rightarrow x_p \in Adj(x_k), \\ a_{l,k} = 0 &\Rightarrow x_l \notin Adj(x_k), \\ a_{m,k} \neq 0 &\Rightarrow x_m \in Adj(x_k). \end{aligned} \tag{4.12}$$

This however is not possible as RCM has numbered all vertices adjacent to x_k before numbering any other vertices. ∎

Definition 10
The *transpose envelope* of a matrix A is defined by

$$Tenv(A) = \{(i,j) \mid j \leq i \wedge \exists k \geq i : a_{kj} \neq 0\}. \tag{4.13}$$

The matrix $A^{(RCM)}$ corresponding to the reverse numbering is correlated to $A^{(CM)}$ by

$$\begin{aligned} a_{i,j}^{(RCM)} \neq 0 \quad &\Longleftrightarrow \quad a_{n+1-i,n+1-j}^{(CM)} \neq 0 \\ &\overset{A^t=A}{\Longleftrightarrow} \quad a_{n+1-j,n+1-i}^{(CM)} \neq 0 \end{aligned} \tag{4.14}$$

This shows that the inverted last row of $A^{(CM)}$ becomes the first column of $A^{(RCM)}$ and so forth. From the definition of $Env(A)$ and $Tenv(A)$ follows directly

$$|Env(A^{(RCM)})| = |Tenv(A^{(CM)})|. \tag{4.15}$$

Theorem 4
The transpose envelope of $A^{(CM)}$ is a subset of the envelope of $A^{(CM)}$, i.e.

$$Tenv(A^{(CM)}) \subseteq Env(A^{(CM)}). \tag{4.16}$$

Proof. Suppose we have $(i,j) \in Tenv(A^{(CM)})$ and $(i,j) \notin Env(A^{(CM)})$. From definition 10 follows: $\exists k \geq i$ such that $a_{kj} \neq 0$. Either

1. $a_{ij} \neq 0 \Rightarrow (i,j) \in Env(A^{(CM)})$ which contradicts the premise, or

2. $a_{ij} = 0$. If $(i,j) \notin Env(A^{(CM)}) \Rightarrow \forall l \leq j : a_{il} = 0 \Rightarrow f_i > j$. On the other hand, we have $a_{kj} \neq 0 \Rightarrow f_k \leq j$. Together this implies $f_k < f_i$ which is impossible by $k \leq i$ and theorem 3. ∎

Theorem 4 implies

$$|Env(A^{(RCM)})| \leq |Env(A^{(RCM)})|, \tag{4.17}$$

i.e. the profile can only be decreased by reversing the numbering. Relevant for the proof is the fact that all vertices in an adjacency set are numbered consecutively before considering any other vertices. We will see that the Schönauer algorithm also has this property.

4.3.3 Application to the test matrix

In this section we consider the application of the algorithm to the test matrix A_t and its corresponding graph given in figure 4.2.

As $d_{median} = 5$ for this graph and $(d_{max} + d_{min})/2 = 5.5$, the RCM algorithm generates level structures rooted at the vertices 9, 17, 23 and 24. The level structure rooted at vertex 9 is shown in figure 4.3.

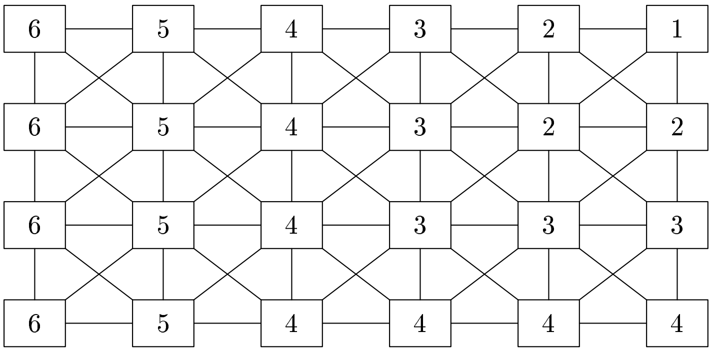

Figure 4.3: Level structure rooted at vertex 9

The other three level structures have of course a similar form. Using the level structure from figure 4.3, we obtain the reverse numbering shown in figure 4.4.

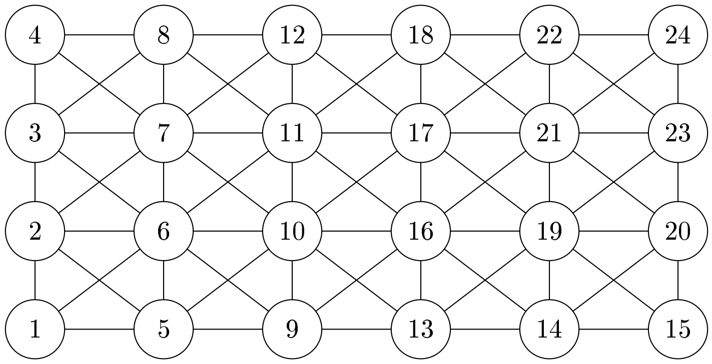

Figure 4.4: Reverse numbering for start vertex 9

For all four level structures the width and the bandwidth are both 7. The edges responsible for this bandwidth are the connections from the vertices 16, 17, and 18 in level 3 to the vertices 9, 10, and 11 respectively in level 4. The resulting matrix shown in figure 4.5 has a profile of 97. The three edges responsible for the bandwidth can be perceived as a thickening in the matrix pattern.

Note that in this example the tie arising when numbering vertices 2 and 15 has been resolved in favour of vertex 15. The implementation given in chapter 5 will favour vertex 2 resulting in a bandwidth of 8 for start vertex 9. It will however resolve the ties for start vertex 24 with more luck and output an equivalent numbering in respect to the pattern and the bandwidth of the permuted matrix.

$$\begin{bmatrix}
x&x&&&x&x&&&&&&&&&&&&&&&&&&\\
x&x&x&&x&x&x&&&&&&&&&&&&&&&&&\\
&x&x&x&&x&x&x&&&&&&&&&&&&&&&&\\
&&x&x&&&x&x&&&&&&&&&&&&&&&&\\
x&x&&&x&x&&&x&x&&&&&&&&&&&&&&\\
x&x&x&&x&x&x&&x&x&x&&&&&&&&&&&&&\\
&x&x&x&&x&x&x&&x&x&x&&&&&&&&&&&&\\
&&x&x&&&x&x&&&x&x&&&&&&&&&&&&\\
&&&&x&x&&&x&x&&&x&&&x&&&&&&&&\\
&&&&x&x&x&&x&x&x&&x&&&x&x&&&&&&&\\
&&&&&x&x&x&&x&x&x&&&&x&x&x&&&&&&\\
&&&&&&x&x&&&x&x&&&&&x&x&&&&&&\\
&&&&&&&&x&x&&&x&x&&x&&&x&&&&&\\
&&&&&&&&&&&&x&x&x&x&&&x&x&&&&\\
&&&&&&&&&&&&&x&x&&&&x&x&&&&\\
&&&&&&&&x&x&x&&x&x&&x&x&&x&&x&&&\\
&&&&&&&&&x&x&x&&&&x&x&x&x&&x&x&&\\
&&&&&&&&&&x&x&&&&&x&x&&&x&x&&\\
&&&&&&&&&&&&x&x&x&x&x&&x&x&x&&x&\\
&&&&&&&&&&&&&x&x&&&&x&x&x&&x&\\
&&&&&&&&&&&&&&&x&x&x&x&x&x&x&x&x\\
&&&&&&&&&&&&&&&&x&x&&&x&x&x&x\\
&&&&&&&&&&&&&&&&&&x&x&x&x&x&x\\
&&&&&&&&&&&&&&&&&&&&x&x&x&x
\end{bmatrix}$$

Figure 4.5: Resulting matrix for the RCM algorithm

4.3.4 Discussion

The major problem of RCM is of course the explicit search for a narrow level structure over many possible start vertices. In the worst case a level structure for each vertex of the graph has to be generated, when all of them have the same degree.

The nearly exhaustive search compensates however for the sensitivity of the bandwidth to the choice of a start vertex.

Another problem is the lower bound on the bandwidth given in equation (4.2) for rooted level structures. This is a limitation as the minimum bandwidth of a graph can be considerably less than the width of any rooted level structure. Our test matrix used throughout this chapter is a good example of such a case.

4.4 The Schönauer single pass algorithm

Although developed independently, the Schönauer single pass algorithm [Sch99] can be seen as a modification to the RCM algorithm. As its application on distributed memory parallel computers influenced its design, the algorithm explicitly omits the search over possible start vertices. This is a deliberate decision trading the attainable bandwidth against execution time and communication overhead. If the resulting bandwidth is good enough to allow the subsequent solvers to complete the computation without exceeding the available memory, then this algorithm is intended to be the cheapest of the examined schemes. We will see in chapter 6 that this is substantiated.

Just as in the RCM algorithm the numbering is based on a rooted level structure of the corresponding graph. This level structure is not generated explicitly however. The numbering of the vertices is done in steps corresponding to the levels of the implicit rooted level structure determined by the choice of the start vertex. This incurs a very low memory consumption.

The numbering phase augments the process of ordering vertices to be labelled with an additional heuristic. Instead of just sorting those vertices by degree, their edges to vertices in the previous level are also consulted to decide on an ordering.

4.4.1 Description

Algorithm S (Schönauer single pass algorithm, in the following denoted by SSP). A numbering h is computed for the irreducible matrix A minimizing the bandwidth and the profile.

S1. [Initialization] Build graph G corresponding to the matrix A as described in definition 6.

S2. [Choose start vertex] Choose a vertex with minimum degree as the starting vertex x of the numbering process and assign it the number 1. Ties are broken arbitrarily.

S3. [Generate next level] Build the list of vertices adjacent to vertices having been labelled in the previous level (either in step S5 or S2) that have not been labelled themselves.

S4. [Finished?] If the list of vertices to be labelled is empty then goto S6.

S5. [Assign numbers to next level] Process the vertices that have been labelled in the previous level in the order of their newly assigned numbers. For the vertex with the new number x assign consecutive integers to its adjacent vertices in the next level in the following order:

1. Label vertices being adjacent to x, $x+1$ and $x+2$ first,
2. then label vertices being adjacent only to x and $x+1$,
3. finally label vertices being adjacent to x only.

If more than one vertex complies to a criterion, they are numbered in order of increasing degree. Continue with step S3.

S6. [Reverse the numbering] Generate the reverse numbering h generated in step S5 by defining $h(x) = n + 1 - g(x)$, for each $x \in E$.

As can be seen from the description of step S5, this algorithm also numbers all vertices adjacent to a previously numbered vertex x before considering other vertices. Having this property, the proof given in section 4.3.2 is also applicable for this algorithm. Therefore the reverse numbering is also computed in step S6 and used as the result.

4.4.2 Application to the test matrix

Again, as for RCM, vertex 9 is chosen as the starting vertex. The different vertex selection first comes into play in level 4 where an internal vertex is numbered first despite its higher degree. The resulting numbering is show in figure 4.6.

Applying this numbering to the original matrix yields the matrix in figure 4.7 with bandwidth $\beta = 9$.

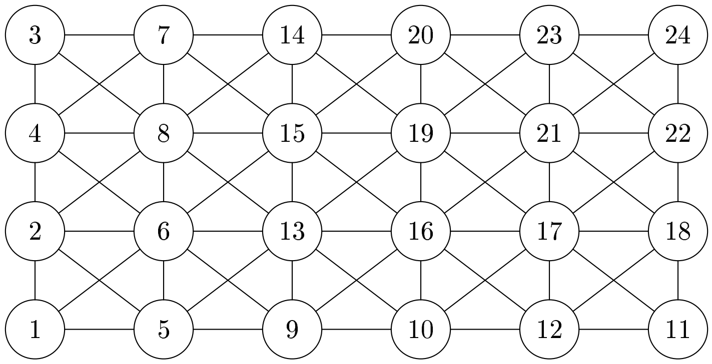

Figure 4.6: Reverse numbering for the Schönauer algorithm

$$
\begin{bmatrix}
x&x&&&x&x&&&&&&&&&&&&&&&&&&\\
x&x&&x&x&x&&x&&&&&&&&&&&&&&&&\\
&&x&x&&&x&x&&&&&&&&&&&&&&&&\\
&x&x&x&&x&x&x&&&&&&&&&&&&&&&&\\
x&x&&&x&x&&&x&&&&x&&&&&&&&&&&\\
x&x&&x&x&x&&x&x&&&&x&&x&&&&&&&&&\\
&&x&x&&&x&x&&&&&&x&x&&&&&&&&&\\
&x&x&x&&x&x&x&&&&&x&x&x&&&&&&&&&\\
&&&&x&x&&&x&x&&&x&&&x&&&&&&&&\\
&&&&&&&&x&x&&x&x&&&x&x&&&&&&&\\
&&&&&&&&&&x&x&&&&&x&x&&&&&&\\
&&&&&&&&&x&x&x&&&&x&x&x&&&&&&\\
&&&&x&x&&x&x&x&&&x&&x&x&&&x&&&&&\\
&&&&&&x&x&&&&&&x&x&&&&x&x&&&&\\
&&&&&x&x&x&&&&&x&x&x&x&&&x&x&&&&\\
&&&&&&&&x&x&&x&x&&x&x&x&&x&&x&&&\\
&&&&&&&&&x&x&x&&&&x&x&x&x&&x&x&&\\
&&&&&&&&&&x&x&&&&&x&x&&&x&x&&\\
&&&&&&&&&&&&x&x&x&x&x&&x&x&x&&x&\\
&&&&&&&&&&&&&x&x&&&&x&x&x&&x&\\
&&&&&&&&&&&&&&&x&x&x&x&x&x&x&x&x\\
&&&&&&&&&&&&&&&&x&x&&&x&x&x&x\\
&&&&&&&&&&&&&&&&&&x&x&x&x&x&x\\
&&&&&&&&&&&&&&&&&&&&x&x&x&x
\end{bmatrix}
$$

Figure 4.7: Resulting matrix for the Schönauer algorithm

4.4.3 Discussion

Clearly the same argument as in section (4.3.4) concerning the attainable bandwidth holds. The choice of the starting vertex determines the result. Omitting the search over possible start vertices however makes this the fastest algorithm from the three examined.

As was discussed for RCM already, the attainable bandwidth is very sensitive to the selection of a start vertex. As the width of the level structure is a lower bound for the bandwidth, selecting a vertex that yields a narrow level structure is the optimal choice. As this cannot be predicted in advance, a vertex with minimum degree is chosen to minimize the width of level 2. Looking further ahead is not possible without compromising the efficiency.

We will see how the GPS algorithm faces these problems inherent in rooted level structures.

The different numbering strategies

RCM and SSP use two different strategies for assigning numbers to the vertices in the next level. They both use a local strategy that is supposed to result in good global results.

With the information of only the two levels available, the best strategy would of course be to assign numbers to neighbours of vertex with the lowest new number in the previous level, as connections from the beginning of a level to vertices near the end of the next level result in the greatest bandwidth. It is therefore best to have vertices with few forward connections at the beginning of a level, which will achieve good results in the next iteration.

Both strategies discussed here only approximate this. They do of course number vertices first that are adjacent to the vertices at the beginning of the previous level. What they both do not compute is the number of forward connections of the candidates. RCM uses the degree of a vertex as an upper bound for this and SSP uses information about some backward connections instead of considering the degree only.

The different choices of the vertices to be numberd in the next level lead to only slight differences in a direct comparison, i.e. when RCM is also applied to only one start vertex. Usually they are equally good, but cases exist for which the one or the other is better. The difference is less than 0.5% in these cases.

4.5 The GPS algorithm

The GPS algorithm [GPS76] tries to improve on the inefficiency of RCM in two different ways. First some effort is invested into the selection of good start vertices for building a level structure. This effort allows for omitting the exhaustive search utilized in RCM. Instead more general level structures are considered and used for a numbering free of the lower bound from equation (4.2). Both RCM and the SSP algorithm utilize only rooted level structures.

4.5.1 Description

Algorithm G (Gibbs-Poole-Stockmeyer algorithm, in the following denoted by GPS). A numbering h is computed for the irreducible matrix A minimizing the bandwidth and the profile.

G1. [Initialization] Build graph $G^{(0)} = (V^{(0)}, E^{(0)})$ corresponding to the matrix A as described in definition 6.

Finding a pseudo diameter

G2. [Pick a start vertex] Select an arbitrary vertex x with minimal degree and set $v \leftarrow x$.

G3. [Generate forward level structure] Generate the rooted level structure $\mathcal{L}_v$. Let S be the set of vertices which are in the last level, i.e.

$$S = L_{e(v)}. \tag{4.18}$$

G4. [Generate backward level structure] In order of increasing degree, generate for each vertex $s \in S$ the rooted level structure $\mathcal{L}_s$. If for some s the depth of $\mathcal{L}_s$ is greater than the depth of $\mathcal{L}_v$, then set $v \leftarrow s$ and continue with step G3.

G5. [Determine best backward level structure] Let u be the vertex of S whose associated level structure has smallest width, with ties broken arbitrarily.

The first phase of the algorithm stops with the vertices u and v the endpoints of a pseudo-diameter and their associated rooted level structures.

Although it is not evident from the description how many iterations are necessary to arrive at a pseudo-diameter, the tests show that very rarely more than two are required.

Minimizing the level width

The two rooted level structures $\mathcal{L}_u = \{L_1, L_2, \ldots, L_k\}$ and $\mathcal{L}_v = \{M_1, M_2, \ldots, M_k\}$ are now combined to produce a level structure $\mathcal{L} = \{N_1, N_2, \ldots, N_k\}$ that usually has less width than either of the two rooted level structures.

The assignment to the new level is carried out according to the following procedure.

G6. [Assign level pairs] Assign to each vertex $w \in E$ the ordered pair (i, j) such that $w \in L_i$ and $w \in M_{k+1-j}$. The pair (i, j) is called the *associated level pair* of the vertex w. Note that the pair $(1, 1)$ is associated with vertex v from the first phase and (k, k) with the vertex u.

G7. [Assign uniquely located vertices] If the associated level pair of a vertex w is of the form (i, i) then vertex w is placed in N_i. All vertices satisfying this condition and their incident edges are removed from the graph, yielding the new graph $G^{(1)} = (V^{(1)}, E^{(1)})$. If $V^{(1)} = \emptyset$ then goto step G10.

G8. [Determine connected components] The graph $G^{(1)}$ now consists of a set of one or more disjoint connected components. Determine the sets of vertices $C_1, C_2, \ldots, C_t$ for these components, such that $|C_1| \geq |C_2| \geq \cdots \geq |C_t|$.

G9. [Assign the connected components] For each set of vertices of the connected components $C_i, i = 1, \ldots, t$, do the following:

1. Compute the vector $n = (n_1, n_2, \ldots, n_k)$ of the level widths of the level structure that is being constructed, i.e. $n_i = |N_i|$.
2. Compute the vectors $h = (h_1, h_2, \ldots, h_k)$ and $l = (l_1, l_2, \ldots, l_k)$ representing the level widths after hypothetically assigning the vertices according to the first or the second element of the associated level pairs, i.e. $h_i = n_i +$ (the number of vertices which would be placed in N_i if the first elements of the associated level pairs were used) and $l_i = n_i +$ (the number of vertices which would be

placed in N_i if the second elements of the associated level pairs were used).

3. Find the maximum width of the levels that were changed by these hypothetical assignments by computing

$$\begin{aligned} h_0 &= \max_{i=1,\ldots,k} \{h_i \mid h_i - n_i > 0\}, \text{ and} \\ l_0 &= \max_{i=1,\ldots,k} \{l_i \mid l_i - n_i > 0\}. \end{aligned} \tag{4.19}$$

4. If $h_0 < l_0$, place all vertices of the connected component in the levels indicated by the first elements of the associated level pairs.
5. If $l_0 < h_0$, use the second elements of the associated level pairs to assign the vertices.
6. If $h_0 = l_0$, then use the elements of the level pairs which arise from the rooted level structure of smaller width. If the widths are also equal then use the first elements.

The stage ends with a level structure for the graph G. Note that this level structure is of a more general type than the rooted level structure. Step G9 assigns the vertices where the two level structures differ, in such a way that the result has the smallest possible width for the two assignment choices.

Note also that although suggested by the terms used in the description, this stage does indeed yield a level structure as defined in definition 9. As the connected components are assigned completely according to one of the associated level pairs, the edges internal to the connected component of course connect vertices in the same or in adjacent levels. The edges external to such a connected component also satisfy this condition as per construction external adjacent vertices have the same level number in both rooted level structures.

With an eye on the implementation, we remark that step G8 is the only place where explicit parallelism can improve the running time of the algorithm by determining the connected components in parallel if possible. This is the sole reason why the parallel GPS implementation can attain a speedup greater than 1 as we will see in chapter 6.

Assigning new numbers according to the level structure

The last phase of the algorithm generates the numbering g by assigning numbers to the vertices level by level as in the RCM or SSP algorithm.

Special care has to be taken here however because the level structure is of a more general form. It is possible that there are several vertices in the first level not adjacent to each other or that vertices are adjacent to vertices in the same level only.

G10. [Possibly interchange u and v] If the degree of u is less than the degree of v, then interchange u and v and reverse the level structure from the previous phase by setting $N_i \leftarrow N_{k-i+1}, i = 1, \ldots, k$. This ensures that the numbering starts from the endpoint of lower degree.

G11. [Number the vertices in level 1]

1. Assign the number 1 to the vertex v.
2. Let w be the lowest numbered vertex of level N_1 which has unnumbered vertices in N_1 adjacent to it. Number these vertices adjacent to w, in order of increasing degree. Repeat this step until all vertices of N_1 adjacent to numbered vertices are themselves numbered.
3. If any unnumbered vertices remain in level N_1, then number the one of minimal degree and go to step G11-2.

G12. [Number the vertices in level $i = 2, \ldots, k$]

1. Let w be the lowest numbered vertex of level N_{i-1} that has unnumbered vertices of level N_i adjacent to it. Number the vertices of N_i adjacent to w in order of increasing degree. Repeat this step until all vertices of level N_i adjacent to vertices in level N_{i-1} are numbered.
2. Let w be the lowest numbered vertex of level N_i which has unnumbered vertices in N_i adjacent to it. Number these vertices adjacent to w, in order of increasing degree. Repeat this step until all vertices of N_i adjacent to numbered vertices are themselves numbered.
3. If any unnumbered vertices remain in level N_i, then number the one of minimal degree and go to step G12-2.

G13. [Possibly reverse the numbering] If one of the two following conditions hold then reverse the obtained numbering by setting $h(x) = n + 1 - g(x)$, otherwise set $h(x) = g(x)$:

- Step G10 interchanged the vertices u and v and the second element of the level pairs was chosen for C_1 in step G9.

- Step G10 did not interchange vertices u and v and the first element of the level pairs was chosen for C_1 in step G9.

The algorithm ends with the numbering h for the vertices of the graph $G^{(0)}$.

Appendix B contains a prototype for a (serial) GPS implementation. It can be used as a reference for the informal specification given above.

4.5.2 Application to the test matrix

The GPS algorithm applied to our test matrix, also selects vertex 9 as the starting vertex and generates the same level structure as in 4.3.

"Reverse" level structures are generated for the vertices 3, 12, 23 and 24 and unlike RCM this algorithm notices that the level structure rooted at 3 gives a smaller level width of 6. As all reverse level structures have the same depth, no iteration in the first phase of the algorithm is necessary. It ends with $v = 9$, $u = 3$, and the level structures rooted at them respectively. The associated level pairs are shown in figure 4.8.

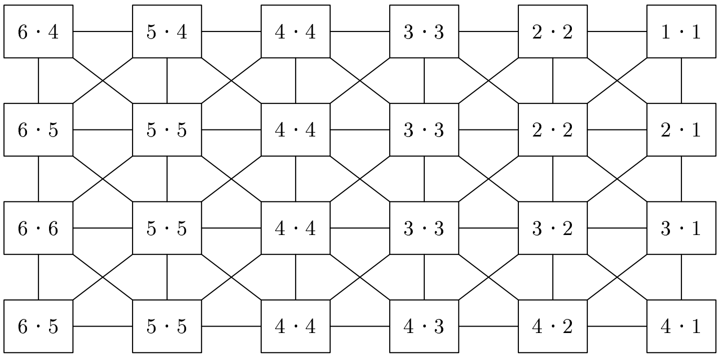

Figure 4.8: Associated level pairs for $v = 9$ and $u = 3$

Level pairs (i, i) are now assigned to level i. The three disjoint connected components remaining are shown in figure 4.9.

As the connected components are processed in order of decreasing cardinality, C_1 is considered first. Before the assignment, the widths of the levels are $(n_1, \ldots, n_6) = (1, 2, 3, 4, 3, 1)$. Assigning all vertices in C_1 according to the level structure rooted at v would yield the respective widths $(h_1, \ldots, h_6) = (1, 3, 5, 7, 3, 1)$, whereas assigning them according to the level structure rooted

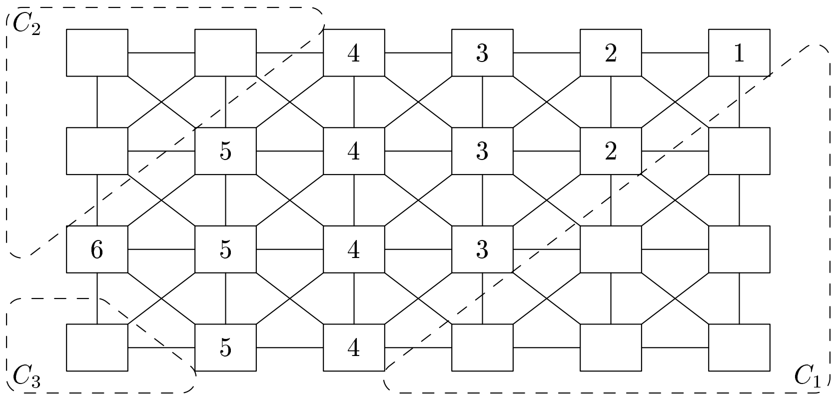

Figure 4.9: New level structure being built

at u would yield $(l_1, \ldots, l_6) = (4, 4, 4, 4, 3, 1)$. The maxima of these widths are $h_0 = 7$ and $l_0 = 4$ respectively, so the second elements in the level pairs are used to minimize the width of the resulting level assignments shown in figure 4.10.

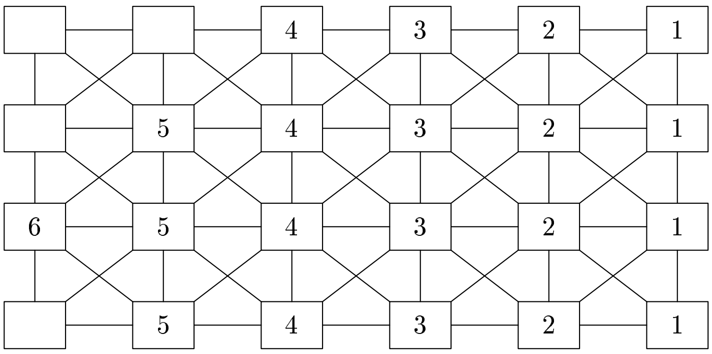

Figure 4.10: New level structure after assigning C_1

Repeating this process for C_2 and C_3 assigns them both according to the first element of the level pairs. The generation of the new level structure thus ends with the result shown in figure 4.11.

Note that this level structure has smaller width $w(\mathcal{L}) = 4$ than any possible rooted level structure for this graph.

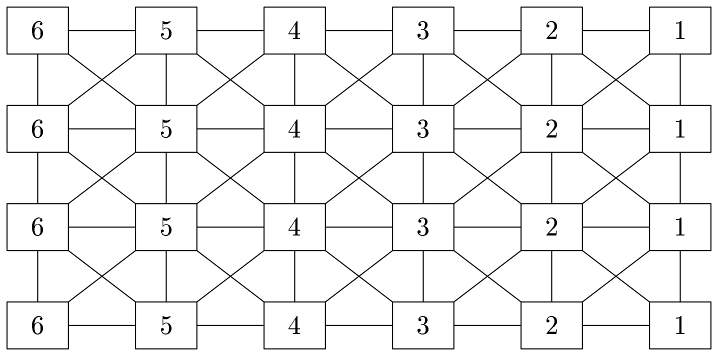

Figure 4.11: New level structure

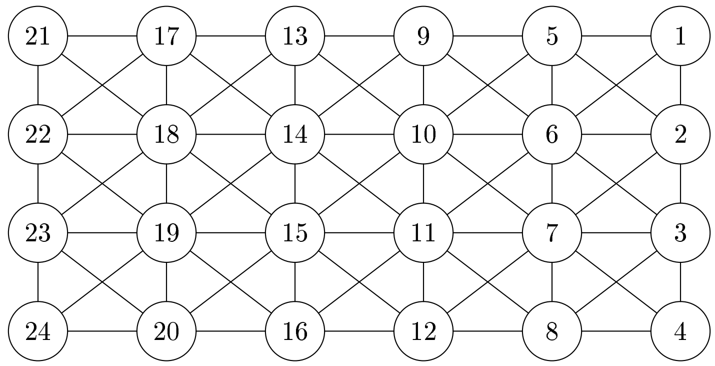

Figure 4.12: Resulting numbering

The numbering applied to the test matrix gives the result shown in figure 4.13 with a bandwidth of $\beta = 5$. It is well known that this cannot be reduced any further.

Note that every north-east south-west diagonal edge has this maximal width. This can be perceived as the homogeneity attained in the resulting matrix.

$$
\begin{bmatrix}
x & x & & & x & x \\
x & x & x & & x & x & x \\
& x & x & x & & x & x & x \\
& & x & x & & & x & x \\
x & x & & & x & x & & & x & x \\
x & x & x & & x & x & x & & x & x & x \\
& x & x & x & & x & x & x & & x & x & x \\
& & x & x & & & x & x & & & x & x \\
& & & & x & x & & & x & x & & & x & x \\
& & & & x & x & x & & x & x & x & & x & x & x \\
& & & & & x & x & x & & x & x & x & & x & x & x \\
& & & & & & x & x & & & x & x & & & x & x \\
& & & & & & & & x & x & & & x & x & & & x & x \\
& & & & & & & & x & x & x & & x & x & x & & x & x & x \\
& & & & & & & & & x & x & x & & x & x & x & & x & x & x \\
& & & & & & & & & & x & x & & & x & x & & & x & x \\
& & & & & & & & & & & & x & x & & & x & x & & & x & x \\
& & & & & & & & & & & & x & x & x & & x & x & x & & x & x & x \\
& & & & & & & & & & & & & x & x & x & & x & x & x & & x & x & x \\
& & & & & & & & & & & & & & x & x & & & x & x & & & x & x \\
& & & & & & & & & & & & & & & & x & x & & & x & x \\
& & & & & & & & & & & & & & & & x & x & x & & x & x & x \\
& & & & & & & & & & & & & & & & & x & x & x & & x & x & x \\
& & & & & & & & & & & & & & & & & & x & x & & & x & x
\end{bmatrix}
$$

Figure 4.13: Resulting matrix for the GPS algorithm

4.5.3 Discussion

The GPS algorithm is capable of constructing a level structure superior to the rooted level structures used by SSP without the explicit search employed by RCM. Two level structures rooted at end-points of a pseudo diameter are combined, yielding a more general level structure that can be smaller than each of them.

As we will see in chapter 6 this can not fully compensate the search of RCM but the results are comparative and attained in a fraction of the time.

4.6 Making the algorithms more applicable

All of the algorithms described in detail in this chapter generate a permutation for an irreducible matrix A. When applied to a reducible matrix they would of course stop after having labelled only one connected component of the associated graph.

One immediate solution is to apply the algorithms iteratively until all vertices have been numbered. This is not very efficient in the case of only the main-diagonal element being non-zero and should be avoided especially for the parallel implementations.

It is better to process these rows in a separate preprocessing stage and then apply the algorithms to the still unnumbered vertices iteratively until all vertices have been processed.

Chapter 5

The parallel implementations

He who can properly define and divide
is to be considered a god.

— Plato

In this chapter the implementations for distributed memory computers of the previously discussed algorithms RCM, SSP, and GPS, will be described. The employed data structures and the communication patterns are exhibited.

All modules were implemented from scratch with only slightly differing contextual demands as we shall see. This enables us in the next chapter to use the measurements for a comparison of the algorithms and not only of the implementation details.

The source code for all modules can be obtained via anonymous FTP from the server of the project group[1]. All three modules are well documented with plenty of embedded comments but are of course in some way dependent on the infrastructure of the LINSOL package, which they are a part of. The following sections should give enough explanations of the used parts of LINSOL to study the source code directly.

Note that the algorithm is a distinct entity from an implementation of it, as different implementations are possible. When we subsequently refer to RCM, SSP or GPS, it should be clear from the context if the algorithm or our implementation of it is meant.

[1]Anonymous FTP under `ftp://ftp-numerik.rz.uni-karlsruhe.de/pub/linsol`

5.1 Context

The algorithms are implemented as modules in the LINSOL [WHS95] package. When the ILU [HSW99a] preconditioner is selected, they are used to optimize the bandwidth of the preconditioning matrix which is then factorized (incompletely).

5.1.1 Implementation language

The LINSOL package is implemented in Fortran 90. Although even the inventor of Fortran predicted in his acceptance speech for the ACM award [Bac78] that modern functional languages with more expressive power (see the excellent [ASS96] for example) will supersede Fortran, implementations failed to deliver the efficiency of Fortran compilers. The functional approach is also much less susceptible to bugs and has some very appealing properties with respect to possible compiler optimizations.

Recent results of the SISAL [Can92] functional language signal a turning-point in the performance issue [Can91], but acceptance of functional languages is very low in the scientific community, despite the mentioned advantages. Fortran is still the most widely available implementation language for numerical applications, guaranteeing good portability.

5.1.2 The Portable MPI library

Distributed memory computers can be programmed using either the virtual shared memory or the message passing paradigm. Although easier for the programmer, the virtual shared memory programming style cannot be as efficient as using explicit message passing under the control of the program implementor.

Using the message passing style, one has to adapt the program to the message passing library available for the target system. As the most efficient system libraries use possible different standards, LINSOL uses its own abstraction layer that can be adapted to almost any system message passing library. This P_MPI library [HS94] shown in figure 5.1 encapsulates the application from the system library and thus improves its portability. The decision, which system library is to be used, is made at link time when one of the existing implementations of P_MPI is selected. Implementations for MPI, PVM, and several other vendor specific system libraries are available.

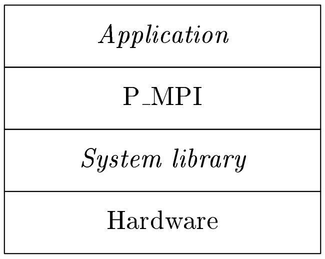

Figure 5.1: The P_MPI abstraction layer

P_MPI is itself highly portable. It is written entirely in Fortran 77 and only four point-to-point and two global communication primitives, listed in table 5.1, are provided. The send and receive primitives are asynchronous, enabling the application to carry out further computations during the delivery of the messages. In the optimal case this can hide the communication overhead entirely behind the computational effort when the underlying hardware provides autonomous communication units. If synchronous operations are needed, the statements can easily be grouped to achieve the desired effect. Group communication primitives, offered by MPI for example, are not available. The provided primitives are easily mapped to the target system as they are in the common denominator of all message passing libraries.

It should be noted that P_MPI does not do any marshalling of data types[2] for performance reasons and can thus not be used in heterogeneous computing networks. This is no serious restriction for our purposes however.

`MPSNDA`	Asynchronous send
`MPSNDW`	Synchronize on corresponding `MPSNDA`
`MPRCVA`	Asynchronous receive
`MPRCVW`	Synchronize on corresponding `MPRCVA`
`MPGATH`	Global reduction primitive
`MPSCAT`	Broadcast one to all

Table 5.1: The P_MPI communication primitives

[2]Data marshalling routines pack and unpack multi-byte data types into and from a standardized format according to the system they run on.

5.1.3 Target architectures and network topologies

LINSOL and thus the implementations described in this chapter, is targeted at and highly tuned for distributed memory vector computers. Wherever possible, vectorizable code is used delivering optimum performance on target systems having vector facilities.

No special properties of the underlying communication topology are taken into consideration. It is rather assumed that the communication from one processor to another is independent of the communication partners, i.e. there are no preferred communication paths. This assumption is justified by most modern distributed memory parallel computers employing mostly crossbar connections instead of special topologies as hyper-cubes for example.

5.1.4 External procedures

Only a few external procedures from the LINSOL core package are used by the bandwidth optimizers. They are summarized in table 5.2 without giving details about their calling conventions.

`LL4MIN`	Global integer minimum
`LL4INM`	Global integer maximum
`LL4ISM`	Global integer sum
`HEAPSORTIX`	Heap sort procedure
`SECONDS`	Get consumed CPU and wall clock time

Table 5.2: External procedures

The `LL4INM`, `LL4INM` and `LL4ISM` operations are built on top of P_MPI and compute the global minimum, maximum, and sum of integer vectors respectively. When using them for a global m-element vector of which each of the n processors stores m/n elements locally, they can either reduce the global vector over the processors yielding one result, or reduce the m/n elements separately yielding m/n results. They are symmetric in the sense that each processor has the result on their successful completion. There is no distinguished processor in sole posession of the result. These routines are often used to compute intermediate values that every processor needs for further processing. With the right inputs they can also be used as a form of broadcast.

`HEAPSORTIX` is a bit unusual in that it does not sort the data vector d but rather returns a permutation vector p such that $p(d)$ is sorted ascendingly.

5.1.5 Common communication patterns

One recurring communication pattern in the implementations is the ring-shift of distributed data. It is used at stages where the separate processors have to work on all of the global data in some way. The ring-shift conceptually treats the n processors as a linear arrangement with wrap-around or more precisely as a circle. Initially every processor P_i passes its relevant local data in a 'basket' b_i to one side and receives a basket from the other side. After n steps every basket b_i has passed every processor and all of them are in possession of their original local data again.

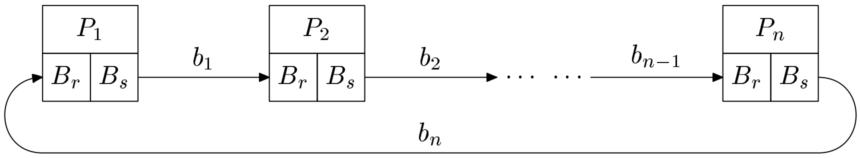

Figure 5.2: The ring-shift communication pattern

The first step of this communication pattern is shown in figure 5.2. In order to allow the simultaneous sending and receiving of data, each processor uses separate send and receive buffers, B_s and B_r respectively. While the communication is in progress, the data in the send buffer can safely be used for the computations if the passed data is static during the ring-shift. If the data in the baskets is to be changed, then the processing of the data cannot proceed asynchronously to the communication.

Formally stated, in step $j = 1, \ldots, n$ the processor with number i receives from the processor number $((i+n-2) \bmod n)+1$, sends to processor number $(i \bmod n)+1$, and processes the basket with the number $((i-j+n) \bmod n)+1$. At the end of step $j < n$, each processor copies its receive buffer into its send buffer and the ring-shift proceeds until $j = n$.

5.1.6 Distribution of the matrix

The matrix A is distributed among the processors in a row-wise fashion. Figure 5.3 shows an example for four processors. If the dimension of the matrix is not divisible by $nproc$, the last processors will have one less row than the others. The exact number of rows stored on processor p can be accessed in $lmatbk_p$ and has to be taken into account in all the implementations.

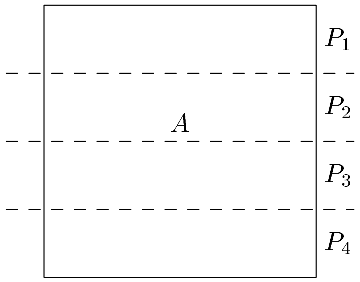

Figure 5.3: Distribution of the matrix A

The arrays created during the bandwidth optimization exhibit the same distribution over the processor with the exception of some tables used having the size of the maximum level width.

5.2 The Reverse Cuthill-McKee algorithm

Although the implementation of RCM is described first, it was in fact derived from the design of Schönauer [Sch99] for the SSP implementation. As the implementations share the basic data structures, they will be described here, although they apply to both algorithms.

Implementing RCM in a straightforward way according to the description in section 4.3 is not efficient for a distributed memory parallel computer. The numbering of the vertices can be accomplished simultaneously to the computation of the level structure. In fact the level structure is only built implicitly without storing it. All that is of interest is the resulting bandwidth of the numbering. As can be seen from the definition of level structures, the bandwidth of a numbering for a level structure can be calculated during the numbering phase as vertices from one level have adjacent vertices only in the same, or in adjacent levels.

This fact is exploited in the search over possible start vertices. Rather than computing all level structures completely, the search can skip to the next start vertex when the minimum bandwidth obtained during the search is exceeded during the computation of a level structure. This can save a considerable amount of time as the vertices dictating the bandwidth of a numbering are usually to be found near the centre of the graph.

As we will see, the implementation of RCM and SSP favour speed versus memory. The implementation of GPS is a bit more conservative in respect to the memory requirements.

In addition to generating the new numbering, the implementation also first builds the graph representation and on completion permutes the matrix A and the right hand side b to the new numbering.

5.2.1 Building the graph $G(A)$

In the initialization phase of the algorithm running on *nproc* processors, the linear arrays `ifirstrow` and `ilastrow` are computed, containing the first and last global row number that each processor holds in his local storage. They are initialized from the array `lmatbk` containing the number of locally stored rows for each processor. For the initialization of the two-dimensional arrays the global upper bound on the number of non-zero entries in a row is computed and stored as `maxdeg`. This has to be adjusted after the symmetrization phase which itself uses the preliminary upper bound for estimating the dimensions of the data structures used in this phase.

5.2.2 LINSOLs packed row format

Then a suitable representation of the graph $G(A)$ for the $n \times n$ matrix A is built from the internal LINSOL data structures. The bandwidth optimization is only performed when the ILU preconditioner is subsequently to be used. As the preconditioner will work on dense storage windows, the ten different supported storage schemes of LINSOL (for details see [HSW99b]) are converted to the *packed row* format suitable for this application. This conversion is carried out before the bandwidth optimization is performed, relieving it of the necessary logic to handle all the storage schemes.

For the packed row format, there is one *vector term* per row of the matrix A. The `info` field contains information on where to find the indices and the matrix coefficients in the linear arrays `index`, and `mat` respectively. The situation is shown schematically in figure 5.4.

Not shown in the picture are the remaining fields in the two dimensional array `info`. The meaning of the fields for a vector term of type packed row, relevant for the interface code, are summarized in table 5.3.

This LINSOL data is now converted to the internal representation shown in figure 5.5 that uses a two dimensional array `mindlist` to store the column

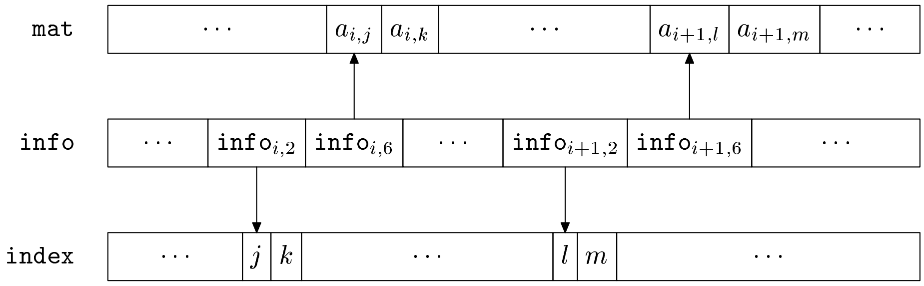

Figure 5.4: The packed row storage scheme

`info(i,1)`	Data type of vector term (9 for packed row)
`info(i,2)`	Pointer into field `index`
`info(i,3)`	Length of vector term
`info(i,5)`	Global row number
`info(i,6)`	Pointer into field `mat`

Table 5.3: Structure of the `info` field

indices of the non-zero entries in a row. These are effectively the numbers of the neighbour vertices. The linear field `mncon` stores the number of such entries, effectively being the degree of the vertex. The first dimension of these fields denoted in the diagram by l is the maximum number of rows stored on each processor, i.e. the maximum of `lmatbk`. Note that the row numbers do not have to be stored as they are implicit from the position in the fields.

Subsequently on we shall denote the rows as vertices and the index lists as adjacency lists, as that is our conceptual view from now on. This signals the shift of the problem into the graph theoretical domain.

Having this data available, it has to be symmetrized in the sense that if vertex i has vertex j in its `mindlist` field, then vertex j must have i in its `mindlist` field. This symmetrization is accomplished with a ring-shift. As this programming pattern recurs quite often in all implementations, it is shown in figure 5.6 as an example of the programming style.

First the sendbuffer is setup to contain the data to pass round the processors. In each step of the ring-shift, the send of the data to processor `totid`, and the receive from processor `fromtid`, are started asynchronously.

The main loop iterates over the data in the send buffer that contains data

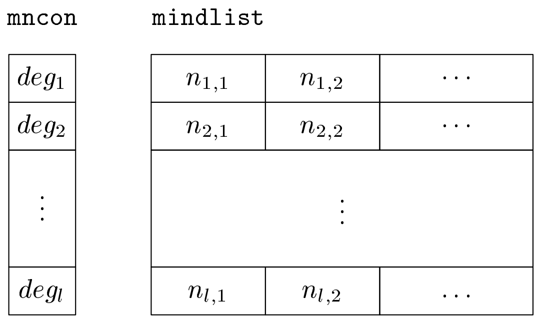

Figure 5.5: The RCM graph data structure

from processor number `ibasketprocess`. In this case the complete array `mindlist` is passed round so we know how much meaningful information is in the sendbuffer by consulting $lmatbk_{ibasketprocess}$.

Not shown in the example above is the actual processing. In the symmetrization phase the adjacency lists in the send buffer are checked against our local adjacency lists. If an entry is found denoting that a vertex v_i from processor `ibasketprocess` has a local vertex v_j in its adjacency list, the local information is updated if necessary.

When the data has been processed, the matching synchronizations for the send and receive operations are called. On their return, the communication is completed. The received data is copied to the sendbuffer and the next iteration begins.

After the ring-shift, our updated adjacency lists are sorted and the final value for `maxdeg` is computed. This ends the computation of the representation of $G(A)$. From now on we will use the notation V to denote all vertices of the graph as usual, and V_p denoting the vertices local to processor p. With this notation we now have `maxdeg` $= \max_{v \in V} deg(v) = d_{max}$.

5.2.3 Determining the search candidates

Next the possible candidates for the search of start vertices are determined. As described in section 4.3, this involves the computation of the median degree of the vertices. Usually the median is computed by sorting the values and taking the middle element of the sorted list. Using a standard sort algorithm based on comparison would have a lower bound for the runtime of

Initialize the sendbuffer

```
! Symmetrize the index list by ring-shift
do itact=1, nproc
  ibasketprocess=modulo(myproc-itact+nproc,nproc)+1
  ibasketrec=modulo(myproc-itact+nproc-1,p)+1
  if (nproc>1) then
    call MPSNDA(totid, nmsg+itact, (maxdeg+1)*l*iint, &
         sendbuf, mids, ierr)
    call MPRCVA(frtid, nmsg+itact, (maxdeg+1)*l*iint, &
         recbuf, midr, ierr)
  endif

  ! Process passing basket
  do i=1, lmatbk(ibasketprocess)
    do j=1, sendbuf(i,1)
      Process the sendbuffer
    enddo               ! j
  enddo                 ! i

  if (nproc>1) then
    call MPSNDW(totid, nmsg+itact, (maxdeg+1)*l*iint, &
         sendbuf, mids, ierr)
    call MPRCVW(frtid, nmsg+itact, (maxdeg+1)*l*iint, &
         recbuf, midr, ierr)
    sendbuf=recbuf
  endif

enddo                   ! itact
```

Figure 5.6: Ring-shift in the symmetrization phase

$O(n \log n)$. In our case we can do better however, because we have additional information about the values the median is computed for. As they are all integers in the interval $1 \leq deg(x) \leq d_{max}$, a linear array c is constructed on all processors $p = 1, \ldots, nproc$ such that $c_i = |\{x \in V_p \,|\, deg(x) = i\}|, i = 1, \ldots, d_{max}$. These local arrays are globally added up element-wise with the `LL4ISM` routine yielding the result array c. In effect we have now accumulated the number of vertices for each fixed degree i in the element c_i of the result vector. We can now determine the median by adding up c_i for i in increasing order, until this sum reaches half the number of total vertices. The median is the index i where this happens. As only one pass over the vertices is needed, this method, borrowing from the idea of *distribution* or more generally *radix sort* [Knu73], has the better lower bound of $O(n)$.

Now each processor determines how many local start vertices will be used to generate a tentative numbering. Processor i stores this number in $startcandnum_i$. Globalizing this vector $nproc$-element vector by element-wise summation provides all processors with the implicit synchronization information for the search phase. The first processor will initiate $startcandnum_1$ numbering generations, after which the control implicitly passes to the second processor and so on.

5.2.4 Generating a tentative numbering

Before we can start the generation of a numbering, the two dimensional fields used in the numbering phase are allocated using estimated upper bounds. The true upper bound is the width of a level which cannot be determined without actually building the level structure. As the RCM and SSP implementations are optimized for speed they use these estimations. This introduces the possibility that the storage is exceeded during the numbering phase. If this happens, the computation is aborted and the user is informed on how to change the parameters for the next run. No attempts to recover from these situations are performed. The tests show that with our estimates this does not happen very often.

During the numbering of the vertices, there is always one processor (the *active* processor) performing the numbering for complete levels until it has numbered at least as many vertices as it stores locally. Control is then transferred to the next logical processor which proceeds with the numbering. During this phase, the other processors (the *passive* processors) are waiting to provide data from their local storage to the active processor. This behaviour, being dependent on the role of each processor is shown in figure 5.7.

```
! Step IV:
! Number the nodes
iglobnewlast=1
if (myproc==1) then
  active=.true.
else
  active=.false.
endif
d1: do                        ! d1 - loops forever
  if ((myproc==1) .and. (active)) then
    ! This is a one-time setup to get things rolling
     One time setup of the data fields
    ! Now everything should be setup for the standard cycle

  endif                   ! (myproc==1).and.(active)


  if (active) then
    cycles: do            ! endless loop over all levels for one proc
      ! Here comes the code for the active processor
      ! He does the numbering and requests msgs from
      ! other procs
       Code for the active processor

  else                    ! active
    ! This is the code for the passive procs.
    ! We wait in an endless loop until we either get control
    ! or the calculation is finished and answer to requests
    ! in the meantime
     Code for the passive processors

  endif                   ! active
enddo d1
```

Figure 5.7: The differing roles during the numbering

5.2.5 The active processor

The one-time setup builds the data structures used during the numbering, for the first level. This means that after assigning 1 to the start vertex, its adjacent vertices are determined by communicating with the processor to which it is local, and store this information locally. This is a special case of the general setup performed for the next level which will be described in more detail later.

The assignment to new numbers is done in steps corresponding to levels in the rooted level structure. The active processor builds arrays with local copies of the relevant data. Figure 5.8 shows these arrays. In addition to arrays corresponding to entries in figure 5.5, the field `ilano` is used to store the old numbers of the previously numbered vertices in the order of their new numbers. In the figure u_1 was assigned the lowest number in the previous level, u_2 the next and so on. In `numb` are the numbers of the vertices associated with the arrays on the right. Vertices in `numb` are adjacent to vertices in `ilano`. The dashed lines are exemplary for such relations. They are represented in the data structures as u_1 appears in the adjacency lists of v_1 and v_2, i.e. in the first and second row of `indlist`, for this example.

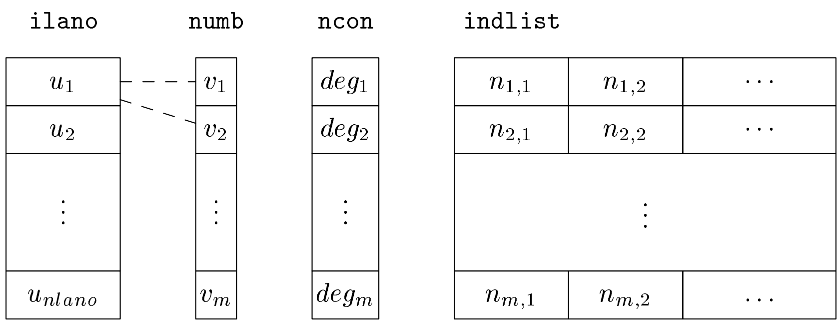

Figure 5.8: The data structures for numbering a level

Requesting the information for the next level

At the beginning of a step in the numbering phase, only the fields `ilano` and `numb` are available. The other information has to be gathered from the relevant processors.

For this to be efficient, the active processor groups the vertices in `numb` according to the processor on which they are stored. Because the information in `numb` is sorted and the vertices are distributed on the processors also in increasing order, it is sufficient to know how many vertices are local to each processor. This information is computed in `nunod` and used to request the required data from each processor in turn through message passing. The format of the messages is shown in figure 5.9 and the possible message types are given in table 5.4.

<table>
<tr><td colspan="7" align="center">Message</td></tr>
<tr><td colspan="3" align="center">Header</td><td colspan="4" align="center">Data</td></tr>
<tr><td>proc-id</td><td>type</td><td>len</td><td>$data_1$</td><td>$data_2$</td><td>$\cdots$</td><td>$data_n$</td></tr>
</table>

Figure 5.9: The format of a standard message

msg-type	Description
1	Request `indlist` data for *len* vertices
2	Request `mncon` data for *len* vertices
3	Transfer control to the receiver
4	Signal the completion of the numbering phase

Table 5.4: Message types

As the passive processors do not know in advance how much information will be requested from them, message sizes of the fixed size `msgsize` have to be used for the communication with the *len* field in the message header signalling how many entries are contained in the *data* section. If data is to be transferred that cannot fit into a single message, the transfer is accomplished in sections. Only the last section of these transfers has to be padded to the standard message size.

First, the active processor requests the degree information that is stored in `ncon`. When all this information is available, the adjacency lists are requested. As we have the degree information available, we can calculate in advance how many entries are sent back in response to our (sectioned) requests for *len* adjacency lists. Thus the response is a variable sized message without useless padding.

Assigning new numbers

When the data structures have been filled, new numbers can be assigned to the `m` candidates in `numb`. First we find all vertices adjacent to u_1, sort them in order of increasing degree and assign consecutive integers to them. Then we continue with neighbours of u_2 not already labelled and so on, until all `m` vertices have been assigned new numbers. The number assignments are stored in the field `iold`, such that $iold_i = j$ means the vertex with the old number j becomes vertex number i in the new numbering. With this storage each processor knows what lines it will hold after the permutation of the matrix.

Now that we have the new numbers for two consecutive levels, we can check whether the tentative numbering already exceeds the bandwidth of a previous trial. If this is the case, the active processor signals the end of the numbering phase and the next search iteration begins.

Signalling the end of the phase to the passive processors by sending an appropriate message also happens when the last processor P_{nproc} is active and all its new vertices have been computed.

When a complete numbering has been generated, having a bandwidth smaller than the current best, it is stored as the new current best. Then the next search iteration is started, until all search candidates have been processed. Note that, as was pointed out earlier, a new iteration can implicitly pass control to the next processor, which in turn initiates another tentative numbering for one of its local candidates.

Preparing the information for the next level

When still more levels are to be numbered, the fields `ilano` and `numb` have to be prepared for the next level. The old numbers of the newly assigned vertices are stored into `ilano` sorted by their new numbers. Of course `m` becomes `nlano` for the next round. The adjacency lists are stripped of vertices in the previous and the same level, duplicates are removed, and the result is the next level, i.e. the `numb` field.

Transferring control

The active processor p continues this process until it has generated at least $lmatbk_p$ vertices. At this stage control is transferred to processor $p + 1$. Because the numbering is done in level steps, usually more new numbers than needed by the active processor were assigned. These excess numbers have to be communicated to the processor becoming active, together with the fields setup for the next step.

A standard size message is constructed and sent to processor $p+1$ containing the number of vertices in the new `ilano` field, the number of entries in the new `numb` field, and the number of already assigned new vertices that the next active processor will hold. On completion of the communication a variable sized message with the contents of these two fields and the excess numbers is sent after which this processor becomes passive.

5.2.6 The passive processors

The passive processors wait for requests of the active processor as shown in the code fragment in figure 5.10. They block on an open receive until a message is received. The receive is open, i.e. the sender is not known, because passive processors cannot know which processor is active at a given time.

For the first two message types, the requested data is packaged and sent to the processor needing it, which is known from the *proc-id* field in the header of the message.

The third message type instructs the passive processor to receive `ilano`, `numb`, and the excess numbers in the `iold` fields. Having received the data, it becomes active and proceeds with assembling the additional needed information for the numbering, as described in section 5.2.5.

5.2.7 Permutation to the new numbering

Having completed the search over all possible start vertices, the best numbering is reversed in another ring-shift. This leaves us with the inverted permutation vector in `iold` that maps new vertex numbers to old vertex numbers. In order to permute the matrix to the new numbering, we also need the permutation vector mapping old to new vertex numbers. This mapping `inewsort` is also constructed in a ring-shift by passing around `iold`.

Now that we have this information available, the processors package the column indices of non-zero entries for local rows, the corresponding matrix coefficients, and the coefficients of the right hand side, and pass them around in two separate baskets. These baskets also use the padded two-dimensional representation, so the LINSOL data is fitted into this scheme. While processing the baskets, each processor copies the information for its new rows into its local storage.

```
! This is the code for the passive procs.
! We wait in an endless loop until we either get control
! or the calculation is finished and answer to requests
! in the meantime
do                    ! d4 (endless loop)
  ! Do an open receive
  call MPRCVA(-1, 0, msgsize*iint, msgbuf1, &
       midr, ierr)
  call MPRCVW(-1, 0, msgsize*iint, msgbuf1, &
       midr, ierr)

  sendproc=msgbuf1(1)
  key=msgbuf1(2)
  number=msgbuf1(3)

  ! According to'key' we have to fulfill different requests
  msgtype: select case (key)

  case (1)
     Send list of connections for the requested nodes

  case (2)
     Send number of connections for the requested nodes

  case (3)
     Gain control and receive the level data structures

  case (4)
    ! We are finished
    exit d1

  endselect msgtype
enddo                 ! d4
```

Figure 5.10: Code for the passive processors

The information for the rows are now permuted according to the best numbering. Only the column indices are still in accordance with the old numbering. To correct this situation, the permutation vector `inewsort` is passed around in a ring-shift and each processor uses it to map its column index lists to the new numbering.

Now each processor has the new data in the internal representation. All that remains is to copy this into the appropriate LINSOL arrays as described in section 5.2.2. Apart from the intended side-effect on the stored matrix, the procedure returns the permutation vector that is needed to reorder the subsequently computed solution.

Note that the implementation cannot handle reducible matrices A at the moment, with the exception of the special case where A contains rows j with only $a_{jj} \neq 0$. These rows, corresponding to isolated vertices in the graph G, are taken care of in a preprocessing step as such matrices can arise in the FDEM program package. Fixing the general case is also possible without changing the current framework.

The module for the RCM optimization totals at 1350 lines of Fortran source code including the plenty comments.

5.3 The Schönauer single pass algorithm

As was noted in chapter 4, SSP can be seen as a variation of RCM without the search over start vertices. The discussion of RCM thus applies to the SSP module with the differences being the omitted search and the different strategy when selecting candidates for the new numbering.

5.3.1 Finding a start vertex

SSP uses only one start vertex which is determined on the distinguished processor P_1 that will also be the first active processor. Every other processor sends information about its vertex with lowest degree to P_1, which reduces it to the global minimum.

5.3.2 Assigning new numbers

When the fields `ilano`, `numb`, and `indlist` have been setup as described in section 5.2.5 the numbering can be done. Where RCM only orders the candidates in order of increasing degree, SSP uses a different strategy to number the vertices in `numb`.

As described in chapter 4, the vertices in `ilano` are processed in order of their new numbering. Let v_1, v_2, and v_3 be vertices from `ilano` that were assigned consecutive numbers previously. When considering v_1, first unnumbered vertices from `numb` being adjacent to v_1 are copied into `stack1` and are marked as used. The separate field `used` keeps track of what vertices from `numb` have been used.

Then the second criterion is applied to the entries in `stack1`, namely the vertices being adjacent to vertex v_2 are copied to `stack2`, and marked in the associated field `used1`. The third criterion similarly marks and copies all vertices from `stack2` to `stack3` being adjacent to v_3. An example is shown in figure 5.11. The vertices that have been marked in the respective `used` fields are shown parenthesized.

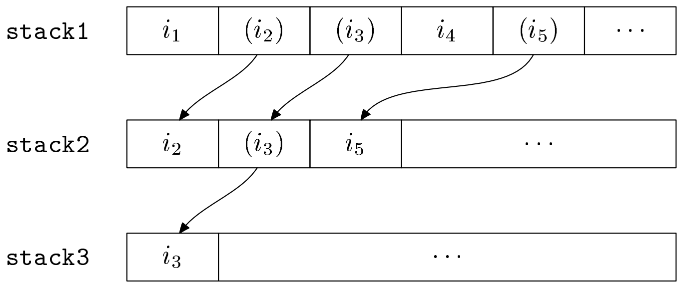

Figure 5.11: The stacks used to select candidates

New numbers are assigned to entries in `stack3` first, in order of increasing degree if there is more than one. The unmarked vertices in `stack2` are processed next, followed by the unmarked entries in `stack1`.

The numbering continues with neighbours of v_2 next, until all vertices in `ilano` are processed.

The SSP module totals at 1290 lines of source code, being comparable to the implementation of RCM.

5.4 The GPS algorithm

In comparison to the RCM and SSP module, the implementation of GPS is more conservative in respect to memory usage. This includes a different representation of the graph shown in figure 5.12. As can be seen, the adjacency lists are stored dense in comparison to the two dimensional fields in the other two implementations. The `graph` and `vertix` fields are parameters computed by a preprocessing procedure `PREPRO` external to the GPS module, which will not be discussed here.

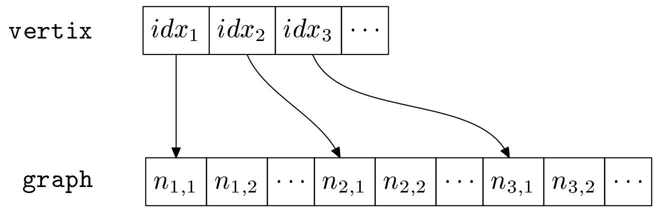

Figure 5.12: The graph structure used by GPS

5.4.1 Preprocessing

LINSOL allows an optional elimination of Dirichlet boundary conditions in a preprocessing step. This will decouple these pseudo-unknowns and produce isolated vertices in the corresponding graph. These vertices are numbered first in a preprocessing step.

For this, each processor p determines the number of isolated vertices local to it, and stores them in $cands_p$. This vector is summed globally with `LL4ISM` so that all processors know how many isolated vertices are on each processor. With this information the numbers starting with 1 can be assigned to them in the local `perm` vector. This vector effectively masks all numbered vertices from subsequent processing.

5.4.2 Determining a pseudo-diameter

The search for a pseudo-diameter begins with a vertex of minimum degree. Ties are broken in favour of the lowest numbered vertex. As the GPS implementation is quite symmetric in the sense that there is no distinction between active and passive processors on a high level, all processors need to know who possesses the start vertex. To do this efficiently, the procedure `LL4INM` is used. Each processor determines its candidate vertex numbered c_p with degree $deg(c_p)$. Computing the global minimum $c = \min_p\{deg(c_p) \times nproc + (p-1)\}$ yields the necessary result in one step, namely processor $((c \bmod nproc) + 1)$ wins.

5.4.3 Building the forward level structure

As it is now known which processor owns the start vertex v, the first level structure can be constructed. The `SETLEVEL` procedure accomplishes this. As this routine also determines the connected components needed later in the process, the detailed discussion will be deferred until then. For now it is enough to know that the local parts of the level structure are available after `SETLEVEL` on each processor in the `levelv` field. The width and depth of the level structure were also computed during its determination.

5.4.4 Building reverse level structures

In step G3, all reverse level structures for the vertices in the last level $S = L_{e(v)}$ have to be computed in order of increasing degree of the vertices. In our case we can circumvent the sorting across the processors, as we again have additional information about the values that have to be sorted. It is known, that they are all in the interval $1 \leq deg(s) \leq d_{max}$. Each processor p determines how many vertices $s \in S$ are local to it and stores this information indexed by $deg(s)$ into the two-dimensional array `cands`. An example is shown in figure 5.13.

$deg(s)$	1	2	3	4	
`cands`	0	0	1	0	P_1
	0	1	0	0	P_2
	0	0	2	0	P_3
	0	0	1	1	P_4

Figure 5.13: Candidates for reverse level structures

In the example, processor P_1 possesses one candidate with degree 3, P_2 one candidate with degree 2, and so on. This data structure is effectively broadcasted with `LL4ISM`. Now that each processor has the necessary information, the generation of the level structures can begin. By iterating over `cands` as is shown in figure 5.14, the synchronization is achieved implicitly. Note that there is only one processor passing a non-negative argument `cand` to `SETLEVEL`. This is the vertex at which the level structure `levels` will be rooted.

```
! (Step G4)
do deg=1, maxdeg
  do p=1,nproc
    cand=0
    per_proc: do i=1,cands(p,deg)
      if (p==myproc) then
         Determine next local candidate
      endif
      levels=0
      call setlevel(graph, vertix, levels, cand, 1, &
                    swidth, sdepth)
       Evaluate reverse level structure levels
    enddo per_proc
  enddo
enddo
```

Figure 5.14: Building the reverse level structures

If a level structure deeper than L_v is found, an iteration occurs as described in chapter 4. The phase ends with the forward level structure `levelv` and the reverse level structure `levelu` of minimal width.

5.4.5 Building the new level structure

In the next phase the new level structure `levell` is assembled from the results of the previous phase. At this stage we also know, by inspecting the generated level structures, whether the graph was connected. Vertices that have not been assigned to a level are now masked out so that they are ignored by the following processing. They will be handled during the next iteration of the whole algorithm.

Finding the connected components

As noted in the description of the GPS algorithm, this step is the only candidate for an effective parallelization. The connected components are determined similar to level structures. A start vertex is marked first and then iteratively all vertices reachable from it that were previously unassigned. For this each processor determines whether a local unassigned vertex exists. This information is globalized and is used to decide whether more connected components exist. The relevant code fragment is shown in figure 5.15.

```
! Find all connected components
! (Step G8)
find_ccs: do c=myproc,l*nproc,nproc
  pos=0
   Find local unassigned vertex in levell

  ! Are there any more components to process?
  minmaxfield(1)=pos
  call LL4INM(1, 1, 1, minmaxfield, minmaxout, minmaxwork, &
       myproc, nproc, tids, nmsg)
  if (minmaxout(1) == 0) exit find_ccs

  ! Determine connected component with setlevel
  call setlevel(graph, vertix, levell, pos, -c, swidth, sdepth)
enddo find_ccs
```

Figure 5.15: Finding the connected components

Every processor having an unassigned vertex uses this as a start vertex for the marking to be done in the `SETLEVEL` procedure. The initial marking is done with a globally unique negative integer. We will now see how the generation of level structures and the determination of connected components is accomplished.

The `SETLEVEL` procedure

The internal `SETLEVEL` procedure is at the core of the implementation. Its task is to mark all vertices reachable from a start vertex. If this marking is done with successive integers, the rooted level structure is determined, and if it is done with some constant, connected components are found.

`SETLEVEL` is called with an integer `startlev`, that is to be assigned to the start vertex. If it is positive, consecutive integers will be assigned and if negative, it is treated as a constant to mark connected components.

The main loop of the procedure propagates the markings in level steps, until no changes are noticed in one iteration. Two fields are used for this purpose. The level structure before an iteration is kept in `lold`. After assigning the next level, the element-wise difference of the new structure `level` and the old level `lold` is computed in `ldiff`. Figure 5.16 shows an example after assigning vertices in level 4.

`level`	$\cdots$	4	2	2	4	1	$\cdots$
`lold`	$\cdots$	0	2	2	0	1	$\cdots$
`ldiff`	$\cdots$	4	0	0	4	0	$\cdots$

Figure 5.16: Determining the changes in one iteration

The entries in `ldiff` are the active vertices for the next level, i.e. only neighbours of vertices having non-zero entries in it have to be checked. Summing `ldiff` element-wise over all processors also provides two intermediate results. If this sum is zero, no further markings were done and `SETLEVEL` is done, otherwise when assigning levels, the sum divided by the number of the last assigned level yields its width.

The marking of the vertices is done in a ring-shift where the current `level` fields are passed around. Each processor consults this information in conjunction with its local adjacency information and `ldiff` to fill the blanks in `level`, thus effectively propagating the markings to the neighbours of the active vertices.

This is all that is needed to generate level structures. When determining connected components, conflicts can arise due to parallel attempts, that have to be dealt with.

Possible conflicts when determining connected components and how to solve them

For the generation of rooted level structures, there is only one start vertex initiating the assignment to levels. The level numbers spread like a wavefront to adjacent vertices. When determining connected components however, each processor possibly starts with an as yet unassigned start vertex w_p, resulting in several wavefronts spreading over the graph. Difficulties now occur when two such wavefronts meet. One of them carries the negative index i and the other the negative index j. When this happens, we have to ensure that after the assignment all of the vertices are marked with either i or j, and not with a mixture of both.

The subproblem that has to be solved, is to determine whether the separate start vertices can be reached from each other through the still unassigned vertices. For this we use another graph G_{reach} that represents these reachability relations among the start vertices. As each processor can have one start vertex, the graph has $nproc$ vertices and is stored using an adjacency matrix representation

$$
\begin{aligned}
A_{reach} &= (\alpha_{ij}), \quad i,j = 1, \ldots nproc, \\
\alpha_{ij} &= \begin{cases} \neq 0, & \text{if } w_i \text{ is reachable from } w_j \text{ through vertices} \\ & \text{yet unassigned to the new level structure;} \\ 0 & \text{otherwise.} \end{cases}
\end{aligned} \tag{5.1}
$$

Whenever a processor notices a conflict between wavefronts for the indices i and j that were initiated by processors P_{-i} and P_{-j} (the relation between the indices and the processors is in fact more general to allow for more iterations), this is marked by setting $\alpha_{-i,-j}$ and $\alpha_{-j,-i}$ to 1 in the initially empty `adjmat` array.

When the marking has been completed, these local graphs are combined by projecting all their edges onto a new graph, yielding the true reachability graph. This is done simply by summing the adjacency matrices globally. Now we have to solve a similar problem of smaller size, namely determining the connected components in the small reachability graph. This is done by the recursive subroutine `MARK` that uses simple back-tracking to mark all reachable vertices in the graph represented by `adjmat`. The marking is done in the `conflict` vector. When all connected components of the reachablitiy graph have been determined, the vector `conflict` is used in a post-processing stage to map the generated negative indices to a marking that correctly represents the connected components, found in this iteration, of the graph to be numbered. If unassigned vertices still exist, this process is iterated.

Assigning the connected components

When the connected components are marked in the `levell` field, each processor counts how many elements it possesses in each component and stores this information in the `ccard` field. Adding this information globally we can now iterate over the connected components in order of decreasing cardinality. It should be noted here that no ordering is prescribed by the algorithm for processing components of the same cardinality although this might result in small differences in the resulting bandwidths. This explains minor differences in the results of GPS when run on differing numbers of processors as the order in which the components are assigned then can also differ.

The vectors `ni`, `hi`, and `li` are used during the assignment, holding the current level widths and the widths of the tentative assignments, as described in the specification of the algorithm. When each processor successfully computed these vectors for one component, their globalized results decide which element from the associated level pairs is used for the assignment of the vertices.

5.4.6 Assigning new numbers

Before the numbering starts, the condition in step G10 is checked by broadcasting the relevant information through a global sum operation. If the condition holds, then every processor reverses its level numbers in the `levell` field.

The numbering is done by iterating over the levels of the general level structure generated in the previous phase. First the lowest available number is assigned to vertex v. This can be different from 1 if isolated vertices were processed in the preprocessing or if the graph is decomposable and several iterations of the whole algorithm are necessary.

The two variables `plevel` and `clevel` respectively denote the number of the previous numbered level and of the level under consideration. The special treatment of the first level in the description of the algorithm can be handled by starting with both of these variables as 1 and incrementing them subsequently, omitting the first increment for level `plevel`. The problem then is to find the lowest numbered vertex in level `plevel` that has unnumbered neighbours in level `clevel`. The number of vertices to be numbered in this level is stored in `remaining` and updated constantly during the numbering.

To efficiently find the candidate vertices, the information about vertices in level `clevel` are passed around in a ring-shift. Each processor builds a local table `clevnodes` from the passing baskets. In the table, the old numbers of unnumbered vertices and their degrees are stored. The table is then sorted in order of increasing degree, as this is needed during the assignment of new numbers.

When every processor has built the `clevnodes` table, each one can search a candidate from among his local vertices, satisfying the condition that it has unnumbered vertices in level `clevel`. The processor in possession of the vertex with the global minimum over the new numbers of these candidates assigns the next numbers. It assigns them in order of increasing degree with the help of the degree information in the `clevnodes` table. The new number information is then broadcasted to all other processors and used by them to update their local `clevnodes` table. The search for a new candidate is then repeated.

When no more candidates are found, the variable `remaining` is checked. If it is zero, the numbering of level `clevel` is complete and the iteration over the levels is continued. If not, each processor offers its candidate with the lowest old number in a global minimum and the winner assigns the next number. The counter in `remaining` is updated and the search for candidates with unnumbered neighbours in level `clevel` is iterated.

When new numbers have been assigned to all levels, the numbering is inverted if one of the conditions described in step G13 holds. If vertices were found not to be part of the level structures during the determination of the forward and reverse level structures, a warning is output to the user informing him that the matrix is reducible, and the whole algorithm is iterated until all vertices have been numbered.

5.4.7 Permuting the matrix

The GPS module does not contain code for permuting the matrix. It simply returns the generated numbering in the form of a permutation vector. The permutation of A and the right-hand side is handled completely external to the module by LINSOL itself.

The implementation of GPS consists of 1023 lines of source code, but lacks the graph generation and the matrix permutation stages in comparison to the RCM and SSP modules.

Chapter 6

Results

The purpose of computing is insight, not numbers!

— R. W. Hamming

Results of applying the implementations of RCM, SSP, and GPS discussed in detail in the previous chapter are given in this section. The test cases are matrices often used for comparisons of sparse matrix packages and matrices generated by the FDEM program package.

6.1 General matrices

The matrices in this section are taken from a repository of test matrices on the Internet[1]. The measurements in this and the next section were done on the IBM SP2 system at the computing centre of the University of Karlsruhe. The performance of the Thin P2SC (120MHz) nodes used for the tests are discussed in detail in Schönauer [Sch00]. For the small problems only one processor was used for the measurements.

Here and below, two times are given for the implementations. The first denotes the net execution time of building the graph representation, renumbering it, and permuting the matrix while the second figure given in parentheses is the execution time of only the graph renumbering phase. Building the graph and permuting the matrix is implemented differently for technical reasons. Namely the implementation shared by the RCM and SSP modules is tuned for speed but needs some parameters which if set incorrectly can

[1] `http://math.nist.gov/MatrixMarket/`

abort the process due to insufficient memory. The implementation for GPS needs no such parameters but turns out to be much slower thus influencing the overall performance significantly. As this routine `PREPRO` is subject to further improvement, the times are broken down allowing a fair comparison.

Oil reservoir and computer system simulation

Table 6.1 summarizes the results of the implementations for two very small examples. The matrices `orsreg1` from the `OILGEN` collection stems from oil reservoir simulations and has 2205 unknowns. Simulation of computer systems led to the `gre512` matrix from the `GRENOBLE` collection with 2205 unknowns. The given execution times are in seconds.

Ordering	$\beta(A_{opt})$	Time	Ordering	$\beta(A_{opt})$	Time
Unordered	441	N.A.	Unordered	64	N.A.
RCM	101	10.76 (10.73)	RCM	36	0.30 (0.29)
SSP	105	0.17 (0.13)	SSP	37	0.03 (0.01)
GPS	101	2.05 (1.05)	GPS	35	0.12 (0.07)

Table 6.1: Results for the `orsreg1` and `gre512` matrices

Electronic circuit design

The study of electronic circuit design led to the `memplus` matrix from the `HAMM` collection. As this matrix with 17758 unknowns is somewhat larger than the others 4 processors were used for the test runs producing the results given in table 6.2.

Ordering	$\beta(A_{opt})$	Time
Unordered	17751	N.A.
RCM	N.A.	> 14400
SSP	8984	43.74 (39.26)
GPS	5467	128.53 (90.63)

Table 6.2: Results for the `memplus` matrix

As noted in the table this problem is already too large for RCM to handle. No result was produced in 4 hours of computing time. This shows the unacceptable performance of the algorithm for non-trivial problems which is also obvious from subsequent examples.

The original matrix and the result produced by SSP are shown in figure 6.1. GPS optimizes the matrix as shown in figure 6.2.

6.1.1 Complete factorization results

Finite Element Modelling

Two medium sized problems have been solved for the comparison of the iterative solver in LINSOL and a complete LU factorization on the bandwidth optimized matrices. The first example `fidapm11` from the `FIDAP` collection stems from Finite Element Modelling and has 22294 unknowns. The results of the tests shown in table 6.3 were computed on 4 processors also. The unordered case was solved with an error bound for the iterative solver of 10^{-6}.

Ordering	$\beta(A_{opt})$	Opt. time	Fact. time	Fill-in	Total time
Unordered	6505	N.A.	N.A.	N.A.	589.25
RCM	N.A.	> 14400	N.A.	N.A.	N.A.
SSP	1307	59.01 (57.82)	149.85	42.6	209.83
GPS	1065	394.44 (276.00)	142.99	46.2	539.10

Table 6.3: Iterative solution and complete LU factorization for `fidapm11`

Again, RCM is unable to produce a result in under 4 hours. Both SSP and GPS yield good results allowing the complete LU factorization to complete in the available storage with faster execution times than the iterative solver iterating with an error bound of 10^{-6}. The unordered matrix and the optimization of SSP are shown in figure 6.3. The matrix optimized by GPS is shown in figure 6.4.

Crystal Growth Simulation

The matrix `cry10000` from the `CRYSTAL` collection arises in the Crystal Growth Simulation with a Diffusion Model. The matrix has 10000 unknowns and the results shown in table 6.4 were obtained on 4 processors. Again all times are in seconds.

The matrix shown in figure 6.5 together with the SSP optimization is a worst case for the iterative solvers. They fail to converge and thus force the application of another technique. The full LU factorization can be carried

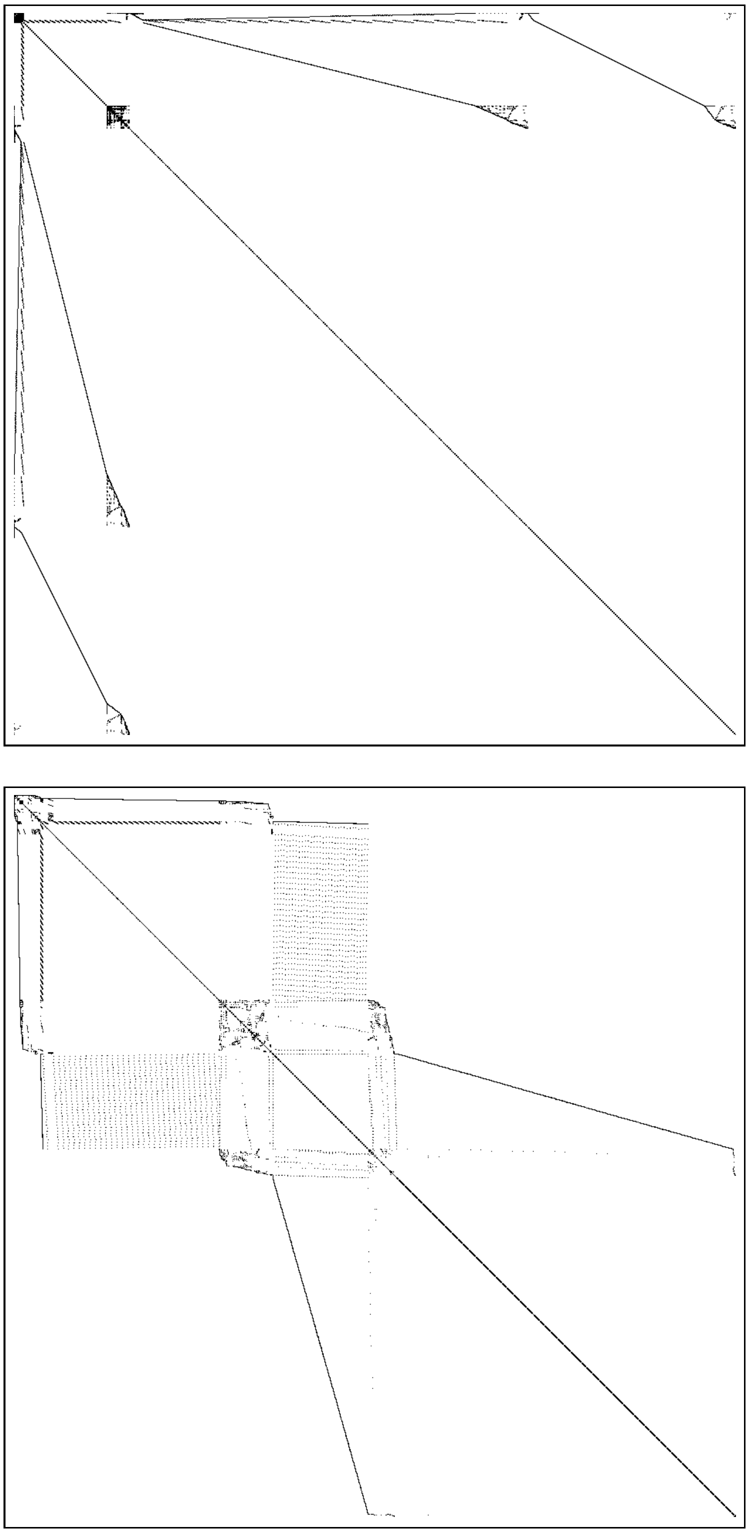

Figure 6.1: Unordered matrix `memplus` and result of SSP

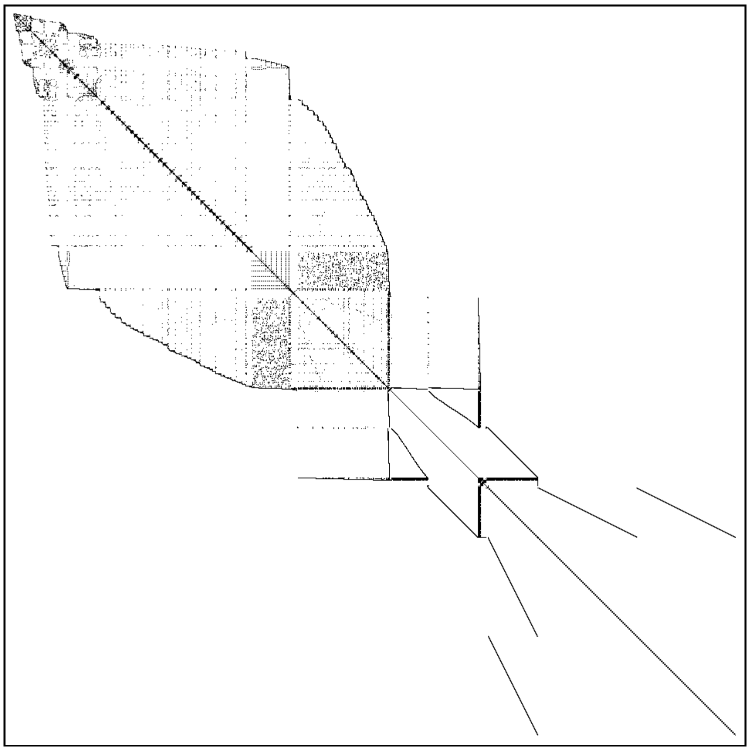

Figure 6.2: Result of GPS for matrix `memplus`

Ordering	$\beta(A_{opt})$	Opt. time	Fact. time	Fill-in	Total time
Unordered	9900	N.A.	N.A.	N.A.	N.A.
RCM	N.A.	> 14400	N.A.	N.A.	N.A.
SSP	102	0.62 (0.53)	12.55	27.5	13.62
GPS	102	20.33 (13.23)	8.53	27.3	29.01

Table 6.4: Complete LU factorization results for `cry10000`

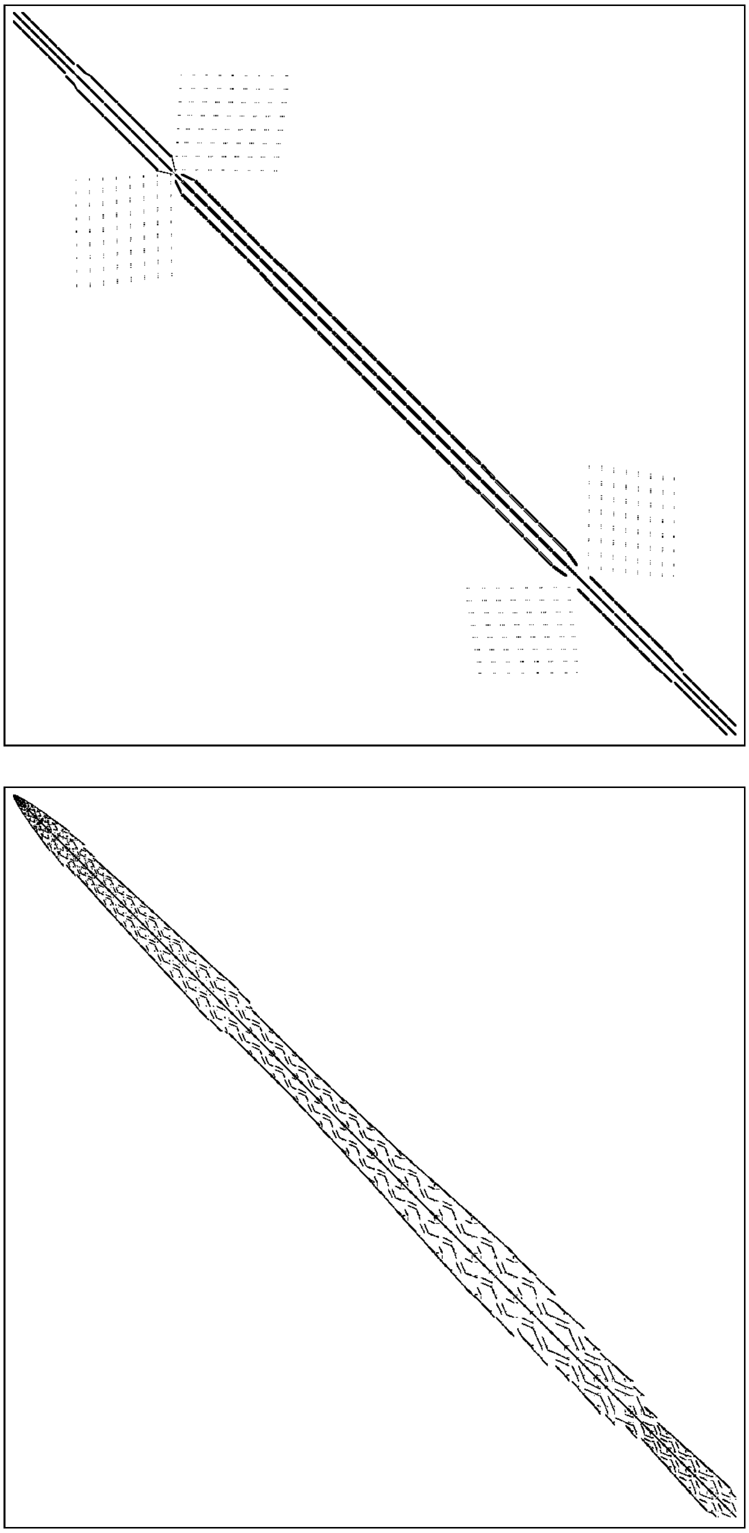

Figure 6.3: Unordered matrix `fidapm11` and result of SSP

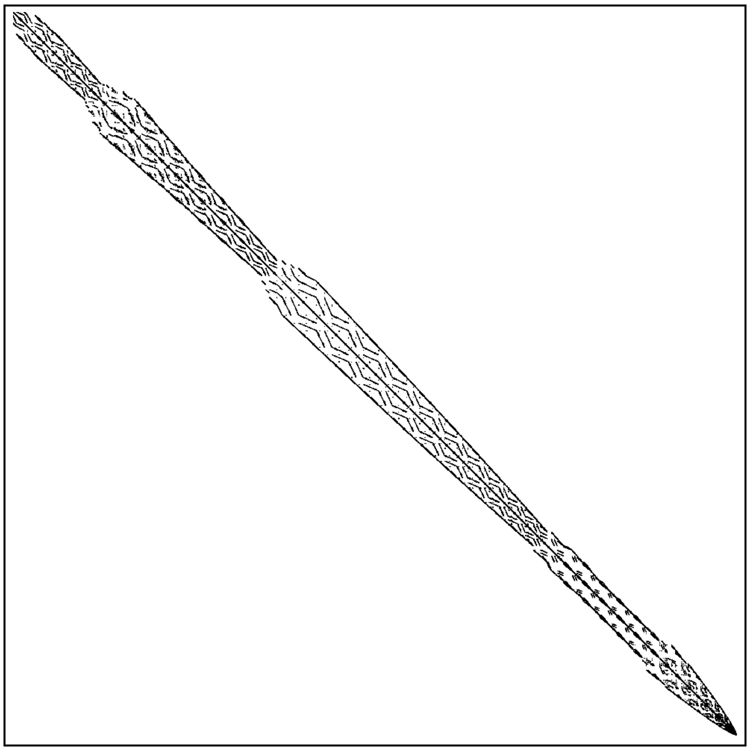

Figure 6.4: Result of GPS for matrix `fidapm11`

out efficiently in the narrow band in the matrices obtained by SSP and GPS. The latter is shown in figure 6.6.

6.2 Two dimensional PDE discretizations

The matrix c2253 is the result of the discretization of a system of 3 PDEs with the FDEM package. The underlying two-dimensional mesh is circular with 751 grid points [SA00b]. For three equations this results in 2253 unknowns. Figures 6.7 and 6.8 show the patterns of the matrix as output from the discretization step and as output by the three implementations. Table 6.5 shows the results on one processor for this example.

Ordering	Bandwidth	Time in seconds
Unordered	2154	N.A.
RCM	106	63.46 (63.41)
SSP	115	0.28 (0.24)
GPS	109	3.23 (1.50)

Table 6.5: Results for the matrix c2253

The matrix c3669 stems from the same problem as c2253. In contrast to the latter, two local refinements on the mesh introduced additional 8754 grid points resulting in a total of 11007 unknowns. The unordered matrix is shown at the top of figure 6.9. The dark grey area in the upper left hand corner corresponds to the mesh before the refinement steps and can be seen to resemble the pattern of c2253. The light grey area corresponds to the unknowns introduced in the first, and the white area to the unknowns introduced in the second refinement step. It should be noted that this matrix was not pre-processed by FDEM as will be described in the next section.

Ordering	Bandwidth	Time in seconds
Unordered	10706	N.A.
RCM	383	11687.46 (11686.90)
SSP	443	6.71 (6.32)
GPS	411	85.44 (19.55)

Table 6.6: Results for the matrix c3669

As can be seen from table 6.6, this matrix is an example of a worst case for the RCM algorithm. The process used to discretize the problem produced

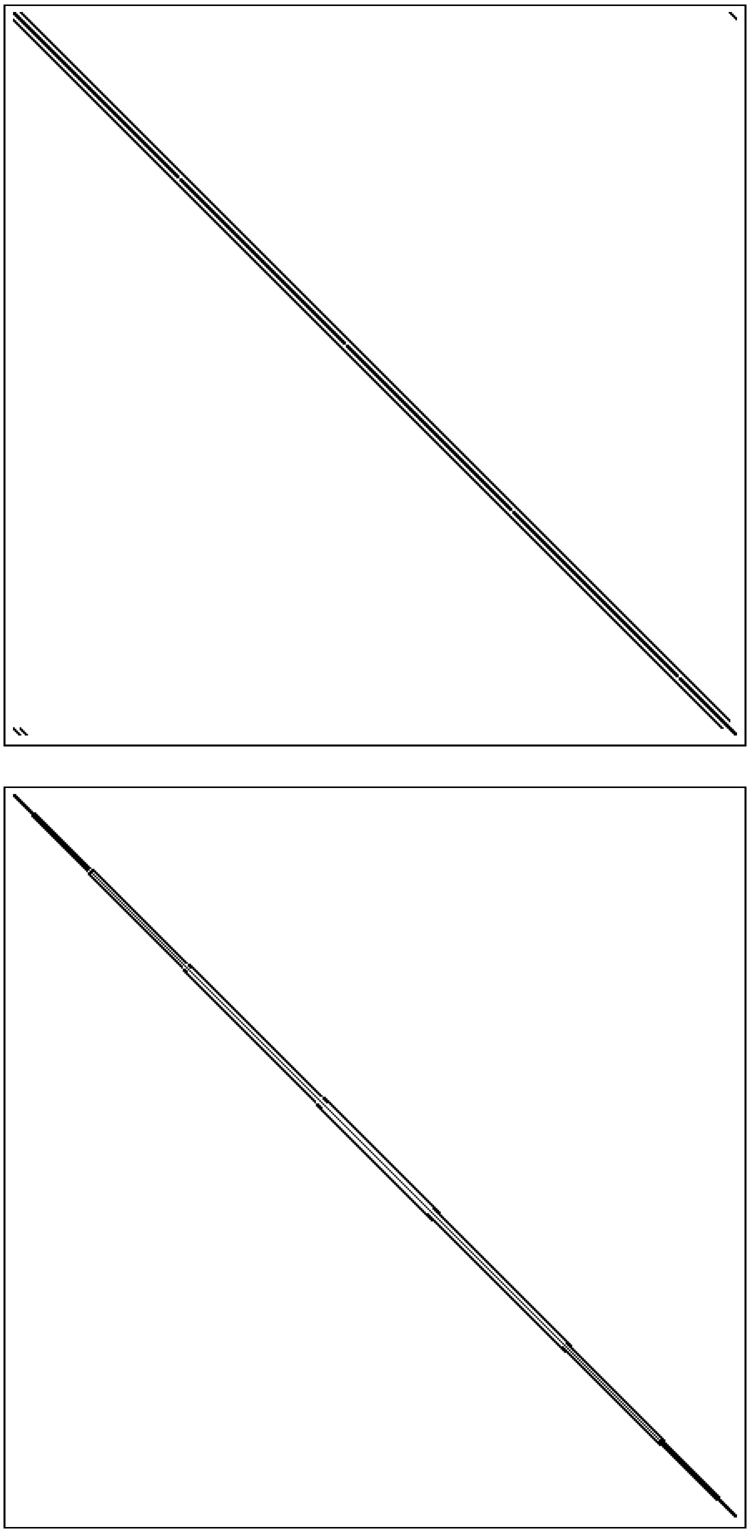

Figure 6.5: Unordered matrix `cry10000` and result of SSP

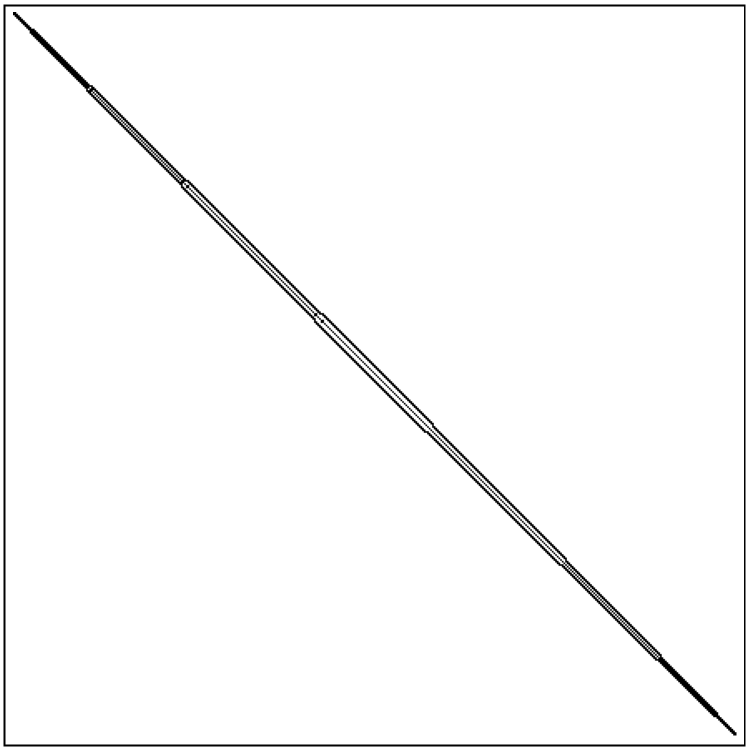

Figure 6.6: Result of GPS for matrix `cry10000`

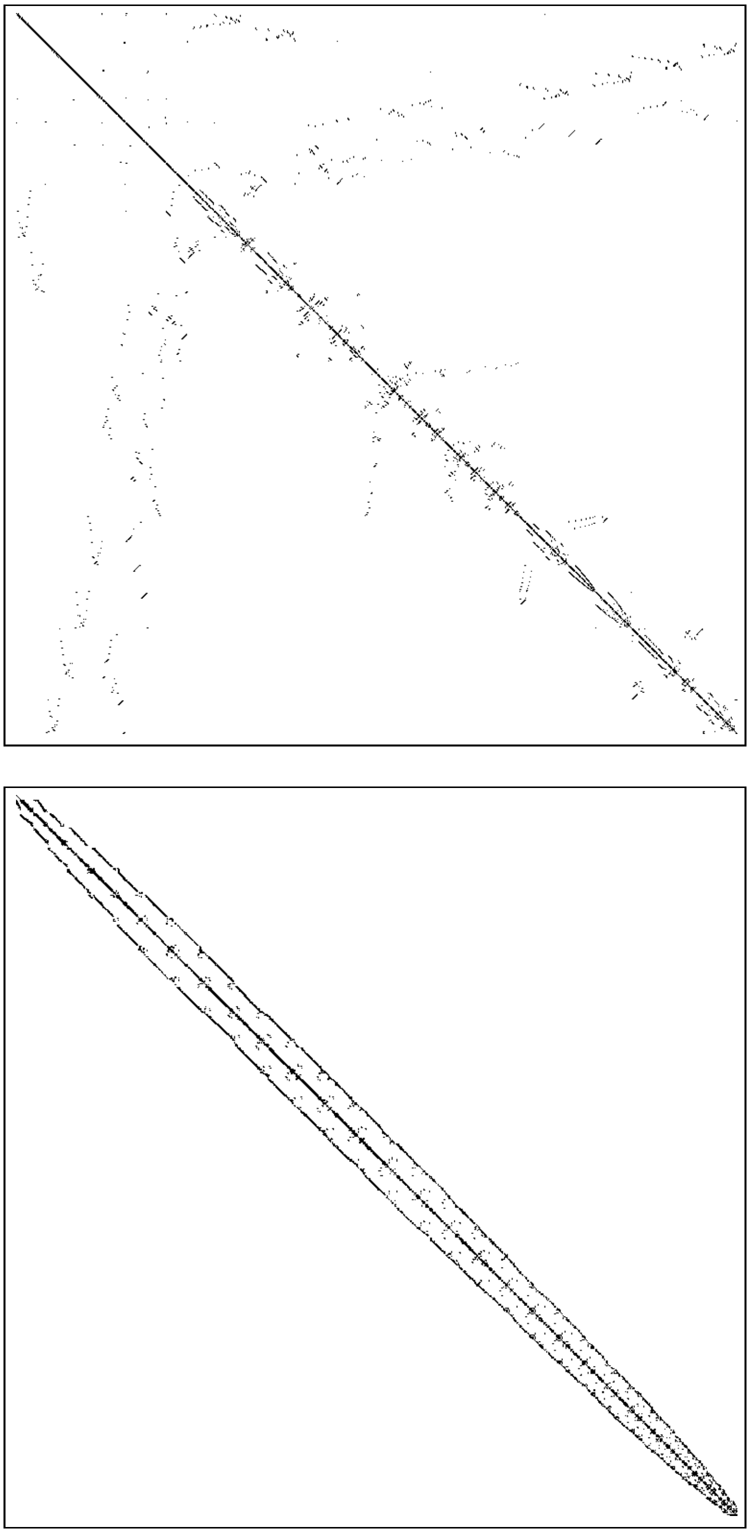

Figure 6.7: Unordered matrix c2253 and result of RCM

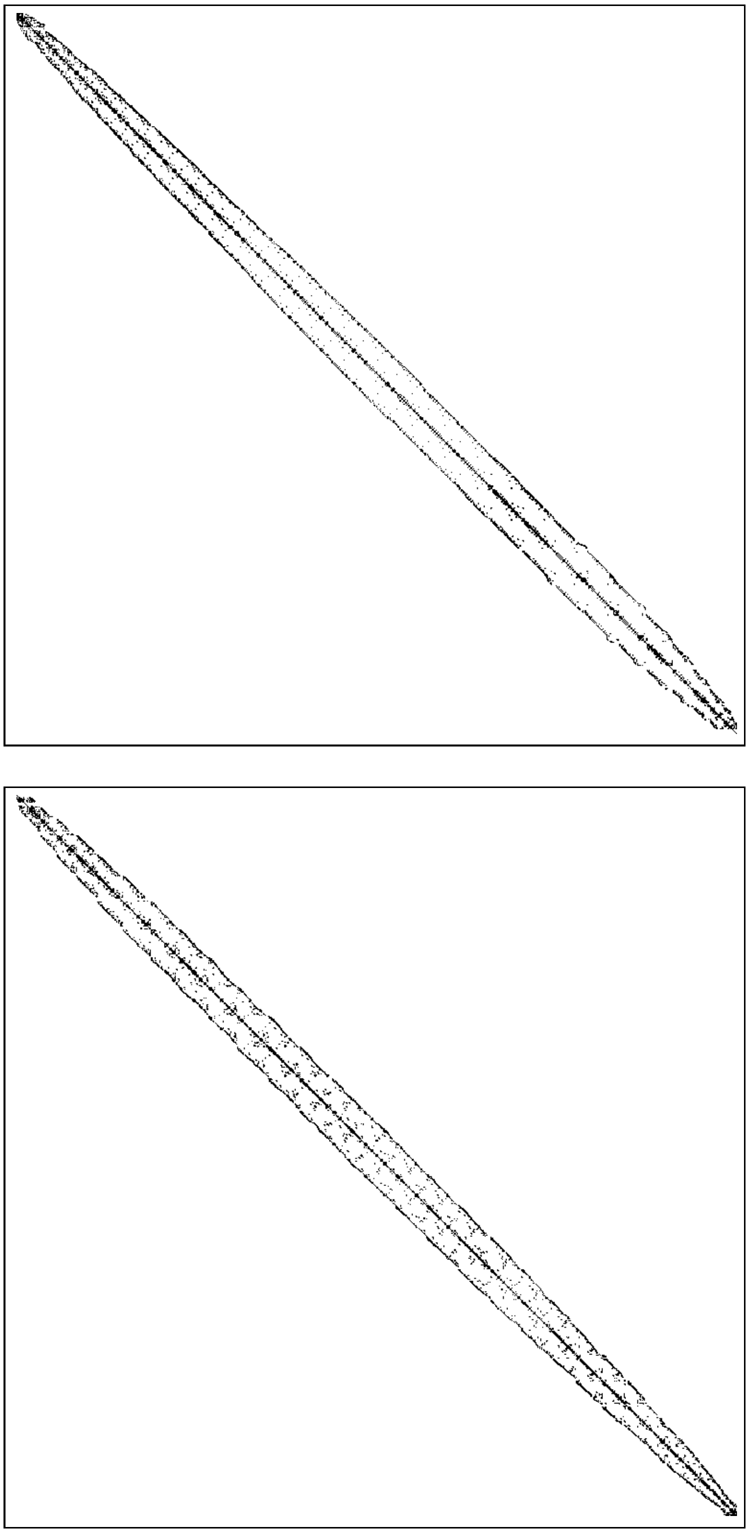

Figure 6.8: Results of SSP and GPS for c2253

exactly 18 entries per row in the symmetric matrix. The resulting matrix forces RCM to iterate over all vertices in the search phase. Nevertheless the example is interesting, as the result for RCM, shown in figure 6.9 is in this case the best attainable numbering for a rooted level structure and the RCM method. The bandwidth of this very expensive result is only 13.5% better than that found by SSP and it is 6.8% better than the GPS result. The respective orderings for these two algorithms are shown in figure 6.10.

Parallel performance of the implementations

The task of renumbering a graph according to a level structure is of course a recursive process. No speedup can be gained by executing the code on more than one processor. In the worst case, the communication overhead even slows down the whole computation. The implementations of RCM and SSP exhibit near optimum behaviour in our tests, namely their execution time for fixed problem size increases only slightly with the number of processors due to the added communication overhead.

The behaviour of the GPS implementation is somewhat more complex and is shown together with the behaviour of SSP in table 6.7. On two processors the communication overhead slows the computation by a factor of two due to the many communications needed in the numbering phase. The better performance of the graph generating module described in section 6.1 more than compensates this however. For 16 processors the execution time of GPS reaches the time that it needs on a single processor. The possible parallel detection of connected components as described in section 4.5.1 accounts for this trend.

Processors	1	2	4	8	16
SSP pure bw opt. in sec.	6.32	7.36	8.67	8.54	8.78
SSP total time in sec.	6.71	7.78	8.96	8.80	9.14
GPS pure bw opt. in sec.	19.55	39.22	28.97	22.48	21.58
GPS total time in sec.	85.44	74.57	48.33	31.39	25.32

Table 6.7: Parallel performance of SSP and GPS

6.2.1 Two dimensional complete LU factorization

This section gives results for the application of the bandwidth optimizers in the FDEM [SA00a] program package. The discretization on a two dimen-

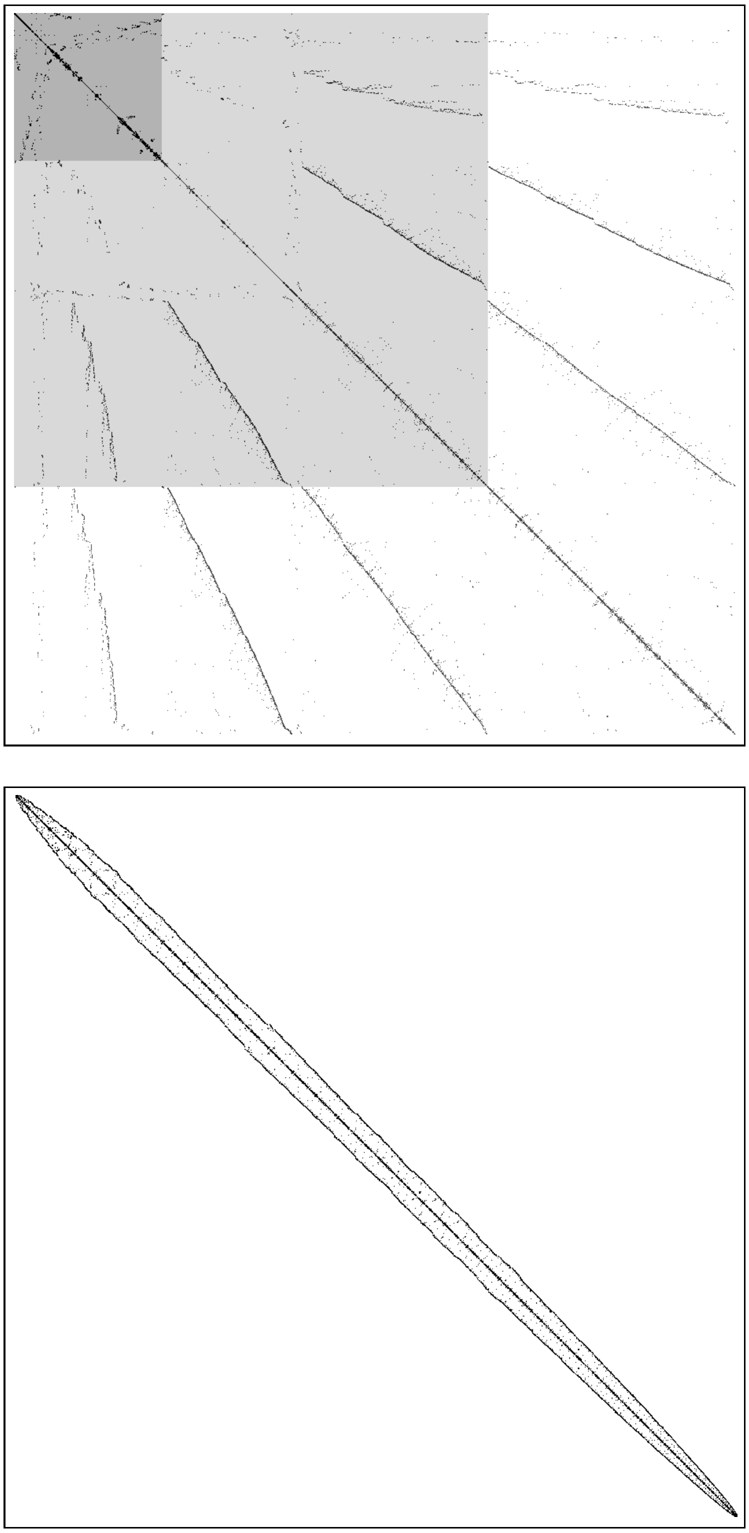

Figure 6.9: Unordered matrix c3669 and result of RCM

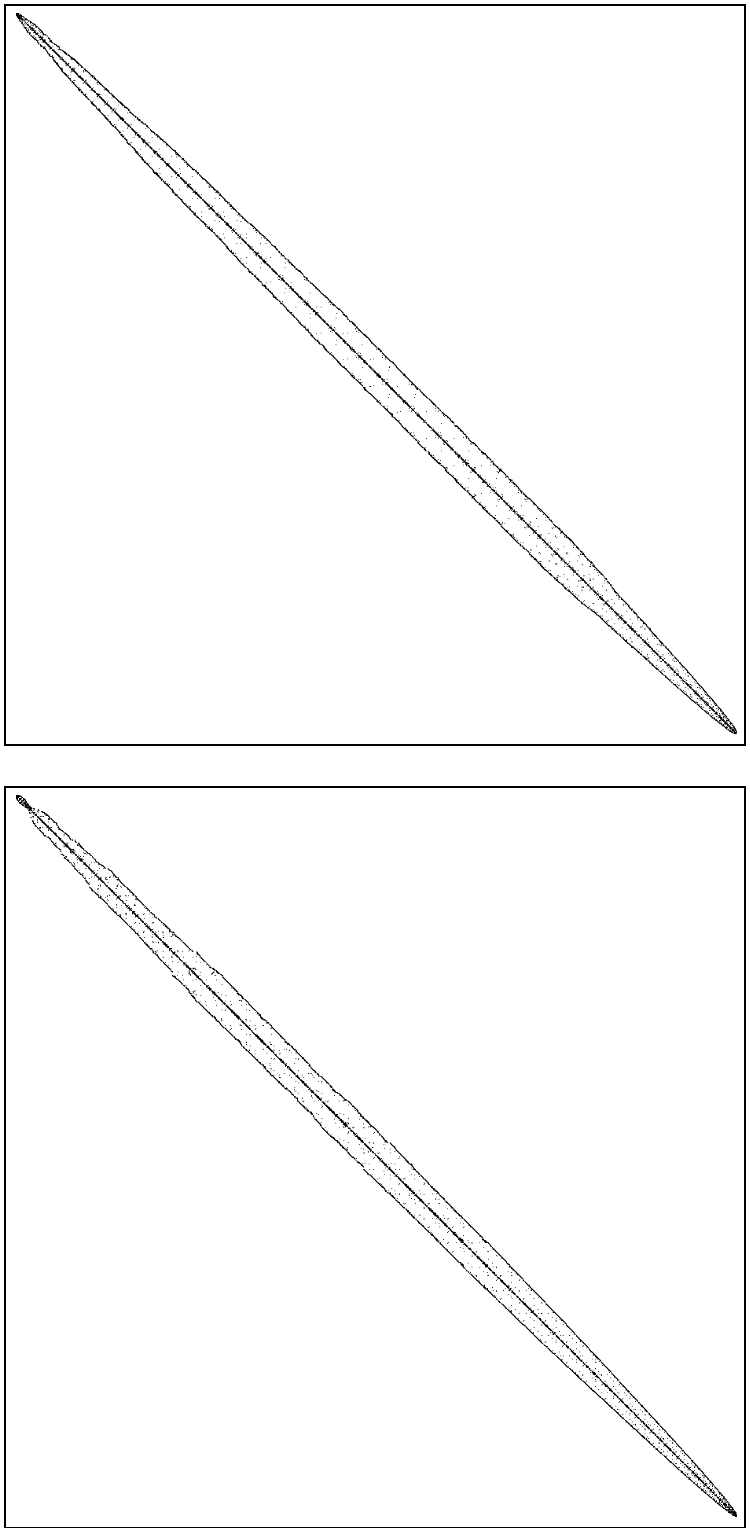

Figure 6.10: Results of SSP and GPS for c3669

sional grid with 65536 points and three equations yields 196608 unknowns. The results shown in table 6.8 are for the full LU version with consistency order 4 on 32 processors. The times are given in seconds.

The bandwidth of the unordered matrix is already quite good. This is the side-effect of the one dimensional domain decomposition described in [SA00a] employed by FDEM for efficient parallel handling of the generated meshes. Because of this pre-processing that can be seen as a very simple graph partitioning scheme, the term "unordered" is misleading in this case.

Although both optimizers can further halve the bandwidth this leads to no relevant improvement with the original bandwidth being only 1/100th of the matrix dimension. As can be seen in table 6.8, bandwidth optimization is only wasted time in this case.

Ordering	$\beta(A_{opt})$	Opt. time	Fact. time	Fill-in	Total time
Unordered	1925	N.A.	923.48	109.9	934.79
RCM	N.A.	> 14400	N.A.	N.A.	N.A.
SSP	922	653.52 (647.61)	733.34	106.2	1400.23
GPS	926	2292.20 (1511.17)	720.91	103.6	3027.70

Table 6.8: Results for bandwidth optimized factorization

6.3 Three dimensional PDE discretization

6.3.1 Three dimensional complete LU factorization

Applying the FDEM package to a three dimensional grid with 16875 points and three equations yields a matrix with 50625 unknowns. The results in table 6.9 are for 32 processors computing the full LU version with consistency order 4.

Ordering	$\beta(A_{opt})$	Opt. time	Fact. time	Fill-in	Total time
Unordered	2622	N.A.	283.44	66.9	287.18
RCM	N.A.	> 14400	N.A.	N.A.	N.A.
SSP	2520	455.83 (453.18)	289.05	66.8	750.08
GPS	2257	1794.57 (1740.99)	282.05	70.8	2081.48

Table 6.9: Results for bandwidth optimized factorization

The effect of the FDEM pre-processing by one dimensional domain decomposition described in the previous section is even more dramatic in this example. The “unordered” bandwidth of 1/20th of the matrix dimension is already that good that SSP can improve on it by only 4%. GPS does somewhat better by improving the result by 14% but the time paid for these results is much too high in both cases and is only wasted time.

Although the storage savings gained by the optimizers might in some cases be valuable, the run time penalty is very high thus limiting their application in the FDEM package.

The examples of section 6.2.1 and section 6.3.1 demonstrate that the efficient data structure used in the FDEM package for the parallelization has an unexpected side effect, namely a nearly optimal bandwidth of the resulting matrix. The bandwidth optimizations have been implemented with the intention to optimize the numbering after each mesh refinement. However, because of the one dimensional domain decomposition we get after each refinement a nearly optimal numbering, and a further bandwidth optimization is thus not needed.

Chapter 7

Conclusion and outlook

> *In the physical world, one cannot increase the size or quantity of anything without changing its quality. Similar figures exist only in pure geometry.*
>
> — Paul Valéry

The results of the measurements shown in the previous chapter are discussed for each implementation. Explanations for their performance, their strengths and weaknesses will be given. A comparison of all results yields a recommendation for the usage of the individual implementations. Possiblities for further improvements end this chapter.

7.1 Discussion of the results

Results of RCM

RCM is the slowest of the three implementations. Even though the implementation does not completely build each level structure as described in chapter 4, the search over possible start vertices takes too much time.

The `c3669` test matrix in section 6.2 shows that the worst case for RCM is not theoretical and may occur often in practice. Even under circumstances not that dramatic, the examples clearly show that it is running orders of magnitude slower than the others with a possible gain of only a few percent in the best case. This seriously limits its application.

The static allocation of arrays used during the computations, can lead to the abortion of the computation when the estimated upper bounds are exceeded. This is a problem for both the RCM and the SSP implementation.

Results of SSP

The implementation of SSP is the fastest in all of our test cases. The main objective of SSP is speed. This speed is bought with larger bandwidths. Except for the one case in section 6.2.1, SSP produced the worst reordering, although the margin here is by far not as big as the margin by which it wins in respect to execution time.

The memory usage of the SSP and RCM implementations can be a problem, when there are few rows in the matrix with exceptionally many non-zero entries. This can lead to the allocation of arrays exceeding the available storage. In our tests, this never happened although the `memplus` matrix came close.

The different strategy of choosing vertices to be numbered does not improve on the results of the procedure used by RCM for only one start vertex. Not shown in the previous chapter is the fact that we also applied such a searchless RCM to the test cases. These tests show that the additional work in this phase done by SSP does not pay off. The non-searching RCM is usually as good as SSP, sometimes better, and in one case marginally worse.

The real novelty of SSP is the serious application of a renumbering scheme that uses only a single run through a level structure. This is its strength and of course its greatest weakness. A bad choice for a start vertex cannot be corrected at anytime during the computation. It is interesting that this is not such a big problem for our test cases as it might seem.

One reason for this certainly is the special class of matrices that we used for the tests. They mostly stem from the discretization of problems on geometrical grids. Interior points of the discretization grid are usually equivalent in respect to the number of connections to neighbouring grid points. Given border conditions result in rows of fewer non-zero entries and thus in vertices of low degree in the graph. The fact that these border conditions are geometrically peripheral usually maps to the fact that the corresponding vertices are peripheral in the graph. Therefore the choice of low degree vertices almost guarantees that they are endpoints of a pseudo-diameter.

For this class of matrices SSP is the most efficient of the three implementations and should be used. For more general matrices, problems can be anticipated. Loosely coupled problems for example, resulting in blocks of non-zero entries in the matrix, coupled only by few rows and columns, will certainly lead SSP to choose such a coupling point as the start vertex. The results will not be optimal in this case. The lack of an “emergency exit” for SSP is enough not to recommend it for general usage.

GPS results

The GPS implementation performs much better than RCM in respect to the execution times and the resulting bandwidths are usually comparable. The current implementation shows some problems in the parallel application that were exhibited in the previous chapter. The serial measurements make it clear however that it will always be slower by a factor of 3–10 in contrast to SSP. This situation cannot be changed by a more efficient parallel implementation.

7.2 Comparison and recommendation of use

The measurements clearly show that using RCM is not efficient in comparison to the other available options to optimize the bandwidth of a matrix. Although it produced the best result in half of the test cases where it reached a result in the provided time, its time requirements are not comparable to SSP or GPS.

As noted in the discussion of SSP above, it is very well suited for matrices from discretizations on geometrical grids. The win in execution time more than compensates the lower quality of the reorderings.

For general use, GPS clearly is the best choice. It is robust, in the sense that it will always run to completion, conservative in its memory usage, and gives good results.

For matrices from discretization steps, SSP seems to be the best choice. This should be consolidated in further tests and the class of matrices for which this holds, should be described more precisely.

7.3 Outlook

A few suggestions are given in this section on how to possibly improve the performance of the current implementations.

7.3.1 Making SSP more robust

Considering the fact that the choice of the SSP start vertex completely determines its outcome, suggests investing more effort in this phase to prevent poor performances. One immediately apparant way is to use the pseudo-diameter search of GPS in connection with SSP. This might be enough to prevent especially poor performances.

7.3.2 Improving the performance of the GPS implementation

The results in the previous chapter show that too much time is spent in the procedure `PREPRO`, external to the GPS implementation to setup the `graph` and `vertix` fields used by it. The comparable pieces from the `SSP` and `RCM` implementation perform better by nearly two orders of magnitude. They do use more memory of course as they work on two-dimensional arrays instead of packed representations but the memory requirements allow them to work on almost all problems. For the GPS implementation to be more competitive, this graph building stage has to be improved substantially. It should remain robust though as this is one of its main advantages.

7.3.3 Using information from the matrix generation stage

When a system of $n > 1$ PDEs is solved on the same discretization grid, the resulting matrix effectively consists of $n \times n$ blocks, sharing the same adjacency relations. This information could be used to build a coarser graph in which each such matrix block is represented as a vertex. This graph consists of only $1/n$ vertices in comparison to the graph used by the current implementations.

This optimization is applicable to all of the implementations shown in this work.

Appendix A

The test matrix

A summary of applying the algorithms to the test matrix presented in chapter 4 is given. Table A.1 summarizes the properties of the original and the reordered matrices.

Reordering	Bandwidth	Profile
Unordered	21	238
RCM	7	97
SSP	9	103
GPS	5	98
Multiple Minimum Degree	22	158

Table A.1: Reorderings for the test matrix

The pattern of the unordered test matrix is shown in figure A.1. Figure A.2 shows the reorderings produced by RCM and SSP. The results of applying GPS and Multiple Minimum Degree[1] algorithms are shown in figure A.3.

[1]From the commercial MATLAB package by The MathWorks Inc.

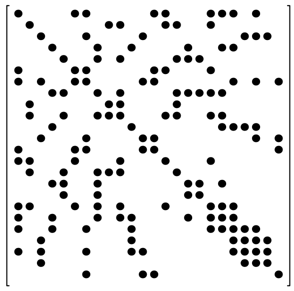

Figure A.1: The unordered test matrix

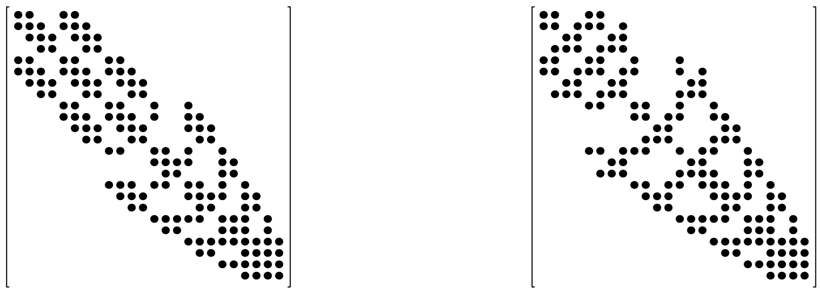

Figure A.2: RCM and SSP reorderings

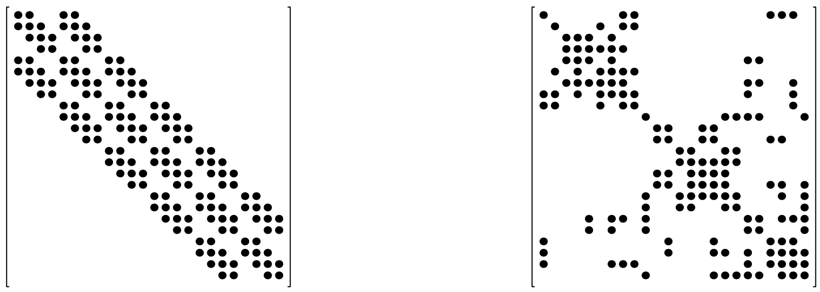

Figure A.3: GPS and Multiple Minimum Degree reorderings

Appendix B

Octave source code for GPS

This section contains a prototype implementation of the GPS algorithm in GNU Octave[1] source code. The free software package Octave is mostly compatible to the MATLAB system. Numerical experiments can be done easily with GNU Octave, as its high-level programming language features expressive array constructs and other useful features for this special purpose. It is very convenient for creating prototypes that usually take much longer to implement in other programming languages. In addition to this, GNU Octave uses well known efficient libraries to achieve good performance.

The source code written for clarity and not for efficiency can be used as a reference for the more involved parallel implementation that has the same underlying structure. It can also be utilized for quick experiments as it can be used immediately from within the Octave framework.

[1] Octave is available on the Internet from `ftp://ftp.che.wisc.edu/pub/octave`, and additional information is available from `http://www.che.wisc.edu/octave`.

```
## Bandwidth optimize matrix MAT according to the GPS algorithm
function [mat_new, permut]=gps(mat)
  false=0;
  true=1;
  n=rows(mat);
  empty_level=zeros(1,n);
  degree=degree_list(mat);
  permut=zeros(1,n);

  ## Phase I:  Find a start vertex
  v=min_index(degree);          # G2
  while (true)
    [levelv, depthv, widthv]=level_structure_for(mat, v, empty_level, 1);
    set_s=sort_indirect(find(levelv==depthv), degree);  # G3
    min_width=inf;
    better_v=false;
    for s=set_s                 # G4
      [levels, depths, widths]=level_structure_for(mat, s, empty_level, 1);
      if (depths>depthv)
        v=s;
        better_v=true;
        break;
      endif;
      if (widths<min_width)     # G5
        u=s;
        min_width=widths;
      endif
    endfor;

    if (!better_v)
      break;                    # Ok, we have found u and v
    endif;
  endwhile;

  ## Phase II: Generate new level structure L
  levell=empty_level;
  [levelu, depthu, widthu]=level_structure_for(mat, u, empty_level, 1);
  levelu=depthv+1-levelu;       # G6
  for i=1:columns(levelv)
    if (levelu(i)==levelv(i))
      levell(i)=levelv(i);      # G7
    endif;
  endfor;

  ## G8 - Find connected components.  Indices for Ci are negative
  c_idx=-1;
  while (any(levell==0))
    levell=level_structure_for(mat, min(find(levell==0)), levell, c_idx);
    c_idx=c_idx-1;
  endwhile;

  c_max=(-c_idx)-1;
  card_c=zeros(1,c_max);
  for c_idx=1:c_max
    card_c(c_idx)=columns(find(levell==(-c_idx)));
  endfor;

  ## G9 - Process Ci
  [foobar, order]=sort(-card_c);
  ni=level_widths(levell, depthv);
  selection_for_c1=0;
  for c_idx=order
    hi=level_widths(level_join(levell, levelv, -c_idx), depthv);
    li=level_widths(level_join(levell, levelu, -c_idx), depthv);
    h0=max(hi(find(hi>ni)));
    l0=max(li(find(li>ni)));
    if (h0<l0)
      levell=level_join(levell, levelv, -c_idx); selected=1;
    elseif (h0>l0)
      levell=level_join(levell, levelu, -c_idx); selected=2;
```

```
    elseif (widthu<widthv)
      levell=level_join(levell, levelu, -c_idx); selected=2;
    else
      levell=level_join(levell, levelv, -c_idx); selected=1;
    endif;
    if (selection_for_c1==0)
      selection_for_c1=selected;
    endif;
  endfor;

  ## Phase III
  if (degree(u)<degree(v))       # G10
    tmp=u; u=v; v=tmp;
    levell=depthv+1-levell;
    a_interchanged=true;
  else
    a_interchanged=false;
  endif;

  newnumber=1;
  permut(v)=newnumber;           # G11/G12
  newnumber=newnumber+1;
  cur_lev=1;
  next_lev=1;
  for lvl=1:depthv
    while true
      cands=find_nr_nb(mat, levell, permut, cur_lev, next_lev);
      if (columns(cands)==0)
        cands=[];
        for el=find(levell==cur_lev);
          if (permut(el)==0)
            cands=[el, cands];
          endif;
        endfor;
        if (columns(cands)>0)
          cands=sort_indirect(cands, degree);
          permut(cands(1))=newnumber;    # G11/12-3
          newnumber=newnumber+1;
          continue;
        else
          break;
        endif;
      endif;
      cands=sort_indirect(cands, permut);
      w=cands(1);                # G11/12-2
      cands=find_nonr_nb_lev(mat, levell, permut, w, next_lev);
      cands=sort_indirect(cands, degree);
      for el=cands
        permut(el)=newnumber;
        newnumber=newnumber+1;
      endfor;
    endwhile;
    if (next_lev>1)
      cur_lev=cur_lev+1;
    endif;
    next_lev=next_lev+1;
  endfor;

  ## G13
  if ((a_interchanged)&&(selection_for_c1==2))|| \
       ((!a_interchanged)&&(selection_for_c1==1))
    permut=depthv+1-permut;
  endif;
  mat_new=permut_matrix(mat, inv_permut(permut));
endfunction;


## Return degrees of rows in matrix m as row vector
function l=degree_list(m)
  l=zeros(1, rows(m));
```

```
  for i=1:rows(m)
    l(i)=columns(find(m(i,:)<>0))-1;
  endfor
endfunction;


## Return index of minimum in row vector
function idx=min_index(v)
  idx=find(v==min(v));
endfunction;


## Build a level structure for a given start vertex U
function [level_out, depth, width]=level_structure_for(mat, u, level, start_lev)
  empty_list_elements_ok=1;

  n=rows(mat);
  width=0;

  cur_level=queue_create;
  cur_depth=start_lev;
  cur_level=queue_enqueue(cur_level, u);
  level(u)=cur_depth;
  used(u)=1;
  if (cur_depth>0)
    cur_depth=cur_depth+1;
  endif;

  while(! queue_empty(cur_level))
    next_level=queue_create;
    cur_width=0;

    while (! queue_empty(cur_level))
      [cur_level, newnode]=queue_dequeue(cur_level);
      for j=1:columns(mat)
        if ((mat(newnode, j)<>0) && (level(j)==0) && (j<>newnode))
          next_level=queue_enqueue(next_level, j);
          used(j)=1;
          level(j)=cur_depth;
          cur_width=cur_width+1;
        endif;
      endfor;
    endwhile;

    cur_level=next_level;
    if (cur_depth>0)
      cur_depth=cur_depth+1;
    endif;
    width=max(cur_width, width);
  endwhile;

  level_out=level;
  depth=cur_depth-2;
endfunction;


## Join level structures LEVELMASK and LEVEL
function level_out=level_join(levelmask, level, mask)
  level_out=zeros(1,columns(level));

  for i=1:columns(level)
    if (levelmask(i)==mask)
      level_out(i)=level(i);
    else
      level_out(i)=levelmask(i);
    endif;
  endfor;
endfunction;
```

```
## Determine widths of levels in levelstructure LEVEL
## (which has max. depth of DEPTH)
function widths=level_widths(level, depth)
  widths=zeros(1,depth);
  for i=1:depth
    widths(i)=columns(find(level==i));
  endfor;
endfunction;


## Find unnumbered neighbours of W in level LVL
function cands=find_nonr_nb_lev(mat, level, permut, w, lvl)
  empty_list_elements_ok=1;
  cands=[];

  for el=find(mat(w,:)<>0)
    if (level(el)==lvl)&&(permut(el)==0)
      cands=[el, cands];
    endif;
  endfor;
endfunction;


## Find all numbered vertices in level NR_LEV with unnumbered neighbours
## in level NB_LEV
function cands=find_nr_nb(mat, level, permut, nr_lev, nb_lev)
  empty_list_elements_ok=1;

  n=columns(level);
  cands=[];
  for i=1:n
    if (level(i)==nr_lev)&&(permut(i)<>0)
      if columns(find_nonr_nb_lev(mat, level, permut, i, nb_lev))>0
        cands=[i, cands];
      endif
    endif;
  endfor;
endfunction;


## Sort indizes by magnitude of indexed elements
function sorted=sort_indirect(ind, list)
  elements=list(ind);
  [elements, order]=sort(elements);
  sorted=ind(order);
endfunction;


## Permute matrix MAT row- and columnwise by rowvector PERM
function mat_new=permut_matrix(mat, perm)
  mat_new=mat(perm,perm);       # Permute matrix row- and columnwise
endfunction;


## Invert permutation vector PERM
function inv=inv_permut(perm)
  [foobar, inv]=sort(perm);
endfunction;


## Create an empty queue
function queue=queue_create
  queue=[];
endfunction;


## Add an element onto a queue
function newqueue=queue_enqueue(queue, el)
  newqueue=[queue, el];
```

```
endfunction;


## Dequeue an element from a queue
function [newqueue, el]=queue_dequeue(queue)
  el=queue(1);
  newqueue=queue(2:columns(queue));
endfunction;


## Queue empty predicate
function pred=queue_empty(queue)
  pred=(columns(queue)==0);
endfunction;
```

Bibliography

[ASS96] Harald Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.

[Bac78] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communicatios of the ACM*, 21(8):613–641, August 1978.

[BPS95] S. T. Barnard, A. Pothen, and H. D. Simon. A spectral algorithm for envelope reduction of sparse matrices. *Numerical Linear Algebra with Applications*, pages 317–334, 1995.

[Can91] D. C. Cann. Retire Fortran? A Debate Rekindled. *Supercomputing 91*, November 1991.

[Can92] D. C. Cann. The optimizing SISAL compiler: Version 12.0. Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, April 1992.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of 24th Nat. Conf. Assoc. Comp. Mach.* ACM Publications, 1969.

[Fie75] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its applications to graph theory. *Czechoslovak Math. J.*, 25:619–633, 1975.

[Geo71] Alan George. Computer implementation of the finite element method. Technical Report STAN-CS-71-208, Computer Science Dept., Stanford Univ., Stanford Calif., 1971.

[Geo73] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal of Numerical Analysis*, 10(2):345–363, 1973.

[GL81] A. George and J. W–H. Liu. *Computer solution of large sparse positive definite systems*. Prentice-Hall, 1981.

[GL89] A. George and J. W–H. Liu. The evolution of the minimum degree ordering. In *SIAM Review*, volume 31, pages 1–19, 1989.

[GPS76] Norman E. Gibbs, William G. Poole Jr., and Paul K. Stockmeyer. An algorithm for reducing the bandwidth of a sparse matrix. *SIAM Journal of Numerical Analysis*, 13(2):236–249, April 1976.

[GRS94] L. Grosz, C. Roll, and W. Schönauer. Nonlinear finite element problems on parallel computers. In J. Dongarra and J. Wasniewski, editors, *Parallel Scientific Computing, 1. Internat. Workshop PARA'94*, Lecture Notes in Computer Science, pages 247–261, Berlin, 1994. Springer Verlag.

[HL95] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. *ACM Proc. Supercomputing 95*, December 1995.

[HS94] Hartmut Häfner and Willi Schönauer. Kommunikation auf verschiedenen Parallelrechnern, Auswirkung auf die Parallelisierung. In A. Schreiner and E. Schnepf, editors, *Drittes ODIN Symposium*, pages 195–206. Rechenzentrum Universität Karlsruhe, 1994.

[HSW99a] H. Häfner, W. Schönauer, and R. Weiss. Parallelization and integration of the LU and ILU algorithm in the LINSOL program package. In V. Malyshkin, editor, *Parallel Computing Technologies*, Springer Lecture Notes in Computer Science 1662, pages 417–427. Berlin, 1999.

[HSW99b] Hartmut Häfner, Willi Schönauer, and Rüdiger Weiss. The program package LINSOL: Basic concepts and realization. *Applied Numerical Mathematics*, 30:213–224, 1999.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, (29):291–307, 1970.

[Knu73] Donald Knuth. *The Art of Computer Programming*, volume 3 – Sorting and Searching. Addison Wesley, 1973.

[KP99] Gary Kumfert and Alex Pothen. An Object-oriented Collection of Minimum Degree Algorithms: Design, Implementation, and

Experiences. Technical Report 99-1, Institute for Computer Applications in Science and Engineering, January 1999.

[Liu85] Joseph W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11:141–153, 1985.

[Meu99] Gerard Meurant. *Computer Solution of Large Linear Systems*, volume 28 of *Studies in Mathematics and its applications*. North Holland, 1999.

[SA00a] Willi Schönauer and Torsten Adolph. The FDEM (Finite Difference Element Method) program package: Fully parallelized solution of elliptic and parabolic PDE's. *to appear in Proc. of ECCOMAS 2000, Barcelona, Sept.*, 2000.

[SA00b] Willi Schönauer and Torsten Adolph. How WE Solve PDEs. *to appear in Journal of Computational and Applied Mathematics*, 2000.

[Sch87] Willi Schönauer. *Scientific Computing on Vector Computers*. North Holland, Amsterdam, 1987.

[Sch99] Willi Schönauer. Private communications, 1999.

[Sch00] Willi Schönauer. *Scientific Supercomputing – Architecture and Use of Shared and Distributed Memory Parallel Computers*. Self-Edition by Willi Schönauer, Germany, ISBN 3-00-005484-7. Information is available under the URL http://www.uni-karlsruhe.de/Uni/RZ/Personen/rz03/book/, 2000.

[SS92] M. Schmauder and W. Schönauer. CADSOL – a fully vectorized black-box solver for 2-D and 3-D partial differential equations. In R. Vichnevetsky, D. Knight, and G. Richter, editors, *Advances in Computer Methods for Partial Differential Equations*, volume VII, pages 639–645. IMACS, 1992.

[SSW87] W. Schönauer, M. Schlichte, and R. Weiss. Wrong Ways and Promising Ways towards robust and efficient iterative linear solvers. In R. Vichnevetsky, D. Knight, and G.Richter, editors, *Advances in Computer Methods for Partial Differential Equations*, volume VI, pages 7–14. IMACS, 1987.

[Vol91] Lutz Volkmann. *Graphen und Digraphen*. Springer-Verlag, 1991.

[WHS95] Rüdiger Weiss, Hartmut Häfner, and Willi Schönauer. LINSOL Description and User's Guide for the parallelized version. Technical Report 61/95, Rechenzentrum der Universität Karlsruhe, 1995.

[Yan81] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal of Algebraic and Discrete Methods*, 2:77–79, 1981.