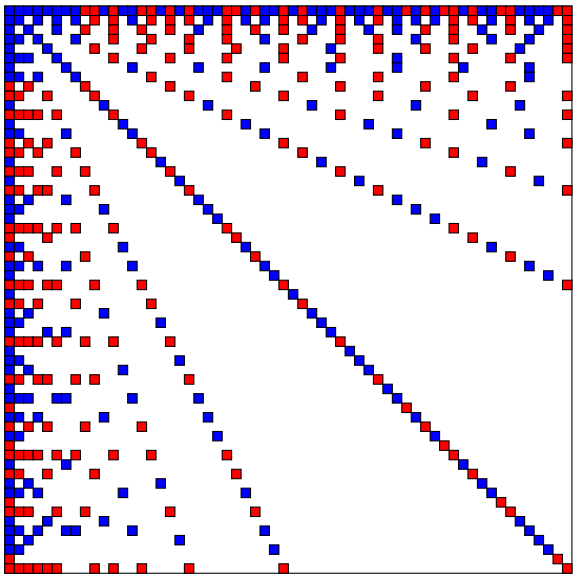
Matrix Partitioning: Optimal bipartitioning and heuristic solutions

Master Thesis Scientific Computing



Daniël Maria Pelt

August 26, 2010

Universiteit Utrecht

[Faculty of Science]

Supervisor: Prof. Dr. R.H. Bisseling

Cover image: Optimal 2-way partitioning of the 60×60 matrix `prime60`, consisting of 462 nonzeroes. The `prime60` matrix elements are defined by:

$$a_{ij} = \begin{cases} 0 & \text{if } i \mod j \neq 0 \text{ or } j \mod i \neq 0 \\ 1 & \text{otherwise} \end{cases}$$

The load imbalance of this partitioning is no larger than $\varepsilon = 3\%$, and its communication volume is equal to 14. The partitioning was calculated by the optimal branch and bound method described in this thesis.

ABSTRACT

An important component of many scientific computations is the matrix-vector multiplication. An efficient parallelization of the matrix-vector multiplication would instantly lower computation times in many areas of scientific computing, and enable solving larger problems than is currently possible. A major obstacle in development of this parallelization is finding out how to distribute the matrix data over all processors without introducing a large communication cost. This problem can be accurately modelled by the matrix partitioning problem with an appropriate cost function.

In this thesis, an optimal branch-and-bound algorithm for 2-way matrix partitioning is developed. Since matrix partitioning is NP-Complete, a polynomial time algorithm is not expected to exist. However, several lower bounds on the cost of branches of the branching-tree are introduced in this thesis, reducing the computation time of the optimal algorithm significantly. Comparisons with Integer Linear Programs that solve the same problem show that the branch-and-bound method is efficient in solving the 2-way matrix partitioning problem. The algorithm was able to find the optimal 2-way partitioning of 75% of all matrices of the University of Florida Sparse Matrix Collection [12] with at most 200 rows or columns. Finally, several suggestions for further research on optimal bipartitioning matrices are given.

In Chapter 3, using information on common characteristics of optimal solutions, a new heuristic method is developed that attempts to find good solutions in polynomial time. This heuristic is not based on hypergraphs, as most publicly available partitioners are, but uses a novel partitioning model. The model assigns individual rows and columns to different processors, ensuring a low computation time, and allows for 2-dimensional partitionings, ensuring a high solution quality. In its current form, the heuristic partitions the matrix in p parts directly when solving the p-way partitioning problem, avoiding the possible negative effects of recursive bisection. Furthermore, a multilevel scheme is included to enable solving extremely large matrices in reasonable time. Comparison of the solution quality of the new heuristic with that of the state-of-the-art matrix partitioning software `Mondriaan 3.0` [40] shows that for some matrices, the new heuristic performs significantly better.

Finally, in Appendix A, an orthogonal array testing method is applied in order to find better default values for the parameters of `Mondriaan 3.0`. This results in values that produce solutions that have, on average, a 10.4 % smaller cost than the default values would produce. Furthermore, the results indicate that the quality of the initial partitioning in the multilevel scheme has a large impact on the final solution quality.

CONTENTS

1 INTRODUCTION

1.1 SPARSE MATRIX–VECTOR MULTIPLICATION

The main focus of this Master thesis is on *sparse matrix partitioning*. A matrix A is called *sparse* if a large number of its elements is equal to zero, such that it is worthwhile to exploit this property. The question remains what proportion of elements has to be zero in order to call the matrix sparse. In practice, the answer to this question depends highly on the application of the matrix. A pragmatic stance is taken in this thesis: the matrices considered in this report are either sparse in view of the definition given above, or because they are taken from a well known database of sparse matrices, such as [12].

The same operations that can be applied to a regular matrix can also be applied to a sparse matrix. However, since we are able to take advantage of the sparsity of the matrix, these operations can often be performed much more efficiently, and therefore, much larger problems can be solved by using sparse matrices. For example, multiplying an $n \times n$ matrix containing N nonzero elements with a vector of length n takes $O(n^2)$ time for regular matrices[1], but can be performed in $O(N)$ time using a sparse matrix representation. Since $N \ll n^2$ (by definition of sparsity), the second operation is much more efficient asymptotically than the first. However, although taking advantage of the sparsity of the matrix allows us to perform very large matrix-vector multiplications, even larger problems can be solved by performing this multiplication in *parallel*. In other words, we need to write the multiplication algorithm such that it can be executed by p processing units efficiently.

The standard method to parallelize matrix-vector multiplications is explained in [7]. The parallel algorithm is specifically described as an example of the Bulk Synchronous Parallel (BSP) programming model [39], but the overall design is similar across all parallel programming models. Before execution of the algorithm, the N nonzero elements of matrix A are distributed over the p processors, along with the $2n$ vector components, consisting of the input vector $\vec{v}$ and the output vector $\vec{u}$. Each processor i is given a subset of A, called the *local* subset A_i. Then, the algorithm calculates $\vec{u} = A\vec{v}$ in four steps: the *fanout* (1), the *local multiplication* (2), the *fanin* (3) and finally the *partial sum summation* (4). During the fanout, input vector elements are communicated between processors, based on the requirements of the different processors. Communication is necessary if an input vector element that needs to be multiplied with an element of the local subset is owned by a different processor. After the fanout, all elements of the local subset are multiplied by the corresponding input vector element and partial sums are created for each output vector element. This operation is performed locally by all processors, and no communication is therefore needed. After this local multiplication step, all nonzero partial sums need to be communicated to the processor that owns the corresponding output vector element. This step is called the fanin. Finally, each processor sums up the partial sums for each output vector element it owns. After these operations, the correct values of $\vec{u}$ are known.

The time it takes to compute $\vec{u}$ depends mainly on two factors: the number of multiplications that need to be performed in step (2) and the number of elements that need to be communicated in steps (1) and (3). During the former, every element of the local subset has to be multiplied and

1 In this thesis, the big-O notation is frequently used. More information on this notation can be found in most undergraduate textbooks on computational science, such as [10]. A short reminder is given here: if a function $f(x)$ is $O(g(x))$, a constant c exists, such that $f(x) \leqslant c \cdot g(x)$ for large enough x.

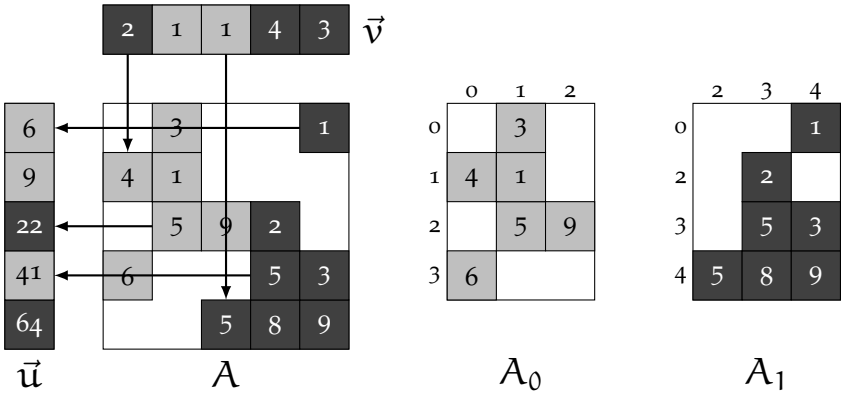


Figure 1.: An example of a parallel matrix-vector multiplication, taken directly from Figures 4.3 and 4.4 of [7]. The numbers inside the cells denote the numerical values a_{ij}, u_i or v_j. Grey cells are owned by processor 0, while dark cells are owned by processor 1. Vector element u_i is shown to the left of the matrix row that produces it; vector element v_j is shown above the matrix column that needs it. Arrows are used to denote the needed communication pattern.

each processor can work in parallel, so the total time requirement of this local multiplication step is $O(\max_i |A_i|)$. In order to minimize this computation time, $\max_i |A_i|$ should be as small as possible: we need to balance the available work between all processors. In practice, this work balance is achieved by defining an allowed *load imbalance* ε, and imposing that the following equation is obeyed by the matrix distribution:

$$\max_i |A_i| \leqslant (1+\varepsilon)\frac{N}{p} \tag{1.1}$$

Minimization of the communication cost of steps (1) and (3) is a much more complicated issue, and will be the main subject of this thesis. Communication of an input vector element is necessary if it is owned by a different processor than an element of a corresponding *column* of A. Similarly, communication for an output vector element is necessary if it is owned by a different processor than a corresponding *row* element. In order to clarify this, Figure 1 shows an example of a parallel matrix-vector multiplication of a 5×5 matrix, using 2 processors. Since $N = 13$, $|A_0| = 6$ and $|A_1| = 7$, the load imbalance $\varepsilon \approx 0.077$. Once an input vector element is communicated to a certain processor, its value can be memorized. Furthermore, before sending row elements to the processor owning an output vector element, all local row values are summed in the partial sum, so only one value (the partial sum) has to be communicated. Therefore, every communication operation of the algorithm consists of sending one value, and each arrow in Figure 1 represents one communication unit. Since there are 5 arrows present in Figure 1, the total communication cost is equal to 5. Note however that we can easily lower this cost to 4 by changing the owner of the first input vector element to processor 0, since all elements of the first column of A are owned by processor 0.

In general, communication along a single column during the fanout step is only strictly needed when the matrix elements of that column are assigned to different processors. In fact, the necessary communication cost of a single column i is equal to:

$$C(i) = \lambda_i - 1 \tag{1.2}$$

where λ_i is the number of different processors that have one or more nonzeroes of that column in their local subset. In order to prove this, let S_i be the list of processor indices that appear in column

Figure 2.: Comparison of an ordinary graph (left) and a hypergraph (right). Notice that a graph can only connect two vertices with each other, while hyperedges (dashed) can connect an arbitrary number of vertices.

i. Only one processor can own the corresponding input vector element v_i. If $v_i \notin S_i$, the value of v_i has to be communicated to all $|S_i|$ processors, inducing a communication cost of $|S_i|$. On the other hand, if $v_i \in S_i$, then one communication unit can be eliminated: the value of v_i is already known by the processor that owns it, so it does not have to be communicated to itself. Therefore, the communication cost is now equal to $|S_i| - 1$. Since we are allowed to choose the owner of the vector elements freely, as they do not affect the load imbalance of Equation 1.1, we can always choose them such that the communication cost of each column is $|S_i| - 1$. Looking at the definition of λ_i in Equation 1.2, we see that $\lambda_i = |S_i|$, thereby completing the proof.

A similar proof can be constructed for rows instead of columns by letting S_i be the list of processor indices that appear in row i. Again, if $u_i \notin S_i$ we need to communicate $|S_i|$ partial sums to the processor that owns u_i, and if $u_i \in S_i$, we can eliminate one communication unit, making the communication cost of the second option equal to $|S_i| - 1$. Since the u_i elements can also be chosen freely, we are always able to choose them such that the communication cost of each row is $|S_i| - 1$. We see that Equation 1.2 is equally valid for rows instead of columns.

By combining both equations for rows and columns and noting that the communication cost of each row and column is independent of the others (there is no communication across different rows or columns), we can define a very important cost function: the total necessary communication cost of a distributed matrix A:

$$V = \sum_i C(i) \tag{1.3}$$

where the sum is taken over *all* rows and columns of the matrix. This measurement of the communication cost of a matrix partitioning is called the *communication volume*. It is a function of both the matrix A, and a partitioning of its nonzero elements over p processors. The rest of this master thesis will deal with methods to minimize this communication volume.

1.2 HYPERGRAPH PARTITIONING

In [8, 9], it is shown that the correct way to model the communication volume of a sparse matrix partitioning is to interpret it as a *hypergraph* partitioning instance. A hypergraph is a generalization of an ordinary undirected graph. With ordinary graphs, an edge connects two vertices with each other: it has only two endpoints. In hypergraphs, edges can connect an arbitrary number of vertices with each other, and are often called *hyper*-edges or *nets* in this context. As with normal graphs, vertices and edges can also be assigned weights, denoted by $w(v)$ or $w(e)$. For an example of the difference between ordinary graphs and hypergraphs, see Figure 2.

A k-way partitioning of a hypergraph $H = (V, E)$, is a distribution of the $|V|$ vertices over k subsets $V_0 \ldots, V_{k-1}$, such that $\bigcup_{i=0,\ldots,k-1} V_i = V$ and $V_i \cap V_j = \emptyset$, $\forall i \neq j$. In other words, every

vertex of the hypergraph is assigned to exactly one subset. Similar to matrix partitioning described earlier, we can define an allowed load imbalance ε, which provides a bound on the size of the largest subset:

$$\max_{i=0,\ldots,k-1} |V_i| \leqslant (1+\varepsilon)\frac{|V|}{k} \tag{1.4}$$

After partitioning, a hyperedge is called *cut* if the vertices included in that hyperedge are assigned to different subsets. Furthermore, we can define λ_i as the number of different subsets the vertices of hyperedge e_i are assigned to. We can now state different *cost* functions of a certain partitioning. For instance, the cost of a partitioning can be defined as the number of cut hyperedges. As an analogue to matrix partitioning, we can also define the cost to be the sum of $(\lambda_i - 1)$ over all hyperedges:

$$C(H, V_0 \ldots V_{k-1}) = \sum_{i=0}^{|E|-1} (\lambda_i - 1) \tag{1.5}$$

Given a cost function and allowed load imbalance, the hypergraph partitioning problem asks to find the minimum cost partitioning obeying the load imbalance equation. As was already indicated by the definitions of Equations 1.4 and 1.5, hypergraph partitioning is very similar to matrix partitioning. In fact, in [9], it is shown that both problems are identical to each other if the matrix is converted to the *finegrain* hypergraph model. In the finegrain model, the matrix nonzeroes correspond to the hypergraph vertices, and the rows and columns of the matrix are converted to hyperedges.

Formally, we can state this as follows: given an $m \times n$ matrix A, for every nonzero a_{ij} we create a vertex with label v_{ij} in the hypergraph, for every column i we create a hyperedge $e_{c,i}$, and for every row j we create a hyperedge $e_{r,j}$. Each vertex of the hypergraph is placed in two hyperedges: vertex v_{ij} is placed in $e_{c,i}$ and $e_{r,j}$. The result is that all nonzeroes of a certain row are placed in a single hyperedge, as are all nonzeroes of a certain column. When the cost function is chosen as in Equation 1.5 and the load imbalance as in Equation 1.4, it is easy to see that this hypergraph partitioning instance is identical to the corresponding matrix partitioning instance. If we look at λ_i, in one case it measures the number of different processor indices in a certain matrix row/column, and in the other case it measures the number of different subset indices in a certain hyperedge. Since hyperedges correspond to matrix rows/columns, λ_i is actually measuring the same quantity in both cases. Also, for a given λ_i, the load imbalance equations and cost functions are exactly the same for both matrix and hypergraph partitioning. It follows that both problems are actually the same.

Other hypergraph models than the finegrain model are also possible. For example, in [8], the *row-net* and *column-net* models are explained. In the row-net model, each column of the matrix is converted to a vertex of the hypergraph, and each row is converted to a hyperedge. All columns that have a nonzero in a certain row are placed in the hyperedge corresponding to that row. Formally, for each column i, we create a vertex v_i, and for each row j, we create a hyperedge e_j. Then, for each nonzero a_{ij} of the matrix, we place vertex v_i in hyperedge e_j. The cost function is again chosen to be Equation 1.5, making the model equivalent to sparse matrix partitioning. Since vertices do not correspond directly to matrix nonzeroes, the load imbalance equation has to be rewritten. If v_i is assigned to subset 0, all nonzeroes of column i are assigned to processor 0. Therefore, the size of A_0, the local subset of processor 0, increases by $n_c(i)$, the number of nonzeroes in column i. If we let the weight $w(v_i)$ of vertex i in the hypergraph be equal to $n_c(i)$, we can define the weighted load imbalance equation:

$$\max_{i=0,\ldots,k-1} \left[\sum_{v_j \in V_i} w(v_j) \right] \leqslant (1+\varepsilon)\frac{W}{k} \tag{1.6}$$

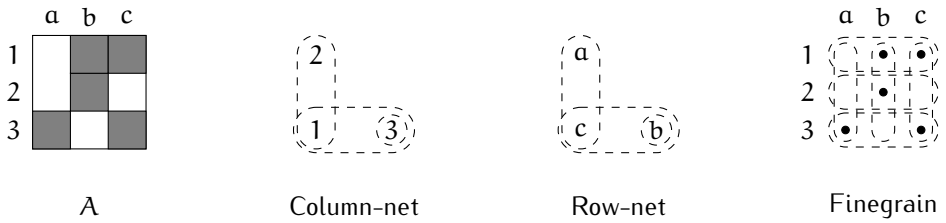


Figure 3.: The three hypergraph models commonly used to model sparse matrix partitioning.

In this equation, W is the sum of all vertex weights: $W = \sum_i w(v_i)$. Since the sum of the weights of all vertices in subset i is equal to the number of nonzeroes in the local subset of processor i, and W is equal to the number of nonzeroes of A, Equation 1.6 correctly models the matrix partitioning load imbalance (Equation 1.1) for the row-net model.

The column-net model is very similar to the row-net model, but with the roles of the rows and columns reversed. The advantage of using the row-net or column-net model is that one source of communication during parallel multiplication is completely eliminated. For instance, using the row-net model, the nonzeroes of a matrix column are always assigned to a single processor. Therefore, no communication is necessary during the fanout phase. Of course, matrix rows can still be cut, so communication during fanin might still occur. The disadvantage of the row-net and column-net models is that they can be too restrictive: sometimes allowing some communication during fanout can help to reduce the overall needed communication volume. The finegrain model does not suffer from this restrictiveness, but has the main disadvantage that it is larger: it contains N vertices, whereas the row-net and column-net models contain m and n vertices respectively. Therefore, computations on the finegrain model of a matrix take longer than computations on the row-net and column-net models of the same matrix, and its large size can make it harder to find good solutions to the finegrain hypergraph partitioning problem. Figure 3 shows all three hypergraph models (finegrain, row-net and column-net) for a small 3×3 example matrix.

1.3 EARLIER WORK

As hypergraph partitioning has many more applications other than modelling sparse matrix-vector multiplications, such as VLSI design [3], the problem has been studied before in the scientific community. However, it has been known for a long time that the hypergraph partitioning problem is NP-Complete ([18]), so it is not expected that a polynomial-time algorithm solving the problem exists. In fact, very little previous research is available that discusses optimal solutions to the hypergraph partitioning problem (for an example, see [41]). Almost all past research on this subject has focussed on developing heuristics for this problem: methods that try to find solutions of low cost in polynomial time, but provide no guarantee on the quality of these solutions. PaToH ([38]), hMetis ([27]) and Mondriaan ([40]) are some of the most popular heuristics available at this moment. Of these, Mondriaan is specifically designed to solve the matrix partitioning problem, while PaToH and hMetis are general hypergraph partitioners that can also be used to partition matrices.

While these partitioners are different in solution quality and execution time, all use a similar method to find good solutions: the Kernighan-Lin method described in [28], specifically with the optimizations of Fiduccia-Mattheyses ([16]). Kernighan-Lin is a local search method, meaning that it works by improving already found solutions. Details of this method are given in Chapter 3, where a

new partitioning method is designed that also uses Kernighan-Lin. Furthermore, all existing methods are based on *recursive bisection*: instead of partitioning the hypergraph into k parts directly, it is recursively split into two parts until k parts are obtained. Although it helps the partitioners, since splitting a hypergraph in two parts is much easier than partitioning it in k parts, recursive bisection has one major flaw: for some problems, the cost of the best solution that can be found by recursive bisection is much larger than that of the optimum solution, as explained in [34]. The new method described in Chapter 3 of this thesis is *not* based on recursive bisection, instead partitioning in k parts directly. Although such methods have been used to partition graphs in the past (see for example [23]), to the best of the author's knowledge, this is the first time that such a method is developed for matrix partitioning.

In order to help the development of better algorithms for sparse matrix operations, some collections of sparse matrix have been compiled, such as the Harwell-Boeing collection [14]. These collections provide a way to compare heuristic methods with each other by defining a set of 'standard' problems. Several collections have been combined and extended in the University of Florida sparse matrix collection [12]. The matrices of this collection will be used in this thesis as problem instances for matrix partitioning.

2 OPTIMAL SOLUTIONS

In this chapter, an algorithm that finds optimum solutions to the matrix partitioning problem is developed: it finds a partitioning that has the lowest communication volume among all partitionings that obey the load imbalance Equation 1.1. As already stated in Chapter 1, the hypergraph partitioning problem, and therefore the matrix partitioning problem, is NP-Complete. As a result, we do not expect that a polynomial-time algorithm that finds optimum solutions exists. The computation time of the algorithm will probably be an exponential function of the size of the problem, the matrix dimensions m, n and N.

This exponential computation time will make solving large instances extremely difficult using current computer hardware. However, even optimal solutions to small instances can be of great interest. Although these will not have an application for actual matrix-vector multiplications, we can use exactly solved instances of the problem to improve the quality of heuristic methods. A publicly available database of optimally solved matrices could be used to test currently available partitioners. Part of the incentive to start this thesis was to create such a database. Furthermore, knowledge of optimal solutions could also give more insight in the properties of good solutions to the matrix partitioning problems. This could lead to new and possibly better heuristics in the future. As an example, the heuristic that is described in Chapter 3 of this thesis was partly based on visual inspection of optimal solutions to various test matrices.

In order to keep computation times low, only the matrix partitioning problem with $k = 2$ is solved optimally in this thesis. Although this is a significant restriction of the type of problem instances that are solved, knowledge of optimal solutions with $k = 2$ is still very important, since almost all heuristic solvers recursively partition matrices in two parts. Therefore, we can use optimal solutions to test the internal bipartitioners of the heuristics. Furthermore, if the optimal bipartitioner is able to perform efficiently, it should be possible to use it as a component of the recursive methods, thereby improving the heuristic solutions. For the rest of this chapter, we take $k = 2$, unless specified otherwise.

Finally, a stricter version of the load imbalance criterion of Equation 1.1 is used when optimally solving the matrix partitioning problem in this thesis:

$$m_1 = N - m_0 = \left\lfloor (1+\varepsilon)\frac{N}{p} \right\rfloor \tag{2.1}$$

In this equation, m_0 is the number of nonzeroes assigned to processor 0, and m_1 is the number of nonzeroes assigned to processor 1. Partitionings that obey Equation 2.1 are called *feasible* solutions. By using this strict load imbalance equation, we define the sizes of the partitioning subsets in advance, and are able to reduce computation time even more, since the number of possible solutions that obey Equation 2.1 is less than the number of those that obey Equation 1.1.

2.1 BRANCH AND BOUND

As a starting point for developing an optimal algorithm, we could simply check all possible solutions, and pick the one with the lowest volume. Since every one of the N nonzeroes of the matrix can be

assigned to k different processors, there are k^N different unique solutions to the matrix partitioning problem. In order to see this, note that we can uniquely specify a partitioning by a list of N processor indices. The value of element i of the list is equal to the index of the processor to which nonzero i is assigned. As there are k different processor indices, and the list contains N elements, we can produce k^N different lists, each one specifying a unique partitioning.

Suppose that we create a simple algorithm that considers all k^N possible solutions, returning the one with the lowest cost, and can check each solution in one microsecond. Although it will probably already be difficult to make the algorithm perform at such high speeds, the fact that the number of solutions that have to be considered is an exponential function of the matrix size will render this algorithm useless in practice. If we would use the algorithm to find the optimum partitioning over 4 parts of a matrix with 30 nonzeroes, it will take over 36000 *years* to finish. Since a matrix with 30 nonzeroes is actually considered to be extremely small (only 0.6 % of the matrices of [12] have $\leqslant 30$ nonzeroes, with the average matrix having 10^5 nonzeroes), it is not hard to conclude that the proposed naive algorithm is not efficient enough to solve the matrix partitioning problem.

An exponential computation time is a common problem that is encountered when solving NP-Complete problems, and a number of techniques exist that try to enable solving larger instances. One of the most universally applicable of these is called *branch and bound*. With branch and bound methods, we divide the main problem in smaller subproblems. These subproblems are then divided in even smaller parts, and this is repeated recursively until the subproblems become trivial to solve. The lowest cost solution of *all* subproblems is equal to the optimum solution of the main problem. We can visualize each subproblem as a vertex in a *branching*-tree, with the main problem at the top, being the root of the tree, and each subproblem 'branching' into smaller subproblems one level down. This explains the 'branch' part of the branch and bound method.

As it is now, it turns out that the number of trivially solvable subproblems will be equal to the number of possible solutions of the underlying problem. In the case of matrix partitioning, the number of subproblems that need to be considered will still be of $O(k^N)$. In order to reduce this number, we will use bounds on the volume of the optimum solutions to both the main problem and the subproblems to skip entire parts of the branching tree. This part of the method is called 'bounding', and will be explained in further detail below.

2.1.1 Bounding the branching tree

In order to explain the process of bounding more clearly, let P be the main minimization problem. During branching, P is divided in smaller subproblems P_i, and each subproblem is then divided in even smaller subproblems P_{ij}, which are in turn divided in ever smaller subproblems P_{ijk}, etcetera. Suppose that a function $U(Q)$ exists, that returns an upper bound on the cost of the optimal solution of problem Q. For instance, a heuristic solution to Q can provide such an upper bound. We can also use the best solution that is encountered in the branching tree so far as an upper bound. Furthermore, suppose we have access to a second function $L(Q)$, that returns a *lower* bound on the cost of the optimal solution to the (sub)problem Q.

During traversal of the branch tree, we might pass a vertex corresponding to a certain subproblem $P_{ijk\ldots}$, for which:

$$L(P_{ijk\ldots}) \geqslant U(P)$$

In other words, the cost of the optimal solution that can be obtained from subproblem $P_{ijk\ldots}$ can never be smaller than an already known upper bound on the cost of the optimal solution of P. Further traversal of the branches originating from $P_{ijk\ldots}$ is not necessary: we can never find a better solution in that part of the branching tree! Therefore, we can skip this part of the branching tree and move to a different part, potentially avoiding the processing of huge numbers of vertices.

This optimization is the 'bounding' part of branch-and-bound methods, and the most important part in lowering execution time. To be able to perform bounding, we need the functions $U(Q)$ and $L(Q)$. The tighter we can set these bounds (e.g. the lower we can get $U(Q)$ and the higher we can get $L(Q)$), the earlier in the branching tree we can bound branches. Therefore, tighter bounds can lead to extreme computation time reduction of the resulting method. However, there is also a trade-off: if computing the bounds is very inefficient, overall computation time might be larger. In general, a decision has to be made: either fast, loose bounds are used and vertices in the branching tree can be traversed quickly, or slow, tight bounds are used and vertices in the branching tree can only be traversed slowly, but less vertices have to be considered.

2.1.2 Matrix partitioning representation

Before designing a branch-and-bound method for solving matrix partitioning, we will first introduce a novel representation of the problem, along with the needed definitions. In the finegrain model, the matrix *nonzeroes* are partitioned individually over two parts. In this new representation, we partition the matrix *rows and columns*. Each row/column can be assigned to three different subsets, with the following descriptions:

Row/column subset	Description
0	all nonzeroes in this row/column are assigned to processor 0
1	all nonzeroes in this row/column are assigned to processor 1
2	this row/column is cut

Using these descriptions, the cost of a certain partitioning is equal to the number of rows and columns that have been assigned to subset 2.

Since there are $m+n$ rows and columns, and each can be assigned to three different subsets, there are 3^{m+n} different solutions possible. Comparing with the finegrain model, we see that the number of possible solutions in the new representation is smaller if $N > (m+n)\log_2 3 \approx 1.58(m+n)$, since in that case, $3^{m+n} < 2^N$. Even if this is not the case, the number of *feasible* solutions of this new representation will be even smaller. For example, take a nonzero a_{ij} of matrix A. Since a_{ij} can only be assigned to either processor 0 or processor 1, row i can not be assigned to subset 0 when column j is assigned to subset 1, or the other way around. This means that a large portion of the possible solutions will be unfeasible, and these can be skipped during the branch-and-bound method. Furthermore, the number of cut rows and columns in the optimal solution will be small compared to the total number of rows and columns. Therefore, solutions that have a large number of cut rows and columns can be skipped as well. Formally stated, if we have an upper bound $U(P)$ on the optimal communication volume, we can skip all solutions that have $\geqslant U(P)$ cut rows and columns. Because of these properties, the new representation of matrix partitioning is especially well suited for implementation in branch-and-bound methods.

2.1.3 Branch-and-bound implementation

Let each solution to a matrix partitioning problem be described by two lists of length m and n: v_c and v_r. Every element of these lists is equal to one of three possible values: 0, 1 and 2. The value of a list element describes to which subset the corresponding row or column is assigned. We can also define *partial solutions*, in which a list element can also be '$-$', meaning that it is not yet assigned to any subset. For example, if $v_c = \{1, 0, -, -, \ldots, -\}$ it means that the first column of A is assigned to processor 1, the second column to processor 0, and the rest of the columns are not assigned yet.

Using this notation, we can describe the matrix partitioning problem as the problem of finding the best assignment of v_c and v_r, starting with $v_c = \{-, -, -, \ldots, -\}$ and $v_r = \{-, -, -, \ldots, -\}$.

We can also *fix* some rows and columns to a certain subset, and ask what the best assignment of the rest of the rows and columns is. The input of this problem is a partial solution, described above. When we use these problems as sub-problems, we can implement the branch-and-bound method as follows. Starting with v_c and v_r completely unassigned as the root problem P, the sub-problems P_i are created by fixing one of the elements to each of the three possible subsets. One level down in the branching tree, the subproblems P_{ij} are created by fixing one more element. This process is repeated further down the tree, until at the bottom, the subproblems consist of completely assigned lists. Since these represent actual solutions to the matrix partitioning problem, the cost and feasibility of these solutions are trivial to calculate, as is required by the branch-and-bound method. Each vertex of the branching tree has three children, one level down: one for each possible subset. Because there are $m + n$ levels in total (one for each element of v_c and v_r), the total number of trivially solvable subproblems is equal to 3^{m+n}, exactly equal to the number of possible solutions, as expected. The total number of subproblems in an unbounded branching tree is equal to the sum of the number of subproblems in each level:

$$N_{tree} = \sum_{i=1}^{m+n} 3^i = \frac{3}{2}\left(3^{m+n} - 1\right) \tag{2.2}$$

The simplest way to fix elements in the branching tree is to alternately pick an element from v_c and v_r, in the order in which they appear. In other words, the subproblems P_i consist of the partial solutions where the first column is fixed to a certain subset, and the subproblems P_{ij} consist of the subproblems where the first column *and* the first row is fixed, etcetera. In practice, the order in which the rows and columns were picked turned out to have a large influence on the computation time of the algorithm. Although finding the best order was not studied intensively in this thesis, it was observed that picking the rows and columns in decreasing order of number of nonzeroes contained in them was beneficial in most cases.

The advantage of this implementation of the new problem representation in a branch-and-bound method is that we are able to skip large parts of the branching tree when partial solutions become unfeasible, or have a high cost. An example is given in Figure 4, where a dense 2×2 matrix is optimally partitioned into two sets. Note that since we can skip large parts of the tree, we only need to consider 18 of the possible 120 subproblems, reducing the number of subproblems to 15 % of the total. For larger matrices, the reductions are even larger, so we only need to consider an extremely small fraction of the branching tree. The addition of better lower bounds to the method will reduce the number of subproblems even further, and will be discussed next.

2.1.4 Lower bounds on partial solutions

The lower bound problem can be specified formally as follows: given a partial assignment of the row and column values, we want to find a lower bound on the communication volume of all solutions that can be obtained by extending this partial assignment. The better (higher) this lower bound is, the earlier we can discard 'bad' branches of the branching tree. In practice, different lower bounds can often be calculated, and these will have to be combined to provide the best possible bound. In the branch-and-bound algorithm used to obtain the results of this thesis, three different lower bounds were used to reduce the number of tree nodes that needed to be considered. We will demonstrate these bounds on an example partial solution of a 5×5 matrix with 16 nonzeroes, given in Figure 5. The first two rows and columns of this matrix have been assigned: the first row is specified to be *cut*, the second row and first column are assigned to processor 0, and the second column is assigned to

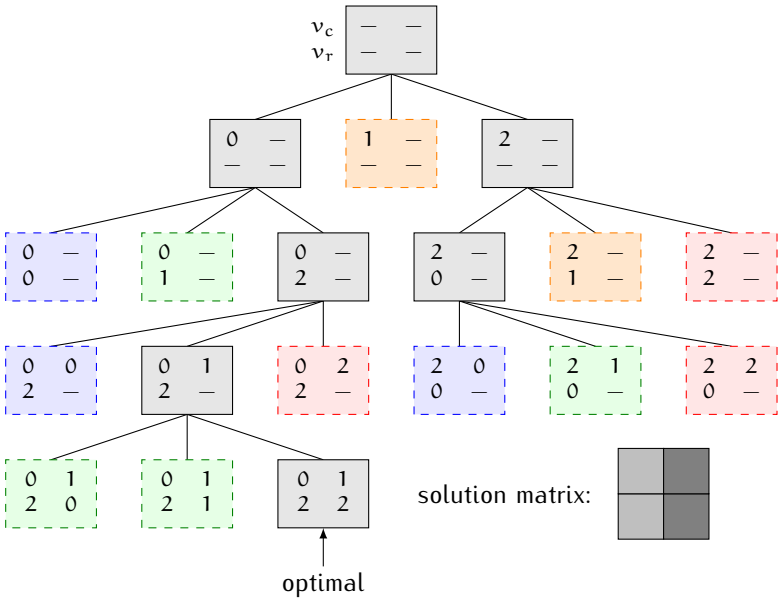


Figure 4.: The branching tree for finding the optimal partitioning of a dense 2×2 matrix over two parts with $\varepsilon = 3\%$ ($m_0 = m_1 = 2$). Each block represents a partial solution, with the two v_c elements shown above the two v_r elements. Dashed blocks represent dead-ends in the tree, where we can skip to the next block. The color indicates the reason why a particular block is a dead-end: blue blocks can be skipped because they violate the strict load imbalance, green blocks are infeasible partial solutions, red blocks represent partial solutions that have a communication volume greater or equal than the best known solution, and orange blocks are skipped because of symmetry considerations (the problem is symmetric in processor index). The tree is traversed in depth-first order, from left to right. In the optimal solution that is found (shown in the bottom right corner), the first column is assigned to processor 0 (light gray), the second column to processor 1 (dark gray), and both rows are cut. The resulting communication volume is equal to 2.

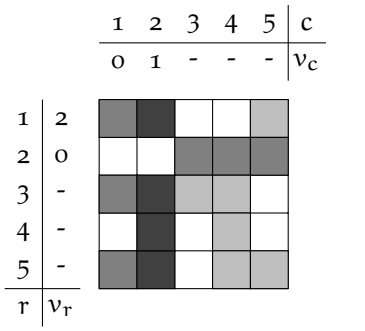


Figure 5.: An example partial solution with the first two rows and columns assigned (dark gray nonzeroes are assigned to processor 1, gray nonzeroes are assigned to processor 0 and light gray nonzeroes are not yet assigned). A lower bound on the communication volume of this partial solution is 5 (see text), while the optimum solution is equal to 4 (see Figure 6).

processor 1. The load imbalance is taken to be 3 %, which in this case means that we can distribute 8 nonzeroes to each processor. The question is: what is a good lower bound on the communication volume of all solutions extended from this partial solution?

The first lower bound (b_1) is fairly straightforward: it is the number of already explicitly cut rows and columns. In other words, the number of rows and columns with a solution value of 2 is a lower bound. This is almost trivial to prove, since any solution extended from this partial solution will also include these cut rows and columns. In the example, only the first row is defined to be cut, so $b_1 = 1$. Note that this bound is the same bound that was used to skip the red blocks of Figure 4.

For the second lower bound (b_2), suppose that we have two columns that both have a nonzero in a row that is yet unassigned. If these two columns are assigned to different processors, we know that the corresponding row *has* to be cut, in order to maintain a feasible solution. In the example, columns 1 and 2 both have a nonzero in row 3, but are assigned to different processors. As is clear in Figure 5, row 3 has to be cut in order to have a consistent solution. Of course, the same lower bound applies for rows instead of columns. In the example, rows 3 and 5 have to be cut, since columns 1 and 2 are assigned to different processors, and therefore $b_2 = 2$. Note that row 1 is not counted in b_2, since it has a solution value of 2 and is therefore already explicitly cut.

The third lower bound (b_3) is slightly more complicated and based on partially assigned rows and columns. For example, if a certain row is assigned to processor 0, all nonzeroes in that row are also assigned to processor 0 by definition of the row assignment. The unassigned columns that these nonzeroes belong to are now *partially* assigned: some of its nonzeroes are assigned to processor 0, and some nonzeroes are still unassigned. In the example of Figure 5, columns 3,4 and 5 are partially assigned to processor 0, and row 4 is partially assigned to processor 1. Note that if a row or column is partially assigned to *both* processor 0 and 1, it is included in b_2, and therefore will not be included in b_3. In order to avoid cutting a column that is partially assigned to processor 0, all remaining unassigned nonzeroes of that column have to be assigned to processor 0 as well. This consideration is the basis of the third lower bound.

Because of the strict load imbalance equation, we are allowed to assign a maximum of m_0 nonzeroes to processor 0. Let n_0 be the number of nonzeroes already assigned to processor 0 in the partial solution, and $s_{c,0}$ be the number of unassigned nonzeroes in all columns that are partially assigned to processor 0. Since $m_0 - n_0$ is the number of nonzeroes that can still be assigned to

processor 0, if $m_0 - n_0 < s_{c,0}$, it is impossible to assign all $s_{c,0}$ nonzeroes to processor 0: cutting one or more columns is unavoidable! The question remains: which columns have to be cut, such that the resulting communication volume is as small as possible? Since the cost of cutting a column is always 1, independent of the number of unassigned nonzeroes in that column, it is best to cut the columns in decreasing order of number of unassigned nonzeroes. In order to see this, note that we can stop cutting columns once $m_0 - n_0 \geqslant s_{c,0} - N_C$, where N_C is the number of unassigned nonzeroes in all partially assigned columns that were just cut. When $m_0 - n_0 \geqslant s_{c,0} - N_C$, enough nonzeroes are left to fill all remaining uncut columns that are partially assigned to processor 0. If we cut the partially assigned columns with the most unassigned nonzeroes first, we ensure that N_C is as large as possible at all times, and that we never cut too many columns. The number of columns that have to be cut until $m_0 - n_0 \geqslant s_{c,0} - N_C$ is a lower bound on the communication volume of the partial solution. Similar but independent bounds can also be calculated for the unassigned rows, and for processor 1. In total, there are 4 such bounds: for each processor, there is a bound for the rows and the columns. Since these bounds are independent of each other, we can define b_3 as the sum of all four bounds.

As an example, we will now calculate b_3 for the partial solution of Figure 5. The third, fourth and fifth columns of the matrix are partially assigned to processor 0, and contain 6 unassigned nonzeroes, so $s_{c,0} = 6$. There are no columns partially assigned to processor 1, and therefore $s_{c,1} = 0$. As for the rows, all rows except the fourth are either assigned or included in b_2, leading to $s_{r,0} = 0$ and $s_{r,1} = 1$. Finally, it is easy to count that $n_0 = 6$ and $n_1 = 4$. Using these values for the variables, we will now check whether:

- $m_0 - n_0 < s_{c,0}$:
  $m_0 - n_0 = 2$ and $s_{c,0} = 6$, so we cannot avoid to cut some of the columns. The column with the most unassigned nonzeroes is the fourth, containing 3 unassigned nonzeroes. As explained above, we cut this column, increasing b_3 by one and N_C by 3. At this point, $m_0 - n_0$ is still smaller than $s_{c,0} - N_C$ ($2 < 3$), so we have to cut another column. The next column with the most unassigned nonzeroes is the fifth, with 2 unassigned nonzeroes. After cutting this column, $b_3 = 2$ and $N_C = 5$. Now, $m_0 - n_0 \geqslant s_{c,0} - N_C$ and we can stop cutting columns.

- $m_1 - n_1 < s_{c,1}$:
  $m_1 - n_1 = 4$ and $s_{c,1} = 0$, so we do not have to cut any columns.

- $m_0 - n_0 < s_{r,0}$:
  $m_0 - n_0 = 2$ and $s_{r,0} = 0$, so we do not have to cut any rows.

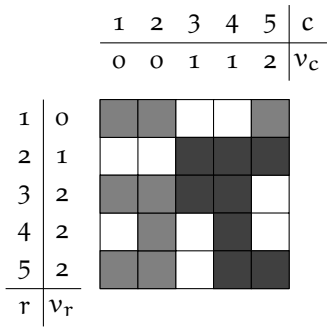
- $m_1 - n_1 < s_{r,1}$:
  $m_1 - n_1 = 4$ and $s_{r,1} = 1$, so we do not have to cut any rows.

We conclude that we have to cut two columns because of partially assigned columns to processor 0, and that $b_3 = 2$.

Combining all lower bounds, we can calculate a lower bound on the communication volume of the partial solution shown in Figure 5:

$$LB = b_1 + b_2 + b_3 = 1 + 2 + 2 = 5$$

This means that we have proven that any solution to the matrix partitioning problem of Figure 5 that can be obtained by extending the given partial assignment has a communication volume $\geqslant 5$. The optimal solution to the problem has a communication volume of 4, and is given in Figure 6. Depending on how the branching tree is traversed, we might be able to skip *all* partial solutions that are extended from the partial solution of Figure 5, leading to a considerable reduction of computation time.



5×5, $N = 16$, $m_0 = m_1 = 8$, $V = 4$

Figure 6.: Optimal solution to the matrix partitioning problem of Figure 5. This solution was found by the branch-and-bound algorithm described in this thesis. Note that the optimal communication volume is equal to 4, and that the lower bound of Figure 5 is equal to 5.

2.1.5 Implementation of the lower bounds

In the previous section, three different lower bounds were described. The following section addresses the implementation of these bounds in this master thesis. We will derive the computation time per partial solution of each lower bound. In order to make this derivation clearer, we define the quantity C_{max} as the number of nonzeroes that appear in the row or column with the most nonzeroes. In the worst case, the number of partial solutions that need to be examined is $O(3^{m+n})$ (see equation 2.2). For each partial solution, we can check its feasibility by iterating over all nonzeroes of the matrix, taking $O(N)$ time. Therefore, if the lower bounds take $O(f(N, m, n))$ time to be computed for each partial solution, the total *worst case* computation time of the exact algorithm will be $O(f(N, m, n)N3^{m+n})$. At first sight, this computation time looks worse than the trivial algorithm that visits all possible solutions, taking $O(N3^{m+n})$ time. Fortunately, the lower bounds enable us to skip large parts of the branch tree, lowering the number of partial solutions that need to be examined to a number $\ll 3^{m+n}$. This makes it worthwhile to spend extra computation time to calculate the different lower bounds.

The first lower bound that was described above can easily be implemented. Starting with $b_1 = 0$, each time a row or column is assigned a solution value of 2, we increment b_1 by one. If a row or column that had a solution value of 2 is unassigned (while traversing the branching tree in upward direction towards the root), we decrement b_1 by one. The computation time of these actions do not depend on the size of the problem, and is therefore $O(1)$.

For the second lower bound, we need to define some new variables that count the number of nonzeroes assigned to processor 0 and 1 in each row/column. Let $cn_i(j)$ be the number of nonzeroes assigned to processor i in column j, and $rn_i(j)$ similarly for the rows. For each unassigned column j, if $cn_0(j) > 0$ *and* $cn_1(j) > 0$, we know that that column is implicitly cut and it can be included in b_2. In order to maintain correct values for $cn_i(j)$ and $rn_i(j)$, we need to update these values for each row/column that is included in the column/row that is just assigned a solution value. For example, after assigning solution value 0 to row j, we need to increment the $cn_0(q)$ values of each column q that is included in row j. Since there are at most C_{max} nonzeroes in each column/row, maintaining the cn and rn lists costs $O(C_{max})$ time per partial solution. During the update of the list values, we can also check whether the updated columns are implicitly cut by checking if $cn_0(j) > 0$ and $cn_1(j) > 0$. Furthermore, we can decrement b_2 by one after assigning a solution

Lower bound	Theoretical basis	Computation time per partial solution
b_1	Cut r/c with value 2	$O(1)$
b_2	Implicitly cut r/c	$O(C_{max})$
b_3	Partially assigned r/c	$O(C_{max})$

Table 1.: Description of the lower bounds used in this report

value of 2 to a row/column that was implicitly cut, taking $O(1)$ time. Therefore, calculating b_2 takes $O(C_{max})$ time per partial solution.

As with its description, the implementation of b_3 is slightly more complicated to explain. The implementation of b_3 has to address several issues: first we need to obtain the values for $s_{c,0}$, $s_{c,1}$, $s_{r,0}$ and $s_{r,1}$. In order to find these values, we can use the $cn_i(j)$ and $rn_i(j)$ variables that were defined above. For example, if $cn_0(j) > 0$ and $cn_1(j) = 0$, we know that column j is partially assigned to processor 0, and we can add $(c_c(j) - cn_0(j))$ to $s_{c,0}$, where $c_c(j)$ is the number of nonzeroes in column j. Similarly, if $rn_1(j) > 0$ and $rn_0(j) = 0$, row j is partially assigned to processor 1, and we can add $(c_r(j) - rn_1(j))$ to $s_{r,1}$, with $c_r(j)$ being the number of nonzeroes in row j. We can maintain the correct values of the $s_{i,j}$ variables during the update of the $cn_i(j)$ and $rn_i(j)$ variables, and therefore, their calculation takes $O(C_{max})$ time per partial solution.

Next, we need to check whether $m_0 - n_0 < s_{c,0}$ along with the other variables, for which we need values for n_0 and n_1. When assigning column j to processor 0, we can increment n_0 by the number of unassigned nonzeroes in column j. Again, we can use the value for $cn_0(j)$ in order to do this. Since we already calculated $cn_0(j)$ for b_2, we can maintain the correct values for n_0 and n_1 in $O(1)$ time per partial solution.

Finally, when $m_0 - n_0 < s_{c,0}$, we need to cut the columns with the most unassigned nonzeroes repeatedly, until $m_0 - n_0 \geqslant s_{c,0} - N_C$. To know which column has the most unassigned nonzeroes, we need a list of all columns, sorted on the number of unassigned nonzeroes in them. Since there are at most C_{max} nonzeroes in a single column, we know that the range of possible values is $[0, C_{max}]$, and therefore we can use a very fast sorting algorithm to sort them: *Counting Sort* (see, for example, [10]), attributed to Harold H. Seward (1959). The counting sort relies on preallocating $C_{max} + 1$ bins, each corresponding to an integer in $[0, C_{max}]$, and placing each item to be sorted in the correct bin. When the sorted list is needed, we can simply list all items in the bins starting with the bin corresponding to C_{max} and ending with the one corresponding to 0. Also, we do not need to know *which* column we have to cut, only how many unassigned nonzeroes are contained in that column, which is the amount by which we increment N_C. This makes all insertion and deletion operations in the sorted list take $O(1)$ time, while obtaining the sorted list takes $O(C_{max})$ time to iterate over all bins. To calculate b_3, we can simply cut columns starting with the bin corresponding to C_{max} and ending with the one corresponding to 0, or until $m_0 - n_0 \geqslant s_{c,0} - N_C$. As explained above, all operations in this calculation can be done in $O(C_{max})$ time.

The worst-case computation time of the implementations of all lower bounds used in this thesis is given in Table 1. Now that we have a branch-and-bound method (Section 2.1.3) combined with several lower bounds on partial solutions (Section 2.1.4) that can be computed efficiently (Section 2.1.5), we are in possession of a powerful method that can find optimal solutions to the matrix partitioning problem. Section 2.3 discusses results that were obtained using this method, and up to what matrix sizes the method remains usable. The next section focusses on further lowering the computation time by parallelizing the algorithm.

2.1.6 Parallelizing the algorithm

In the previous sections of this chapter, an algorithm for solving the matrix partitioning problem was developed. This branch and bound algorithm is *sequential* in nature: it is designed to be executed by one processing unit. Since matrix partitioning is NP-hard, the time to execute the algorithm can grow very large, even for relatively small instances. In order to enable us to solve larger problems, it is therefore necessary to *parallelize* the algorithm: redesign it so it can be executed efficiently by several processing units working in parallel. Using p processors to execute an algorithm in parallel, in theory it can be possible to speed up the sequential execution time T_{seq} by a factor p:

$$T(p) = \frac{T_{seq}}{p} \tag{2.3}$$

In practice however, the execution time is often larger than that of Equation 2.3, because of several effects[1]. First, the process of setting up and maintaining the p processing units can induce a time penalty on the system that the sequential algorithm does not suffer from. More importantly, in most practical parallel algorithms, the different processing units need to communicate between each other. Normally, the overhead involved in these communications is large: for example, communication between processors often travels slower than a single processor can access its local memory. Finally, Equation 2.3 assumes that we are able to distribute the total work of the sequential algorithm evenly over the p processors, which is often very difficult in practice. When developing parallel algorithms, it is therefore important to balance the work evenly while keeping the communication costs low.

In the specific case of branch-and-bound algorithms, each subproblem is an independent computation unit. Therefore, a very natural parallelization of branch-and-bound algorithms exists: simply distribute all subproblems over the available processors. Because of the independence of each subproblem, no communication between processors is necessary, seemingly making Equation 2.3 a reality. However, since the subproblems are connected to each other in a branching tree, we are only able to distribute parts of the tree over the processors, instead of individual nodes of the tree. Although this is not problematic by itself, the actual *shape* of the branch tree is not known beforehand. This makes it hard to evenly distribute the subproblems over the different processors, thereby creating an imbalanced work distribution. Take, for example, the small branching tree of Figure 7. If we would distribute the leftmost part of the tree to processor 0, the middle part to processor 1 and the rightmost part to processor 2, a large work imbalance is created: the processors have to examine 4, 13 and 7 nodes respectively. Since there are 24 nodes in total, in the optimal distribution each processor examines 8 nodes. Therefore, in the naive distribution of Figure 7, processor 1 has to perform 62.5% more work than optimal. However, although computational load balance suffers, the naive distribution is relatively easy to implement and incurs no communication between processors, making it a possible parallelization of the branch-and-bound method.

A better load balance can be achieved by implementing a *dynamic* distribution scheme, where the subproblems are dynamically distributed to the different processors, depending on information on the shape of the branching tree. Several types of distribution schemes are explained in [35], and in this thesis, a relatively simple approach is taken. In principle, this simple dynamic distribution scheme works as follows. At the start of the algorithm, a pool of N_{pool} subproblems is created sequentially by *every* processor. This pool is created in a breadth-first manner and in such a way that each processor has the same pool. Each processor then picks a different subproblem from the pool, and starts traversing the subtree of the branching tree rooted at that subproblem. This traversal

1 In fact, in some rare cases the execution time can be *smaller* than that of Equation 2.3. The main reason for this apparent paradox is the cache design of modern processors. If the parallel algorithm is able to divide the needed data into small enough pieces, it might fit in a lower cache level of each processor, making it possible to process the needed work faster than the sequential algorithm. This phenomenon is called *superlinear speedup*.

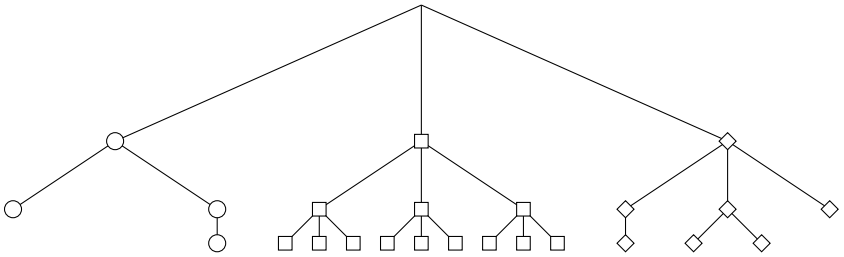


Figure 7.: A branching-tree distributed over 3 processors: circle, square and diamond. Because of the irregular shape of the tree, the branch in the middle consists of the most vertices, and the processor responsible for this part of the tree has to perform more work than the others.

is performed in exactly the same way that the sequential branch-and-bound algorithm works. Each processor marks all subproblems that were picked as *assigned*, meaning that they do not have to be checked again. After each N_{sync} steps, all processors synchronize and send each other a certain *state* variable z_i, indicating its current state. Note that this incurs a communication cost on the parallel algorithm. There are several possibilities for the state variable z_i:

- $z_i = -1$: processor i is still traversing its subtree, and has not found a new feasible solution with a lower cost than the current upper bound.
- $z_i = -2$: processor i has finished traversing its subtree, and has not found a new feasible solution with a lower cost than the current upper bound.
- $z_i = a$, with $a \in \{0, 1, \ldots, m+n-1\}$: processor i is still traversing its subtree, and has found a new feasible solution with cost a.
- $z_i = b$, with $b \in \{m+n, m+n+1, \ldots, 2(m+n)-1\}$: processor i has finished traversing its subtree, and has found a new feasible solution with cost $b-(m+n)$.

After synchronization, each processor owns a list of p state variables, one for each processor in the parallel algorithm. We check the value of each state variable, and perform an action depending on its value, where s is the local processor index:

- $z_i = -1$: no action is required.
- $z_i = -2$: if $s = i$, take the next unassigned subproblem from the pool, and start traversing the subtree rooted at this subproblem after all state variables have been checked. If $s \neq i$, mark the next unassigned subproblem from the pool as assigned, as it is just picked up by processor i.
- $z_i = a$: if $a < U$ (where U is the current upper bound), set U equal to a.
- $z_i = b$: if $s = i$, take the next unassigned subproblem from the pool, and start traversing the subtree rooted at this subproblem after all state variables have been checked. If $s \neq i$, mark the next unassigned subproblem from the pool as assigned, as it is just picked up by processor i. Furthermore, if $b-(m+n) < U$, set U equal to $b-(m+n)$.

At some point, all subproblems in the pool become assigned, at which point the processors that are finished with their subtree become idle. When all processors become idle, the entire branching tree has been traversed, and the parallel algorithm is finished, with the optimal solution having a

communication volume of U. During each synchronization, each processor needs to send 1 integer to all $(p-1)$ processors, making the communication cost of each synchronization $O(p)$.

The advantage of this dynamic distribution scheme is that processors that get assigned a small subtree and become idle after a few subproblems will now get a new subtree assigned to them after at most N_{sync} steps. In the naive algorithm, these processors would be idle for the rest of the computation. In order to work correctly, N_{pool} has to be chosen large enough, such that there are enough small subtrees to distribute over the processors. However, N_{pool} should not be too large as well, since in that case, the sequential pool creation part will take more time than the parallel part of the algorithm.

In the optimal distributed case, the maximum number of subproblems that any single processors needs to consider, N_{max}, will be equal to the total number of subproblems divided by the number of processors:

$$N_{max}^{OPT} = \frac{N_{tree}}{p}$$

If N_{max} is much larger than N_{max}^{OPT}, the distribution scheme is not distributing the subproblems evenly. In order to quantify this, we define:

$$\eta = \frac{N_{max}}{N_{max}^{OPT}}$$

The closer η is to 1, the better the subproblems are distributed.

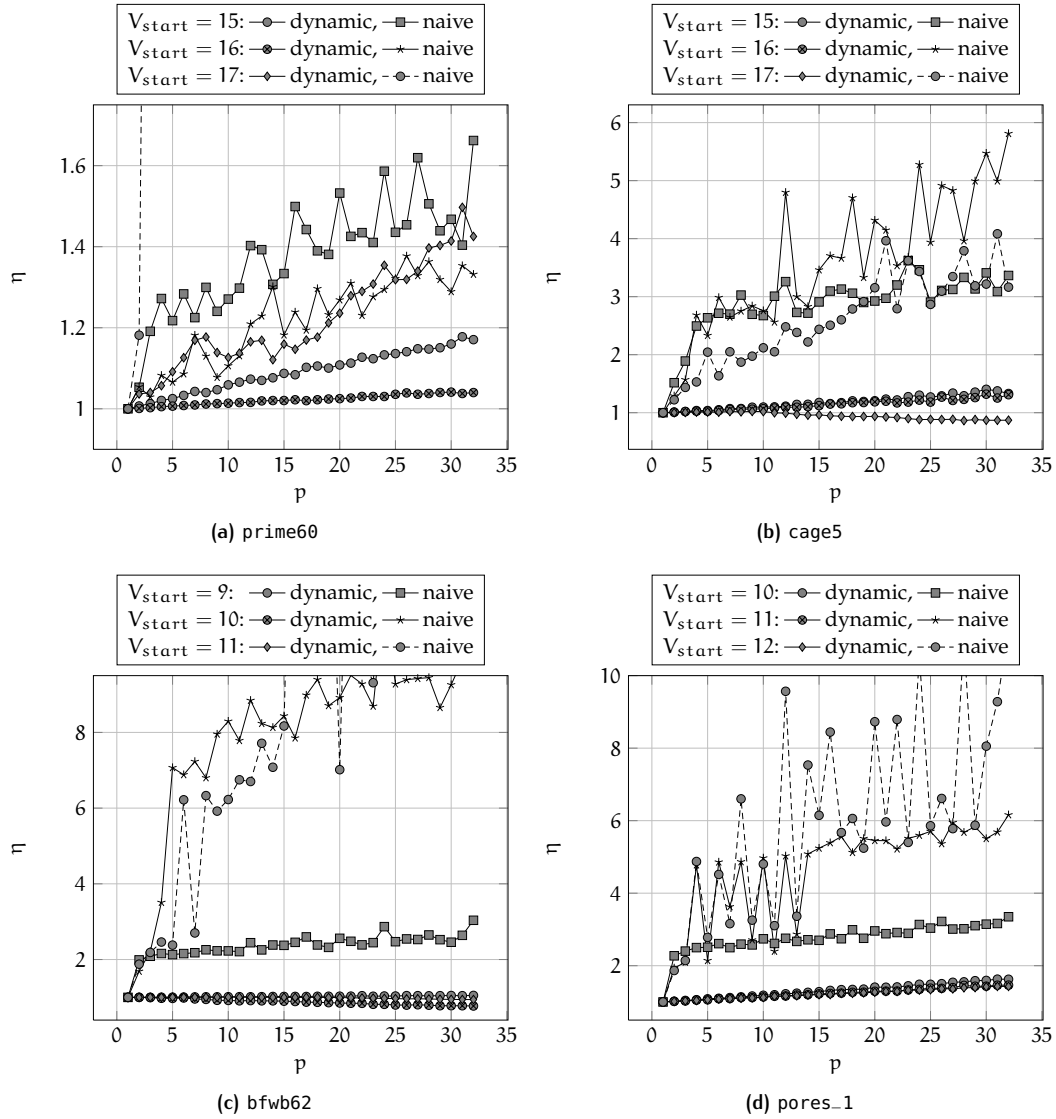
The parallel algorithms used in this report were implemented using MPI-1 [20], using the Bulk Synchronous Parallel (BSP) programming model ([39]). The parallel code itself was written in MPI-1 and compiled by the GNU Compiler Collection (GCC) version 4.5.0 [2] and OpenMPI version 1.4.1 [17]. All measurements were performed on an Intel Core i5 processor with 4 processor cores. In Figure 8, η values are given for both the naive and the dynamic distribution scheme described above, for different matrices and starting upper bounds. As is clear from the Figure, the naive distribution has a large η value even for small values of p. The dynamic distribution scheme is much better at achieving a good computational load balance. It is also important to note that the behaviour of the dynamic scheme is much more predictable than that of the naive scheme: for different matrices or starting upper bound, the values for η are very different for the naive scheme. Because of the better load balance and its predictable behaviour, the dynamic distribution scheme is used in this thesis.

An interesting case is the dynamic scheme for `cage5` with $V_{start} = 17$ and `bfwb62` with $V_{start} = 10$ and $V_{start} = 11$. For larger p, η becomes *smaller* than 1, indicating superlinear speedup. Apparently, using multiple processors enables the algorithm to find a good feasible solution in shorter time. Since the current upper bounds are communicated to the other processors in the parallel algorithm, the cost of this feasible solution can be used as the new upper bound to reduce the number of subproblems that need to be traversed. Therefore it is possible that the total number of subproblems is smaller than the sequential algorithm. Basically, the branching-tree is traversed in a different order by the parallel algorithm than by the sequential algorithm, and in some cases this can be very beneficial.

To show the actual computational speedup that is possible with the parallel algorithm, Figure 9 gives the measured computation time of the parallel algorithm compared to the sequential algorithm. Shown is the measured parallel *speedup*, defined as:

$$speedup(p) = \frac{T_{seq}}{T_p} \tag{2.4}$$

In this equation, T_{seq} is the computation time of the sequential algorithm, and T_p is the computation time of the parallel algorithm using p processing units. For a good parallelization, the speedup is



(a) `prime60` **(b)** `cage5`

(c) `bfwb62` **(d)** `pores_1`

Figure 8.: Computational load balance of the naive and dynamic distribution scheme, for different matrices and starting upper bound V_{start}.

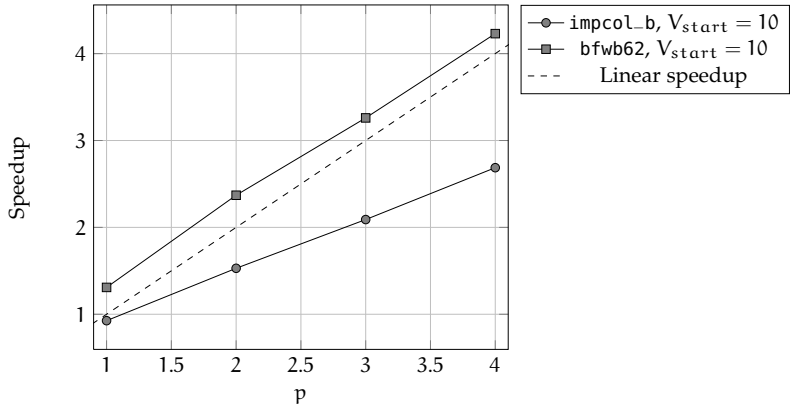


Figure 9.: Measurements of the parallel speedup achieved by the dynamic parallel branch-and-bound algorithm, for `impcol_b`, `bfwb62` and $V_{start} = 10$. Note that superlinear speedup is achieved for `bfwb62`.

linear: $speedup(p) \approx p$. The results from Figure 9 show that for $p \leqslant 4$, the parallelization of the branch-and-bound method performs very well.

2.2 INTEGER LINEAR PROGRAMMING

As an alternative to the branch and bound method described above, we can also formulate the matrix partitioning problem as an *integer linear program*, and use commercial optimizing software to solve it. A linear program is an optimization problem, that can be generally described as:

$$\begin{aligned} \text{Minimize:} \quad & \vec{c} \cdot \vec{x} \\ \text{Subject to:} \quad & A \cdot \vec{x} \leqslant \vec{b} \\ & x \geqslant 0 \qquad \forall \quad x \in \vec{x} \end{aligned} \tag{2.5}$$

The values of the input variables $\vec{c}$, $\vec{b}$ and A define which problem is modelled by a linear program. It turns out that linear programs are very flexible as a model: almost all naturally arising combinatorial problems that can be solved in polynomial time can be written as a linear program. Various efficient algorithms that solve linear programs have been developed: in 1947, the popular simplex method was described first by Dantzig [11] and later, in 1984, the barrier method was invented [26]. The existence of these methods has as result that if a problem can be written as a linear program, it can be solved efficiently (e.g. in polynomial time) in practice.

As was clear from the results of the branch and bound algorithm, an efficient algorithm for matrix partitioning does not exist. Therefore, we do not expect that we are able to model the matrix partitioning problem as a linear program. What is possible however, is to write it as a closely related model: *integer linear programming*. An integer linear program is the same as a linear program, with one crucial difference: the variables $\vec{x}$ are restricted to the set of integers $\geqslant 0$. As such, a general integer linear program can be written as:

$$\begin{aligned} \text{Minimize:} \quad & \vec{c} \cdot \vec{x} \\ \text{Subject to:} \quad & A \cdot \vec{x} \leqslant \vec{b} \\ & x \in \mathbb{N} \qquad \forall \quad x \in \vec{x} \end{aligned} \tag{2.6}$$

This small change in definition has large implications: it is shown that solving a general integer linear program is NP-Complete [32], and no efficient algorithms for solving integer linear programs exist. However, its power as a modelling tool remains: almost all known NP-Complete problems can be written as an integer linear program. Because a large number of important problems (such as the TSP [4] and vehicle routing [36]) can be modelled by integer linear programs, a lot of effort has been put into solving them as efficiently as possible. Several breakthroughs such as *cutting plane* techniques [19] and branch-and-cut methods [21] have made it possible to solve moderately sized instances of integer linear programs. Various commercial software packages are available that combine different solving methods in order to allow larger problems to be solved. One of the most popular of these packages is CPLEX [1], which is used in this thesis.

The extend to which a certain problem can be solved using integer linear programs highly depends on the actual formulation of the program. It is often possible to model a certain problem by very different integer linear programs, with some of them easier to solve than others. In this thesis, three different programs for matrix partitioning over two parts are developed, and their execution time is compared to both each other *and* the branch and bound method which was described above.

2.2.1 Model 1: A simple program

For each nonzero i of the $m \times n$ matrix A, define a variable x_i that can take two values: 0 and 1. If x_i is 0, nonzero i is assigned to processor 0, and if x_i is 1, it is assigned to processor 1. Using this definition, it is easy to specify the strict load imbalance equation.

$$\sum_{i=0}^{N-1} x_i = m_1 \tag{2.7}$$

A correct formulation of the cost of a partitioning is a bit more complicated. For every row i, define variables r_i^0 and r_i^1, and for every column i, define c_i^0 and c_i^1. Now we want to add constraints, such that $(r_i^0 - r_i^1)$ is equal to 1 if row i is cut, and equal to 0 if it is not cut, and similarly for columns. If this is the case, the communication volume of a partitioning, which is equal to the number of cut rows and columns if $k = 2$, can be written as:

$$V = \sum_{i=0}^{m-1} \left(c_i^0 - c_i^1\right) + \sum_{i=0}^{n-1} \left(r_i^0 - r_i^1\right) \tag{2.8}$$

The question remains which constraints have to be added to ensure the above values for $(c_i^0 - c_i^1)$ and $(r_i^0 - r_i^1)$. For r_i^0, we propose:

$$r_i^0 \geqslant \frac{\sum_{j \in \text{row } i} x_j}{n_r(i)}$$

If all nonzeroes of row i are assigned to processor 0, the right hand side of the above equation is zero, and r_i^0 is allowed to become 0. Note that since we are minimizing r_i^0, even though r_i^0 can also be 1 in this case, setting it to 0 will always lead to a lower cost solution, and will therefore be chosen. If one or more nonzeroes of row i are assigned to processor 1, the right hand side will be equal to a number $\in (0, 1]$, and r_i^0 *has* to be equal to 1.

Let's define r_i^1 similar to r_i^0, with $\geqslant$ replaced by $\leqslant$:

$$r_i^1 \leqslant \frac{\sum_{j \in \text{row } i} x_j}{n_r(i)}$$

In this case, r_i^1 can only be set to 1 if all nonzeroes in row i are assigned to processor 1. If one or more nonzeroes are assigned to processor 0, r_i^1 *has* to be set to 0.

Combining both definitions, we see that:

$$r_i^0 - r_i^1 = \begin{cases} 0-0=0 & \text{if all nonzeroes of row } i \text{ are assigned to processor 0} \\ 1-1=0 & \text{if all nonzeroes of row } i \text{ are assigned to processor 1} \\ 1-0=1 & \text{otherwise} \end{cases} \tag{2.9}$$

The above equation exactly describes the needed behaviour: $(r_i^0 - r_i^1)$ is equal to 1 if row i is cut, and equal to 0 if it is not cut!

A similar definition of the column variables leads to the following integer linear program:

$$\begin{array}{rll} \text{Minimize:} & \sum_i (c_i^0 - c_i^1) + \sum_i (r_i^0 - r_i^1) & \\ \text{Subject to:} & \sum_i x_i = m_1 & \\ & n_r(i) \cdot r_i^0 - \sum_{j \in \text{row } i} x_j \geqslant 0 & \forall \quad i \in \{0,1,2,\ldots,n-1\} \\ & n_r(i) \cdot r_i^1 - \sum_{j \in \text{row } i} x_j \leqslant 0 & \forall \quad i \in \{0,1,2,\ldots,n-1\} \\ & n_c(i) \cdot c_i^0 - \sum_{j \in \text{col } i} x_j \geqslant 0 & \forall \quad i \in \{0,1,2,\ldots,m-1\} \\ & n_c(i) \cdot c_i^1 - \sum_{j \in \text{col } i} x_j \leqslant 0 & \forall \quad i \in \{0,1,2,\ldots,m-1\} \\ & x_i \in \{0,1\} & \forall \quad i \in \{0,1,2,\ldots,N-1\} \\ & r_i^0 \in \{0,1\} & \forall \quad i \in \{0,1,2,\ldots,n-1\} \\ & r_i^1 \in \{0,1\} & \forall \quad i \in \{0,1,2,\ldots,n-1\} \\ & c_i^0 \in \{0,1\} & \forall \quad i \in \{0,1,2,\ldots,m-1\} \\ & c_i^1 \in \{0,1\} & \forall \quad i \in \{0,1,2,\ldots,m-1\} \end{array} \tag{2.10}$$

Solving this program is equivalent to solving the matrix partitioning problem for $k = 2$. A solution to the above linear program includes a vector $\vec{x}$ with element $\in \{0,1\}$. This vector states to which processor each nonzero of the matrix is assigned in the optimal solution to the corresponding matrix partitioning problem. Excluding the constraints that specify that each variable is $\in \{0,1\}$, the integer linear program consists of $N + 2(m+n)$ variables, and $2(m+n)+1$ constraints.

2.2.2 Model 2: A larger model

For this integer linear program, a solution method more similar to the branch-and-bound method is attempted. To this end, we define the same solution values as in the branch and bound method:

$$r_{0i} = \begin{cases} 1 & \text{if row } i \text{ is assigned to processor 0} \\ 0 & \text{otherwise} \end{cases}$$

$$r_{1i} = \begin{cases} 1 & \text{if row } i \text{ is assigned to processor 1} \\ 0 & \text{otherwise} \end{cases}$$

$$r_{2i} = \begin{cases} 1 & \text{if row } i \text{ is cut} \\ 0 & \text{otherwise} \end{cases}$$

$$c_{0i} = \begin{cases} 1 & \text{if column } i \text{ is assigned to processor 0} \\ 0 & \text{otherwise} \end{cases}$$

$$c_{1i} = \begin{cases} 1 & \text{if column } i \text{ is assigned to processor 1} \\ 0 & \text{otherwise} \end{cases}$$

$$c_{2i} = \begin{cases} 1 & \text{if column } i \text{ is cut} \\ 0 & \text{otherwise} \end{cases}$$

Each row and column can only be in one of the three states, and therefore, the following constraints are added:

$$\begin{aligned} c_{0i} + c_{1i} + c_{2i} &= 1 \qquad \forall \quad i \in 0, 1, 2, \ldots, m-1 \\ r_{0i} + r_{1i} + r_{2i} &= 1 \qquad \forall \quad i \in 0, 1, 2, \ldots, n-1 \end{aligned} \tag{2.11}$$

We can use these constraints to eliminate two variables of the program. For instance, we can define:

$$\begin{aligned} c_{0i} &= 1 - c_{2i} - c_{1i} \\ r_{0i} &= 1 - r_{2i} - r_{1i} \end{aligned}$$

The cost of a certain partitioning is equal to the number of cut rows, and therefore easily specified:

$$V = \sum_{i=0}^{m-1} c_{2i} + \sum_{i=0}^{n-1} r_{2i} \tag{2.12}$$

For each nonzero a_{ij}, it is impossible that its row is assigned to processor 0 and its column is assigned to processor 1, or the other way around. In order to prevent this in the integer linear program, two constraints are added for each nonzero a_{ij}:

$$\begin{aligned} c_{1i} + r_{0j} &= 1 \qquad \forall \quad a_{ij} \\ c_{0i} + r_{1j} &= 1 \qquad \forall \quad a_{ij} \end{aligned}$$

In this integer linear program, the strict load imbalance equation is the complicated equation to include. In order to include it, we need to define a new binary variable for each nonzero a_{ij}:

$$x_{ij} = \begin{cases} 0 & \text{if nonzero } a_{ij} \text{ is assigned to processor 0} \\ 1 & \text{otherwise} \end{cases} \tag{2.13}$$

Given this definition, it is easy to see that we can define the strict load imbalance equation as:

$$\sum_{a_{ij} \in A} x_{ij} = m_1 \tag{2.14}$$

To define constraints that ensure the needed behaviour for x_{ij}, note that nonzero a_{ij} is allowed to be assigned to processor 1 if either its row or column value is equal to 1, *or* both its row and column value is equal to 2. To ensure the first possibility, we can state:

$$\begin{aligned} x_{ij} &\leqslant c_{1i} + r_{1j} \\ x_{ij} &\geqslant c_{1i} \\ x_{ij} &\geqslant r_{1j} \end{aligned}$$

The first equation states that if both c_{1i} and r_{1j} are equal to 0, x_{ij} has to be equal to 0 as well. The other two equations state that if either c_{1i} or r_{1j} is equal to 1, x_{ij} has to be equal to one as well.

We can ensure the second possibility (row and column value both equal to 2) by stating:

$$x_{ij} \leqslant \frac{1}{2}\left(c_{2i} + r_{2j}\right)$$

In this case, only if both c_{2i} and r_{2j} are equal to 1, x_{ij} can be set to 1. Combining both possibilities, we get the following constraints:

$$\begin{aligned} x_{ij} &\leqslant c_{1i} + r_{1j} + \frac{1}{2}\left(c_{2i} + r_{2j}\right) \\ x_{ij} &\geqslant c_{1i} \\ x_{ij} &\geqslant r_{1j} \end{aligned}$$

As explained above, these constraints are enough to ensure the behaviour described in Equation 2.13.

Combining everything, we get the following integer linear program:

$$\begin{array}{rll} \text{Minimize:} & \sum_i c_{2i} + \sum_i r_{2i} & \\ \text{Subject to:} & \sum_{a_{ij} \in A} x_{ij} = m_1 & \\ & c_{1i} - r_{1j} - r_{2j} = 0 & \forall \quad a_{ij} \in A \\ & r_{1j} - c_{1i} - c_{2i} = 0 & \forall \quad a_{ij} \in A \\ & 2(x_{ij} - c_{1i} - r_{1j}) - c_{2i} - r_{2j} \leqslant 0 & \forall \quad a_{ij} \in A \\ & x_{ij} - c_{1i} \geqslant 0 & \forall \quad a_{ij} \in A \\ & x_{ij} - r_{1j} \geqslant 0 & \forall \quad a_{ij} \in A \\ & c_{1i} \in \{0,1\} & \forall \quad i \in \{0,1,2,\ldots,m-1\} \\ & c_{2i} \in \{0,1\} & \forall \quad i \in \{0,1,2,\ldots,m-1\} \\ & r_{1i} \in \{0,1\} & \forall \quad i \in \{0,1,2,\ldots,n-1\} \\ & r_{2i} \in \{0,1\} & \forall \quad i \in \{0,1,2,\ldots,n-1\} \\ & x_{ij} \in \{0,1\} & \forall \quad a_{ij} \in A \end{array} \tag{2.15}$$

This program solves the matrix partitioning problem using $N + 2(m+n)$ variables and $5N + 1$ constraints.

2.2.3 Model 3: A hypergraph model

As the hypergraph partitioning problem has been studied before in the past, several integer linear programs that model hypergraph partitioning have been proposed. Of course, we can use these

programs to solve matrix partitioning as well, by applying the finegrain hypergraph model. In this thesis, we will use the hypergraph integer linear program from [5], with slight modifications such as addition of the strict load imbalance equation. The program can be described as follows, with n_m being the number of vertices of the hypergraph, n_n the number of nets and n_b the number of subsets we want to partition in. Since we partition in two sets, $n_b = 2$ for the rest of this section. Note that in this section a different notational style is used than in the rest of the report, in order to maintain compatibility with [5].

Define:

$$x_{ik} = \begin{cases} 1 & \text{if vertex } i \text{ is placed in subset } k \\ 0 & \text{otherwise} \end{cases}$$

$$y_{jk} = \begin{cases} 1 & \text{if net } j \text{ is placed entirely in subset } k \\ 0 & \text{otherwise} \end{cases}$$

Since we want to minimize the number of cut nets, we can also choose to maximize the number of uncut nets:

$$\max \sum_{j=1}^{n_n} y_{j0} + y_{j1}$$

First we have to state the *vertex placement* constraints that each vertex should be assigned to one of the subsets:

$$x_{i0} + x_{i1} = 1 \qquad \forall \quad i \in \{1, 2, \ldots, n_m\}$$

Next, we can add the load imbalance constraint for one of the subsets:

$$\sum_{i=1}^{n_m} x_{i1} = m_1$$

Finally, hyperedge constraints are added to ensure that y_{jk} can only be 1 if all vertices in hyperedge j are assigned to block k:

$$\begin{aligned} y_{j0} &\leqslant x_{i0} \qquad 1 \leqslant j \leqslant n_n, 1 \leqslant i \leqslant n_m, i \in \text{hyperedge } j \\ y_{j1} &\leqslant x_{i1} \qquad 1 \leqslant j \leqslant n_n, 1 \leqslant i \leqslant n_m, i \in \text{hyperedge } j \end{aligned}$$

Combining the above, we get the following integer linear program:

$$\begin{aligned} \text{Maximize:} \quad & \textstyle\sum_{j=1}^{n_n} y_{j0} + y_{j1} \\ \text{Subject to:} \quad & \textstyle\sum_{i=1}^{n_m} x_{i1} = m_1 \\ & x_{i0} + x_{i1} = 1 \qquad \forall \quad i \in \{1, 2, \ldots, n_m\} \\ & y_{j0} \leqslant x_{i0} \qquad 1 \leqslant j \leqslant n_n, 1 \leqslant i \leqslant n_m, i \in \text{hyperedge } j \\ & y_{j1} \leqslant x_{i1} \qquad 1 \leqslant j \leqslant n_n, 1 \leqslant i \leqslant n_m, i \in \text{hyperedge } j \\ & x_{ik} \in \{0, 1\} \qquad 1 \leqslant i \leqslant n_m, 1 \leqslant k \leqslant n_b \\ & y_{jk} \in \{0, 1\} \qquad 1 \leqslant j \leqslant n_n, 1 \leqslant k \leqslant n_b \end{aligned} \tag{2.16}$$

In the finegrain model of matrix partitioning, there are N vertices and $m + n$ nets. In this case, $n_m = N$ and $n_n = m + n$. Since $n_b = 2$, the above program consists of $2(N + m + n)$ variables and $5N + 1$ constraints. In order to see this, note that there are $2N$ x_{ik} variables and $2(m + n)$ y_{ik} variables. Furthermore, each of the x_{ik} variables appears in two different constraints (one for

its column hyperedge and one for its row hyperedge), leading to 4N constraints. There are also N vertex placement constraints (one for each vertex), and an additional strict load imbalance constraint.

However, when using this program to solve matrix partitioning over two parts, we can use the vertex placement constraints to eliminate one of the x_{ik} variables for each nonzero i. Rearranging the vertex placement constraints, we get:

$$x_{i0} = 1 - x_{i1} \qquad \forall \quad i \in \{1, 2, \ldots, n_m\}$$

Replacing x_{i0} with $1 - x_{i1}$ in the above program, we get:

$$\begin{array}{rll} \text{Maximize:} & \sum_{j=1}^{n_n} y_{j0} + y_{j1} & \\ \text{Subject to:} & \sum_{i=1}^{n_m} x_{i1} = m_1 & \\ & y_{j0} \leqslant 1 - x_{i1} & 1 \leqslant j \leqslant n_n, 1 \leqslant i \leqslant n_m, i \in \text{hyperedge } j \\ & y_{j1} \leqslant x_{i1} & 1 \leqslant j \leqslant n_n, 1 \leqslant i \leqslant n_m, i \in \text{hyperedge } j \\ & x_{i1} \in \{0, 1\} & 1 \leqslant i \leqslant n_m \\ & y_{jk} \in \{0, 1\} & 1 \leqslant j \leqslant n_n, 1 \leqslant k \leqslant n_b \end{array} \tag{2.17}$$

Since we have eliminated N variables (the x_{i0}'s) and N constraints (the vertex placement constraints), the final program consists of $N + 2(m + n)$ variables and $4N + 1$ constraints.

2.2.4 Results

We will now compare the three integer linear program with each other and with the earlier sequential branch-and-bound method. Since it is proven that all methods provide the optimal two-way partitioning of a matrix, the communication volumes that are found by the different methods are equal. More interesting is the question which of the methods is the most time efficient in finding the optimal solution. In order to test this, the integer linear programs were solved by IBM ILOG CPLEX 12.1 [1], using one processing core of an Intel Core Atom N270 processor. The times given in Table 2 are the execution times that were reported by CPLEX. To make the comparison fair, the branch-and-bound method was also executed using one core of the same processor. Implemented in C, the branch-and-bound method was compiled using the Gnu Compiler Collection version 4.5, using the compiler switches `-march=native` and `-O3`.

Since the time it takes to heuristically find feasible upper bounds is included in the execution time of CPLEX, a similar approach is taken with the branch-and-bound method. Before starting branching and bounding, we first run the Mondriaan software package, version 2.01 with the `hybrid` setting, ten times and return the lowest volume that was found. The output of this heuristic is an upper bound on the communication volume of the optimal solution. This upper bound is then used as the starting upper bound in the branch-and-bound method. The average total time (including the heuristic and the branch-and-bound method) of a single run is reported.

The execution times of all methods are given in Table 2 for a selection of small sparse matrices. Note that for all but one of the matrices, the branch and bound method performs much better than the Integer Linear Programs. In the case of the `prime60` matrix, the branch and bound method was almost 150 times faster than the best Integer Linear Program. Although the integer linear programs that were used are relatively simple, these results show that developing a problem-specific branch and bound method for solving the matrix partitioning problem can be worthwhile. Furthermore, of the three ILP models that were tested in this thesis, the hypergraph model showed the best results, being the fastest in most cases. Further research on ILP models for the matrix partitioning problem can focus on extending the hypergraph model in order to enable it to solve larger matrices.

An interesting case is the `impcol_b` matrix, where the ILPs actually outperformed the branch and bound method by a large amount. In the authors view, this presents a challenge and a possible new

	Computation time							
	Branch and Bound		Integer Linear Program					
			Model 1		Model 2		Model 3	
Matrix	**Abs** (s)	**Rel**	**Abs** (s)	**Rel**	**Abs** (s)	**Rel**	**Abs** (s)	**Rel**
cage4	0.08	1	2.77	33.78	0.63	7.68	0.66	8.05
cage5	1.76	1	251.47	143.1	369.86	210.5	100.48	57.19
bfwb62	4.34	1	95.56	22.02	63.50	14.63	41.06	9.46
Hamrle1	0.25	1	1.50	6.07	1.83	7.41	0.53	2.15
impcol_b	186.6	1	45.22	0.24	9.61	0.05	4.31	0.02
pores_1	0.75	1	26.13	35.12	6.13	8.24	10.09	13.56
prime60	3.07	1	1240.48	404.1	1252.91	408.1	449.16	146.3

Table 2.: Comparison of the computation time of all exact algorithms that were used in this thesis. Results are given in seconds (**Abs**), and in time relative to the sequential branch and bound method (**Rel**).

direction for further research on the branch and bound method: designing new lower bounds for partial solutions that benefit finding the optimal solution to the impcol_b matrix, possibly lowering the computation time for other matrices as well.

2.3 BENCHMARK RESULTS

For all matrices of the University of Florida Sparse matrix collection [12] with $m \leqslant 200$ and $n \leqslant 200$, 199 in total, the communication volume of its optimal solution ($\varepsilon = 0.03$) is given in Appendix B. The method that was used to find these optimal solutions is the sequential branch and bound method that is described in Section 2.1. For some matrices, computation time of the optimal solution exceeded 1 day, at which point the algorithm was terminated. In these cases, the best known communication volume is given, along with an indication that it may not be optimal.

As shown in Figure 10, more than 75% of the 199 matrices have been solved optimally. The largest matrix that was solved in terms of number of rows and columns is bwm200, a 200×200 matrix with 796 nonzeroes and an optimal communication volume of 4. The optimally solved matrix with the most nonzeroes was bcsstk04, a 132×132 matrix with 3648 nonzeroes and an optimal communication volume of 48. In addition, bcsstk04 has the largest optimal communication volume of all solved matrices. The matrices with the largest fraction of cut rows and columns are cage9 and Stranke94, with exactly half of all rows and columns cut in the optimal solution.

In Figures 11 through 19, some example results are given for which the Mondriaan 3 software package performed suboptimal. In these Figures, the optimal communication volume and matrix partitioning are given, along with a histogram and average value of the communication volumes found by 1000 runs of Mondriaan, using the hybrid setting.

For the GD98_a matrix, Mondriaan is unable to find the optimal solution with a communication volume of 0, finding solutions with $V = 7$ instead. Even worse is the GD06_theory matrix, which also has an optimal volume of 0, while Mondriaan produces solutions that are much worse. From a visual inspection of the optimal solution, shown in Figure 16, it is clear that a solution with no communication is possible, and it is interesting to find the reason of Mondriaan's suboptimal behaviour. As such, these examples are not intended to criticise Mondriaan and other partitioners, but rather given as a means to improve these partitioners beyond their current capabilities.

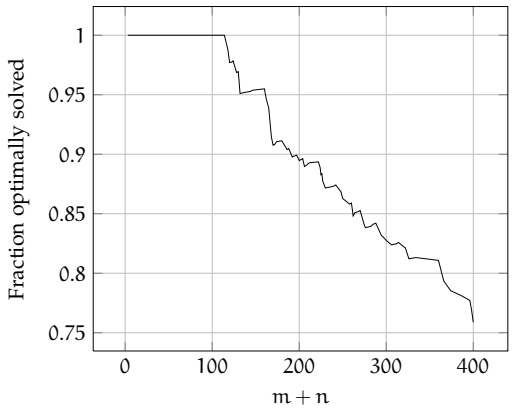
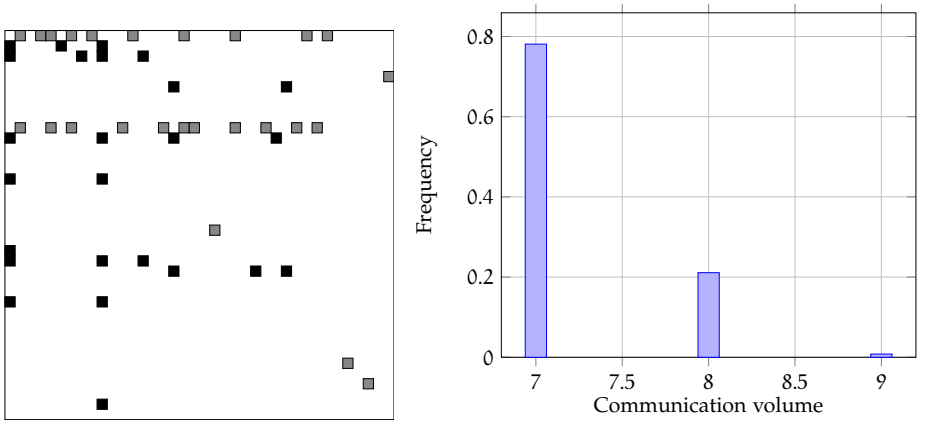


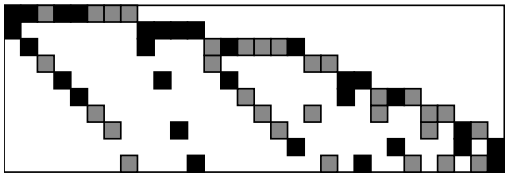
Figure 10.: Fraction of matrices from [12] with $m \leqslant 200$ and $n \leqslant 200$ that were optimally solvable in less than one day using the sequential branch and bound algorithm. Given is the fraction of matrices with $m + n$ smaller than or equal to the x-axis value that were optimally solvable.



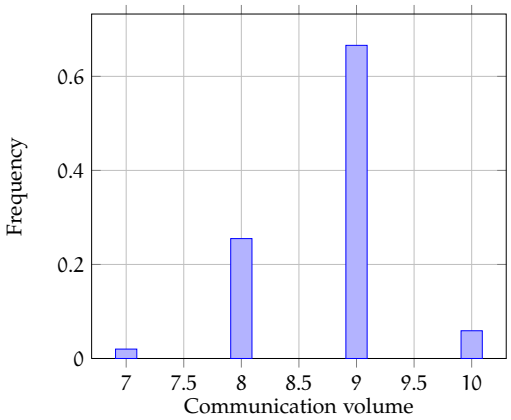
(a) Optimal solution, $V = 0$

(b) `Mondriaan 2.01 hybrid` results, $\langle V \rangle = 7.23$, 1000 runs

Figure 11.: Optimal solution and `Mondriaan` results for the `GD98_a` matrix.

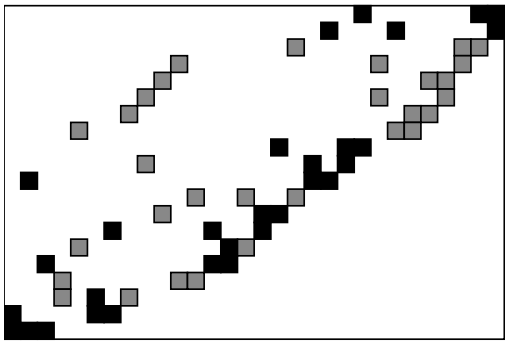


(a) Optimal solution, $V = 5$

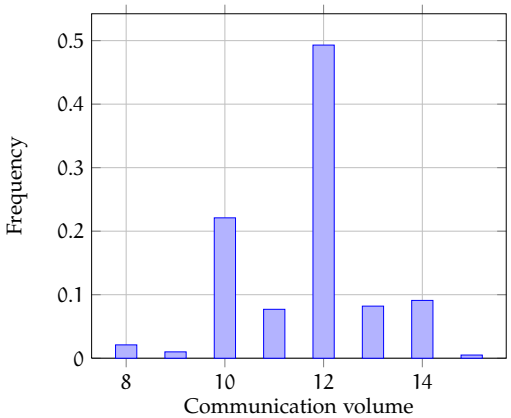


(b) `Mondriaan 2.01 hybrid` results, $\langle V \rangle = 8.76$, 1000 runs

Figure 12.: Optimal solution and `Mondriaan` results for the `klein-b1` matrix.

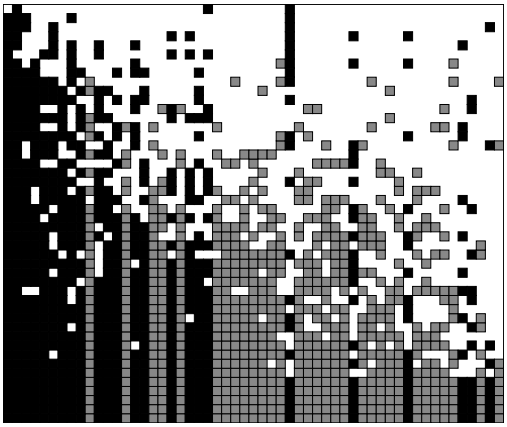


(a) Optimal solution, $V = 6$

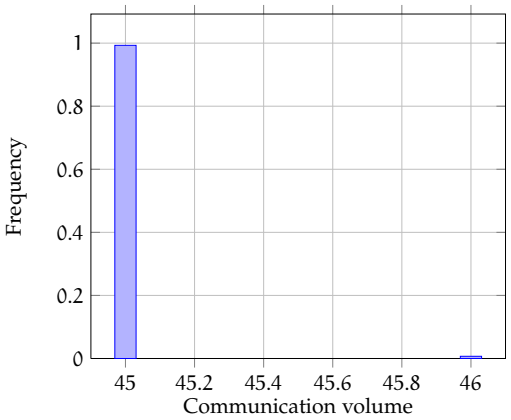


(b) `Mondriaan 2.01 hybrid` results, $\langle V \rangle = 11.65$, 1000 runs

Figure 13.: Optimal solution and `Mondriaan` results for the `klein-b2` matrix.

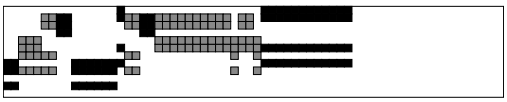


(a) Optimal solution, $V = 38$

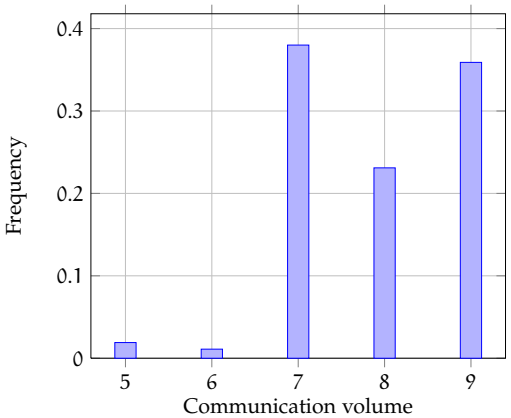


(b) `Mondriaan 2.01 hybrid` results, $\langle V \rangle = 45.01$, 1000 runs

Figure 14.: Optimal solution and `Mondriaan` results for the `Cities` matrix.

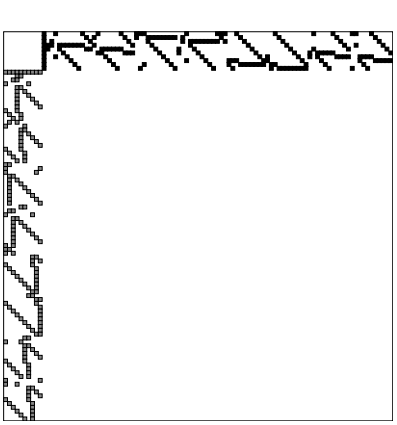


(a) Optimal solution, $V = 4$

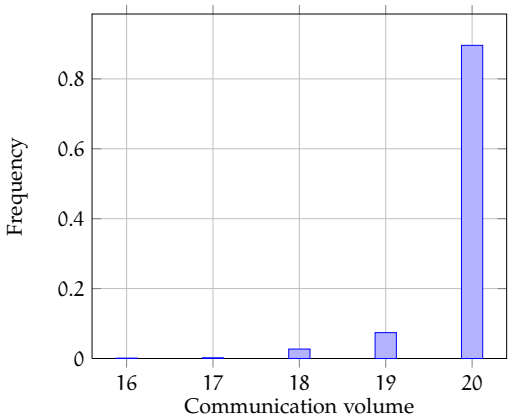


(b) `Mondriaan 2.01 hybrid` results, $\langle V \rangle = 7.90$, 1000 runs

Figure 15.: Optimal solution and `Mondriaan` results for the `relat4` matrix.

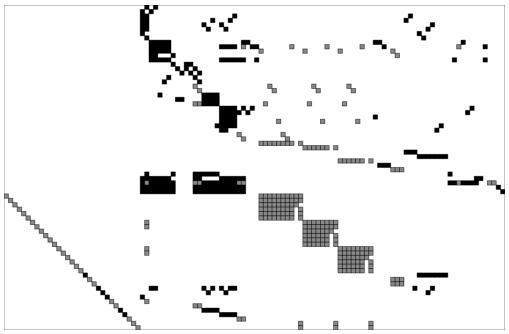


(a) Optimal solution, $V = 0$

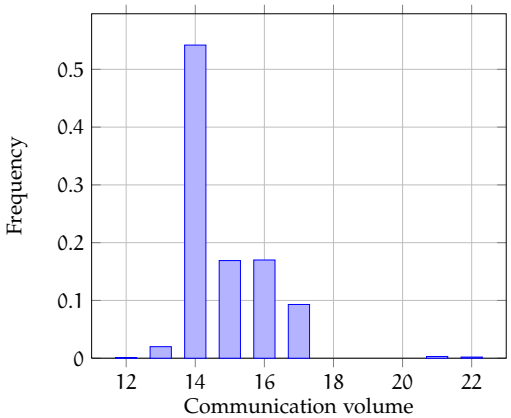


(b) `Mondriaan 2.01 hybrid` results, $\langle V \rangle = 19.86$, 1000 runs

Figure 16.: Optimal solution and `Mondriaan` results for the `GD06_theory` matrix.

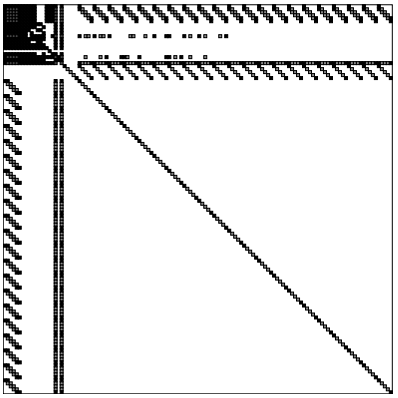


(a) Optimal solution, $V = 11$

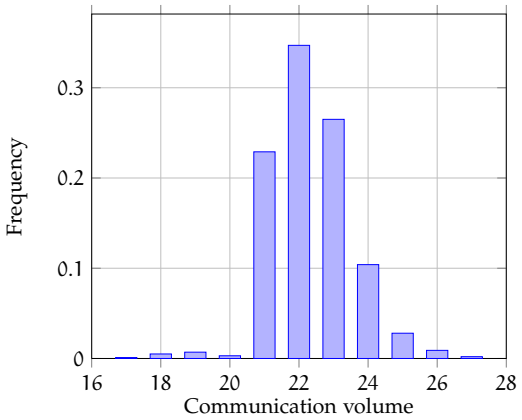


(b) `Mondriaan 2.01 hybrid` results, $\langle V \rangle = 14.80$, 1000 runs

Figure 17.: Optimal solution and `Mondriaan` results for the `lp_blend` matrix.

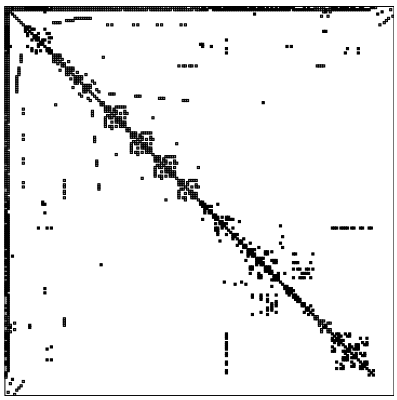


(a) Optimal solution, $V = 14$

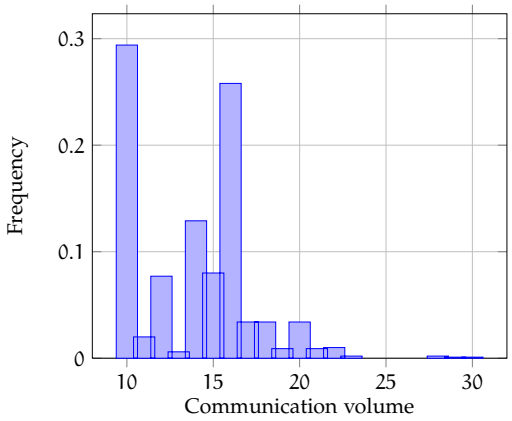


(b) `Mondriaan 2.01 hybrid` results, $\langle V \rangle = 22.32$, 1000 runs

Figure 18.: Optimal solution and `Mondriaan` results for the `arc130` matrix.



(a) Optimal solution, $V = 10$



(b) `Mondriaan 2.01 hybrid` results, $\langle V \rangle = 13.91$, 1000 runs

Figure 19.: Optimal solution and `Mondriaan` results for the `rajat14` matrix.

2.4 LOWER BOUNDS ON THE COMMUNICATION VOLUME

Instead of finding a matrix partitioning with the lowest volume, we can also focus on calculating a lower bound on the optimal communication volume. In other words, given a matrix partitioning instance (a matrix A, the number of processors p, and an allowed load imbalance ε), we want to find a lower bound volume V_{LB}, such that *every* solution of that instance (including the optimal solution) has a volume $\geqslant V_{LB}$. Note that this is a fundamentally different problem than the one discussed in Section 2.1.4. In Section 2.1.4, we were interested in lower bounds on *partial* solutions, where one or more rows and columns were fixed to a certain index. In this case, we try to find a lower bound for the root problem of the branching-tree: in other words, the partial solution with *all* rows and columns unassigned.

An efficient algorithm for computing (good) lower bounds can be used to test available heuristic methods. If a heuristic solution volume V is much larger than V_{LB}, we know that either the lower bound has a low quality ($V_{LB} \ll V_{OPT}$) *or*, more importantly, that the heuristic solution is of low quality. Therefore, a lower bound algorithm that provides provable tight bounds (tight meaning that $V_{LB} \not\ll V_{OPT}$) can be used to spot low quality heuristic solutions, and help to improve the available heuristics.

Furthermore, we might be able to use a sufficiently tight lower bound algorithm together with a good heuristic method to find *optimal* solutions. If the heuristic method (such as `Mondriaan` [40]) is able to produce a solution with a communication volume equal to V_{LB}, we know that this is the optimal solution, since we are guaranteed that no solution with a lower volume than V_{LB} does exist. In this way, we might be able to find optimal solutions to some matrices in polynomial time, but since matrix partitioning is NP-Complete, this method will not work for every possible matrix. As with the optimal solution methods, we will restrict ourselves to the $p = 2$ case in this section.

As explained in the introduction, the matrix partitioning problem can be modelled accurately by a hypergraph partitioning problem. Therefore, a lower bound algorithm for the hypergraph partitioning problem would provide a lower bound for matrix partitioning as well. Unfortunately, to the best of the author's knowledge, a direct lower bound to hypergraph partitioning does not exist yet. As an alternative to a direct lower bound, some studies have focussed on translating hypergraph partitioning problems to graph partitioning problems, and applied known lower bound algorithms to these translated graphs. In this section, we will implement the method proposed in ([22]) for the matrix partitioning problem.

2.4.1 Clique-graphs

The *clique*-graph G of a hypergraph H is defined as follows: for each vertex $v \in H$, we introduce a vertex in G. Furthermore, each hyperedge $e \in H$ becomes a clique in G connecting all vertices of that hyperedge. In the view of matrix partitioning using the finegrain model, the clique-graph has a vertex for each nonzero of the matrix, and each vertex is connected to all vertices that are in the same row or column. Since two neighbouring nonzeroes of the matrix are either in the same row *or* the same column, but not both, there will be no multiple edges in the clique-graph, and the cliques will not overlap each other. Figure 20 shows the transformation procedure for a small matrix.

It is important to note that every vertex partitioning of the clique-graph can be mapped directly to a vertex partitioning of the hypergraph, since both share the same vertices. However, the number of cut edges in the clique-graph is not equal to the number of cut hyperedges in the hypergraph. Therefore, in order to use the clique-graph to obtain a lower bound to the partitioning cost, we need to assign weights to its edges. These weights should be taken such that *every* cut in the clique corresponding to a hyperedge will have a total edge weight $\leqslant 1$. In that case, every partitioning of the clique-graph will have a total cut weight smaller than or equal to the corresponding hypergraph

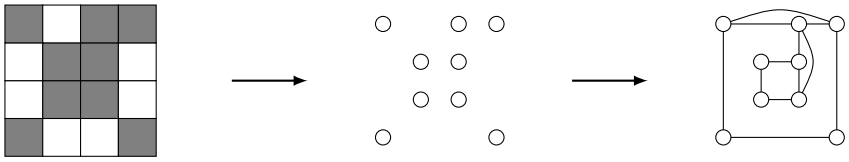


Figure 20.: Transforming a (sparse) matrix to a clique-graph

partitioning, and therefore, a lower bound for the clique-graph partitioning will also be a lower bound for the hypergraph partitioning.

In order to clarify this, let $C_{cl}(i)$ be the total weight of all cut edges of a certain partitioning of clique i, where the cliques are labelled in an arbitrary order. If $C_{cl}(i) = 0$, the vertices of clique i are all assigned to the same subset, and the clique and its corresponding hyperedge are not cut. In this case, the cost of that hyperedge in the hypergraph partitioning will be 0. If $C_{cl}(i) > 0$, the clique and its corresponding hyperedge are cut, and this hyperedge will induce a cost of 1 in the hypergraph partitioning. If we ensure that $C_{cl}(i) \leqslant 1$ for every partitioning, we know that $C_{cl}(i)$ is always smaller than or equal to the cost of the corresponding hyperedge in the hypergraph partitioning. Since there are no overlapping cliques, the cost of a partitioning of the clique-graph is equal to the sum of the $C_{cl}(i)$ values for all cliques. Similarly, the total cost of a partitioning of a hypergraph is the sum of the cost of all hyperedges. Therefore, it is also true that the total cost of a partitioning in the clique-graph will be smaller than or equal to the total cost of the *same* partitioning in the hypergraph.

If we take the optimal partitioning of the hypergraph, its corresponding clique-graph partitioning will have a total cost smaller than or equal to the optimal communication volume, as explained above. The optimal partitioning of the clique-graph will have a cost equal to or smaller than the cost of this partitioning, since the optimal partitioning has the smallest cost of all possible partitionings, by definition. Combining both observations, we can conclude that the optimal partitioning of the clique-graph will have a cost smaller than or equal to the cost of the optimal partitioning of the hypergraph. Therefore, a lower bound to the clique-graph partitioning problem will also be a lower bound to the corresponding hypergraph partitioning problem. This is the basis of the lower bound procedure from [22]. Figure 21 shows the above proof in schematic form.

2.4.2 Edge weights

In order for the above proof to be valid, we will need to define edge weights such that *every* cut in the clique corresponding to a hyperedge will have a total edge weight $\leqslant 1$. If we take a single hyperedge containing N_h vertices, we can calculate the maximum number of cut edges of the corresponding clique. This procedure is shown in Figure 22 for $N_h = 5$. Starting with a clique partitioned to one processor, we assign increasingly more vertices to the other processor. When assigning the ith vertex to the other processor, it is easy to see that we cut $N_h - i$ new edges, but also remove $i - 1$ cut edges (note that in this case, we start counting with $i = 1$). It is trivially true that the number of cut edges of a clique assigned to one processor is equal to zero. Combining the previous considerations, we can write down an equation for the number of cut edges after assigning the ith vertex to the other processor:

$$N_C(i) = \sum_{j=1}^{i} [N_h - j - (j-1)] = \sum_{j=1}^{i} [N_h - 2j + 1] \tag{2.18}$$

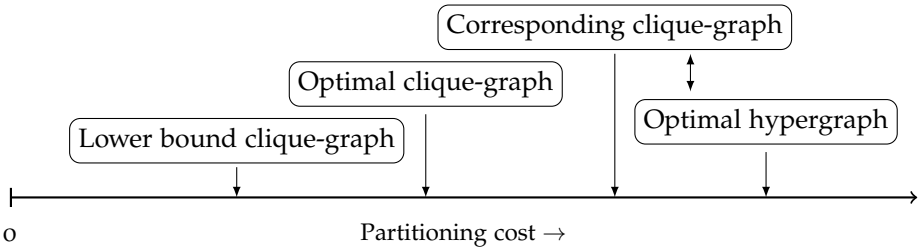


Figure 21.: Schematic representation of the lower bound procedure. The optimal solution to the hypergraph partitioning problem will also be a solution to the corresponding clique-graph partitioning problem with a lower or equal cost. The optimal solution to the clique-graph partitioning problem will have a cost lower than or equal to the cost of this solution. Finally, a lower bound on the optimal cost of the clique-graph partitioning problem will always be smaller than or equal to the optimal cost. The conclusion is that the lower bound for the clique-graph partitioning problem will also be a lower bound for the optimal hypergraph partitioning.

This sum is equal to:

$$N_C(i) = iN_h - i^2 \tag{2.19}$$

When assigning edge weights, we need to know the maximum possible number of cut edges in each clique, and therefore, we find the maximum of $N_C(i)$ by setting the first derivative to zero:

$$\begin{aligned} \frac{d}{di}N_C(i) &= 0 \\ N_h - 2i &= 0 \\ i &= \frac{N_h}{2} \end{aligned}$$

Since N_h is integer, $\frac{N_h}{2}$ is either integer or half integer, so we can round i up or down to the nearest integer with the same result:

$$M_C := \max_i N_C(i) = \left\lfloor \frac{N_h}{2} \right\rfloor N_h - \left\lfloor \frac{N_h}{2} \right\rfloor^2 \tag{2.20}$$

This result is intuitively correct as well: the maximum number of edges are cut if half of all vertices are assigned to the other processor.

We now know that for every clique in the clique-graph, M_C is the maximum number of cut edges. If we set the weight of each edge in the clique to M_C^{-1}, it is therefore trivially true that the maximum cut weight is equal to $M_C \times M_C^{-1} = 1$. So by setting the edge weights of a clique equal to M_C^{-1} we always obey the requirement that *any* cut of the clique must have a total edge weight $\leqslant 1$. Furthermore, there is at least one cut with total edge weight 1 (namely, the one that cuts the maximum M_C edges), so these weights are optimal: we cannot increase them further without violating the requirement that *every* cut in the clique corresponding to a hyperedge will have a total edge weight $\leqslant 1$.

2.4.3 Graph partitioning lower bound

After obtaining a weighted clique-graph from a hypergraph, we still need to find a lower bound on the partitioning cost of the clique-graph. In this report, we will use the method from [15], which is

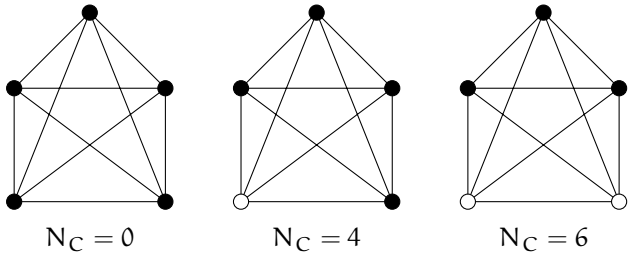


Figure 22.: Number of cut edges N_C for all possible partitionings of a clique of size $N_h = 5$ (symmetries are removed)

an extension of the well-known Donath-Hoffman bound [13]. In this report, we will only discuss the application of the method, but a detailed derivation is given in [15].

The adjacency-matrix A of an edge-weighted graph $G = (V, E)$ is defined as:

$$a_{ij} = \begin{cases} 0 & \text{if } (i,j) \notin E \\ w_{ij} & \text{if } (i,j) \in E \end{cases} \tag{2.21}$$

In this equation, w_{ij} is the edge weight of (i,j). As is clear, A is a symmetric $|V| \times |V|$ matrix with $2|E|$ nonzeroes. Now, let q_i be defined as the sum of all edge weights connected to vertex i. Then, the matrix D is equal to:

$$d_{ij} = \begin{cases} q_i & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{2.22}$$

The calculation of the bipartitioning lower bound of an edge-weighted graph $G = (V, E)$ is performed using the *Laplacian* matrix L of G. Using the definitions that were given above, we can write the Laplacian matrix as $L = D - A$. It is a well-known result from matrix theory that a Laplacian matrix will have only nonnegative, real-valued eigenvalues. For this report, the most interesting result from [15] is that, given an edge-weighted graph $G = (V, E)$ (with its corresponding Laplacian matrix L) and two block sizes m_1 and m_2 (such that $m_1 + m_2 = |V|$), the total edge weight of the optimal partitioning of G over blocks m_1 and m_2 is lower bounded by:

$$LB(G, m_1, m_2) = \lambda_2 \frac{m_1 m_2}{|V|} \tag{2.23}$$

In this equation, λ_2 denotes the second smallest eigenvalue of L. Given a matrix A ($m \times n, nnz$ nonzeroes) and a strict load imbalance ε, we can calculate the block sizes:

$$\begin{aligned} m_1 &= \left\lfloor (1+\varepsilon)\frac{nnz}{2} \right\rfloor \\ m_2 &= nnz - m_1 \end{aligned}$$

Using the above, we can eliminate m_2 from Equation 2.23:

$$LB(G, m_1) = \lambda_2 \frac{nnz \cdot m_1 - m_1^2}{nnz} \tag{2.24}$$

2.4.4 Connected Components

Matrix theory states that the number of zero eigenvalues of a Laplacian matrix is equal to the number of connected components of graph corresponding to the matrix [25]. A connected component

of a graph is a maximal connected subgraph of that graph. In other words, it is a subgraph in which any two vertices are connected to each other by paths, and to which no more vertices or edges can be added without violating the connectivity. If a graph is connected, it consists of one connected component: the graph itself. Otherwise, the number of connected components is larger than one.

If a clique-graph is not connected, the number of connected components is larger than one. Therefore, the number of zero eigenvalues of its Laplacian matrix is larger than one as well, and the second smallest eigenvalue (λ_2) is equal to 0. This presents a problem for the lower bound algorithm described above: $LB(G, m_1)$ of Equation 2.24 will be equal to 0 independent of m_1! It is unlikely that this will represent a tight lower bound: the fact that the clique-graph is not connected should not imply that *every* partitioning of that graph has a cost of 0. Therefore, we would like to be able to extend the lower bound procedure to handle unconnected clique-graphs.

Suppose that we can separate the different connected components of the clique-graph using an efficient algorithm. A simple Depth-First-Search or Breadth-First-Search algorithm can be used to perform this task in $O(|V| + |E|)$ time. By definition of connected components, we know that there are no edges between the different connected components. Therefore, the partitioning cost of the entire clique-graph is equal to the sum of all partitioning costs of its connected components. This presents a method of calculating a lower bound for unconnected clique-graphs: calculate a lower bound for each connected components, and sum all bound to get the final lower bound. Since we are guaranteed that a connected component is a connected graph, we can use the original lower bound method to calculate a lower bound for it. In order to do this, the block sizes m_1 and m_2 have to be defined for each connected component. The only constraint on setting these block sizes is the load imbalance constraint: the sum of all m_1 values for the connected components should be equal to the m_1 value of the original clique-graph. Since the block sizes are dependent on each other, setting them to the correct value is not trivial.

Suppose that we obtained K connected components from the clique graph ($K \geqslant 2$). Let $m_1(i)$ be the m_1 value of connected component i, $\lambda_2(i)$ be the second smallest eigenvalue of the Laplacian of connected component i, and $nnz(i)$ be the number of vertices of connected component i. Using this notation, let $LB(G, i, m_1(i))$ be the lower bound on the partitioning cost of connected component i:

$$LB(G, i, m_1(i)) = \lambda_2(i)\frac{nnz(i)m_1(i) - m_1(i)^2}{nnz(i)} \qquad 0 \leqslant m_1(i) \leqslant nnz(i) \tag{2.25}$$

The lower bound for the entire clique-graph is equal to the sum of all these lower bounds:

$$LB(G, m_1, m_1(0), m_1(1), \ldots, m_1(K-1)) = \sum_{i=0}^{K-1} LB(G, i, m_1(i)) \qquad \sum_{i=0}^{K-1} m_1(i) = m_1 \tag{2.26}$$

Note that this bound depends on how we choose the different $m_1(i)$ values.

Since every distribution of $m_1(i)$ values is possible, the correct lower bound for the partitioning cost of the clique-graph is equal to the *smallest* possible value of Equation 2.26. If we let $\vec{m_1}$ be a vector of length K, denoting how we distribute the $m_1(i)$ values, we can write:

$$LB(G, m_1) = \min_{|\vec{m_1}|=m_1} LB(G, m_1, \vec{m_1}) \tag{2.27}$$

In this case, we define the length of vector $\vec{m_q}$ as the sum of its elements:

$$|\vec{m_1}| = \sum_{i=0}^{K-1} m_1(i) \tag{2.28}$$

The function $LB(G, i, m_1(i))$ is a concave function of $m_1(i)$. Therefore, the minimization function of Equation 2.27 is an example of a *Concave Minimization* problem. Unfortunately, concave minimization is a difficult problem to solve, being NP-Complete [33]. Luckily, an important property of concave minimization is that its minimum will be on the boundaries of the feasible polyhedron [33]. In our case, the feasible polyhedron is defined by the constraint that $|\vec{m_1}| = m_1$. Furthermore, the number of connected components in most graphs will be extremely small (< 10). Therefore, we can use a simple branch-and-bound algorithm to search for the minimum on the boundary. Such a method was used in this thesis.

2.4.5 Results

To summarize the above, we are now in possession of a lower bound algorithm for the matrix partitioning problem:

- Create the clique-graph of A by taking the nonzeroes as vertices, and connecting all nonzeroes that are in the same row or column to each other
- Assign edge weights M_C^{-1} in this clique-graph, according to Equation 2.20
- Find the number of connected components of the clique-graph, and separate them if necessary
- Obtain the Laplacian matrix L from the edge-weighted clique-graph of each connected component
- Calculate the second smallest Laplacian eigenvalue λ_2 for each connected component
- Calculate the partitioning lower bound:
  - If the number of connected components $= 1$: use Equation 2.23 to calculate the lower bound
  - If the number of connected components > 1: use Equation 2.27 to calculate the lower bound, using a branch-and-bound method to solve the concave minimization problem

The above algorithm was implemented in this thesis in the `C` programming language, using the `ARPACK` library [31] to calculate the second smallest eigenvalues. The resulting program was compiled using the `GNU Compiler Collection, version 4.5.0` [2].

The resulting lower bounds for various small and large matrices are given in Table 3. To the best of the author's knowledge, these results represent the first known lower bounds for the matrix partitioning problem. For the small matrices, the lower bound is compared to the optimal communication volume, as given in Appendix B. In case of the larger matrices, a comparison is made with the smallest communication volume found by 1000 runs of the `Mondriaan 3.0` software. The gap between the lower bound V_{LB} and its comparison volume V_{COMP} is defined as:

$$\text{Gap} = \begin{cases} 0 & \text{if } V_{COMP} = 0 \\ \frac{V_{COMP} - V_{LB}}{V_{COMP}} & \text{Otherwise} \end{cases}$$

The computation time of the entire lower bound algorithm is also given in Table 3. Note that, even though its implementation was not fully optimized, the time it takes to calculate the lower bounds is reasonable for most matrices. For the larger matrices, most of the computation time was spend calculating λ_2 using `ARPACK`.

As for the actual lower bounds that were produced, Table 3 shows that in most cases, the gap is very large, indicating that the bounds are not tight. A positive example is the `tols90` matrix, where

Matrix	K	Time (s)	V_{LB}	V_{OPT}	Gap
arc130	1	2.02	8	13	38.5%
bcsstk04	1	1.10	15	38	60.5%
CAG_mat72	1	1.45	4	13	69.2%
can_144	1	0.79	2	12	83.3%
ck104	2	0.44	0	0	0%
rajat11	1	2.10	2	8	75%
rajat14	1	4.94	5	10	50%
robot	1	0.74	2	12	83.3%
steam3	1	0.65	1	8	87.5%
tols90	1	1.26	18	18	0%

(a) Small matrices

Matrix	K	Time (s)	V_{LB}	V_{Best}	Gap
cage10	1	20.92	191	1738	89.0%
gemat11	2	99.48	2	34	94.1%
lhr34	1	$\gg 1000$	1	64	98.4%
memplus	1	427.9	45	192	76.6%
onetone2	5	712.0	12	204	94.1%
west0381	1	0.063	6	50	88.0%

(b) Larger matrices

Table 3.: Results of the lowerbound algorithm for small matrices (of which the optimal volume is known) and larger matrices (with unknown optimal volumes).

the lower bound is equal to the optimal communication volume. For the other matrices, most bounds were much less than half the optimal volume. Furthermore, the quality of the lower bounds seems to decrease with matrix size. We can conclude that the algorithm described in this section is not very practical, since it produces bounds with low quality ($V_{LB} \ll V_{OPT}$). The resulting bounds are therefore not suited for testing heuristic solutions, or for finding optimal solutions. This shows that there is still a need for a high quality direct lower bound algorithm for either the hypergraph or the matrix partitioning problem.

2.5 FURTHER RESEARCH

Although the branch-and-bound method that was developed in this chapter was able to solve fairly large matrices, further improvement is possible. The largest improvements are expected to come from developing new lower bounds to partial solutions. For instance, the fact that the Integer Linear Programs were able to outperform the branch-and-bound method when solving the impcol_b matrix, but were slower for all other matrices, indicates that better lower bounds exist. In fact, theory from integer linear or semidefinite programming could be used to develop these new bounds.

Aside from newer bounds, the actual implementation of the branch-and-bound method might be improved as well. For instance, at the moment the algorithm chooses the next branch to consider in the branching tree by simply processing them in the order in which they appear. Using a smarter

choice of branching, it might be possible to find feasible solutions faster, thereby lowering the current upper bound and reducing the total number of branches that have to be considered.

If the algorithm can be improved so much that it is able to efficiently solve *all* matrices with dimensions smaller than a given bound, it can be used as an optimal bipartitioner in current heuristics. Since the multilevel scheme of most current heuristics allow the input matrix to become arbitrarily small, it can be made small enough for the optimal bipartitioner to solve. However, whether this will result in a better quality heuristic remains unclear.

Instead of focussing on bipartitioning, it is also interesting to obtain optimal results for p values larger than 2. In this case, the worst-case computation time will be much worse than for bipartitioning, since the exponential factors will have p as base. Therefore, the matrices that can be solved will probably be even smaller than for bipartitioning, although it might also be possible to develop new lower bounds that are specific for the $p > 2$ case. However, even optimal solutions for small matrices and p values are of great value, in order to test whether the popular recursive bipartitioning scheme, which is used by almost all current partitioners, is able to produce good results for them.

Information about the optimal solutions to matrices of a certain database, such as the University of Florida Sparse Matrix Collection [12], can be made public in order to help heuristic developers test their partitioners or develop new heuristic methods. The results shown in Appendix B could already be useful in this respect. Information about optimal solutions can be included in the University of Florida Sparse Matrix Collection, similar to the graph drawings of Yifan Hu's drawing algorithm [24]. The advantage of inclusion in the collection is that developers will have a single well-known database to test their heuristics on.

3 HEURISTIC SOLUTIONS

In this chapter, we will describe a novel *heuristic* for the matrix partitioning problem. As opposed to the previous chapter, in this case we are not looking for the optimal solution to the problem, which can take exponential time, but rather try to find a *good* solution in polynomial time. During development of this new heuristic, knowledge of the optimal solutions to the matrices of Appendix B was heavily used.

While studying optimal solutions to the matrix bipartitioning problem, using the algorithms described in Chapter 2, several common characteristics are distinguishable:

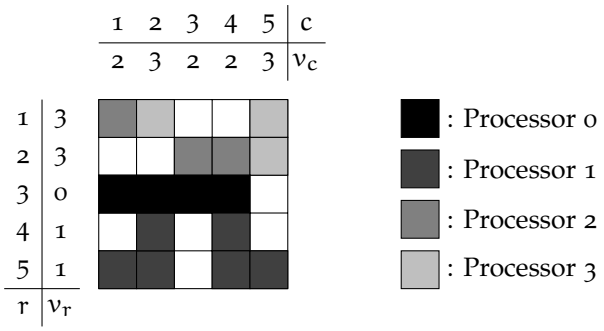
- Most optimal solutions are two-dimensional: both rows *and* columns are cut, instead of only rows *or* columns.
- Most rows and columns are assigned completely to a single processor: communication volumes are generally small compared to the dimensions of the matrix.
- Once a row is assigned completely to a processor, most columns that are connected to this row are assigned completely to the same processor. This is similar for the columns of the matrix.

As a good example of these characteristics, take the optimal solution to the `prime60` matrix, shown on the cover of this thesis, or the `arc130` matrix, shown in Figure 18.

The first observation is an indication that one-dimensional solution methods, often applied in past research, are suboptimal with respect to matrix partitioning. The new heuristic should therefore be two-dimensional in nature. We can use the other observations to help the heuristic find good solutions: the heuristic should encourage these characteristics in the solutions it produces.

The most widely used two-dimensional solution methods ([38, 27, 40]) are based on the `finegrain` model described in Section 1.2. Although this model has the advantage that it is able to exactly describe *any* solution to the matrix partitioning problem, it has the disadvantage that it is too general. For example, the second observation above is not used in any way in the `finegrain` method. In fact, methods that use the `finegrain` model can only assign one nonzero element of a matrix to a single processor at a time. In order to have a row completely assigned to a single processor, all nonzeroes in that row have to be assigned to that processor, one after the other. In most cases, it can be difficult to implement this effectively in a heuristic. Therefore, the new heuristic should not assign individual nonzeroes to a processor, but entire rows and columns. In this way, we ensure that most rows and columns are assigned completely to a single processor.

Summarizing the above, the new heuristic should use a model that is two-dimensional and has rows and columns as its basic structure. A natural model would be to define a variable $v_c(i)$ for each column i and a variable $v_r(i)$ for each row i. The value of $v_q(i)$ describes to which processor row/column i is completely assigned. In other words, row i is defined to be completely assigned to processor $v_r(i)$, and column i is completely assigned to processor $v_c(i)$. However, a problem arises when the row and column of a matrix nonzero a_{ij} are assigned to different processors: to which processor should a_{ij} be assigned? In this new model, it is chosen that the processor with the lowest index takes precedence. For example, if there is a nonzero in row i and column j, and $v_r(i) = 0$ and $v_c(j) = 1$, the nonzero is assigned to processor 0. Finally, we also define a value m_i for each processor index, counting the number of nonzeroes assigned to processor i. An example matrix partitioning over 4 processors, using this model, is shown in Figure 23.



5×5, $N = 16$, $m_0 = 4, m_1 = 6, m_2 = 3, m_3 = 3$, $V = 9$

Figure 23.: Example of the partitioning model used by the new heuristic. The value of a row or column defines to which processor the nonzeroes in that row or column are assigned. When there is a conflict between the row and column value of a matrix nonzero, lower processor indices take precedence.

3.1 A GREEDY HEURISTIC

We can use the model described above to define a simple *greedy* heuristic for the matrix partitioning problem. The $m \times n$ matrix A (N nonzeroes) will be partitioned over p processors, each with a label $\in \{0, 1, \ldots, p-1\}$. Given a certain assignment of v_c and v_r values, a row or column *flip* is defined as changing one (and only one) of the v_c or v_r values to a different processor. We can state the flip function as follows: $\mathtt{flip}(q, i, j)$ changes the value of $v_q(i)$ to j. The *flip cost* is defined as the difference of the cost of the partitioning after and before the flip. If the flip cost is negative, the partitioning after the flip has a lower cost than before.

The heuristic starts with setting:

$$v_r(i) = v_c(j) = p - 1 \qquad \forall \quad 0 \leqslant i < n, 0 \leqslant j < m \tag{3.1}$$

At this point, all nonzeroes of A are assigned to processor $p-1$, and therefore the communication volume is 0. However, there is a large load imbalance since $m_{p-1} = N$ and $m_{i,i\neq p-1} = 0$. We will now repeatedly apply the $\mathtt{flip}(q, i, 0)$ function that has the lowest cost. In other words, out of all possible column and row flips from processor $p-1$ to processor 0, we apply the one with the lowest flip cost. If there is more than one flip with the lowest flip cost, we randomly select one to apply. Furthermore, we only allow flips for which the new partitioning has a m_0 value that is smaller than the right hand side of the load imbalance constraint (Equation 1.1):

$$\max_i m_i \leqslant (1 + \varepsilon)\frac{N}{p}$$

This process of flipping rows and columns from processor $p-1$ to processor 0 is repeated until no such flips exist. At this point, all $\mathtt{flip}(q, i, 0)$ functions that are left will result in a m_0 value that violates the load imbalance constraint.

The next step is to apply the same procedure to processor 1: out of all possible column and row flips from processor $p-1$ to processor 1, we repeatedly apply the one with the lowest flip cost. Again, we only allow flips where m_1 obeys the load imbalance constraint, stopping when no such flips exist. Since lower processor indices take precedence, all nonzeroes assigned to processor 0 remain

unchanged, and m_0 is unchanged by all flips from processor $p-1$ to processor 1. Therefore, when assigning the rows and columns to processor i, all m_j values with $0 \geqslant j < i$ remain unchanged, and will obey the load imbalance constraint. We proceed in this way for all processor indices $< p-1$. After these steps, we are guaranteed that all processors $< p-2$ obey the load imbalance constraint.

While processing the final processor index $(p-2)$, we keep track of all partitionings that obey the load imbalance constraint for *all* processors. Although there is no guarantee that such partitionings exist, the fact that all processors with an index $< p-2$ *just* obey the load imbalance has the result that the probability that at least one feasible partitioning will be found is very high in practice. After finishing assigning processor $p-2$, the heuristic returns the partitioning that has the lowest communication volume out of all that were found obeying the load imbalance constraint.

This heuristic is *greedy* in nature: at each point, a locally optimal choice is made which row or column to flip. Although greedy algorithms produce optimal results for some problems (see for example, Kruskal's algorithm for the minimum spanning tree of a graph [29]), in general they can produce very bad results. In order to improve the quality of a greedy heuristic we can apply a local search algorithm to the solution it produces.

3.2 LOCAL SEARCH

A local search algorithm is a method that takes a given partitioning obeying the load imbalance constraint and repeatedly apply small changes to that partitioning in order to lower its cost. In our case, the small changes will be the row and column flips defined above. Unlike the greedy heuristic, we allow *all* $\mathtt{flip}(q,i,j)$ functions for which the resulting partitioning will obey the load imbalance constraint.

For the matrix, graph and hypergraph partitioning problem, a very popular local search method is the Kernighan-Lin method [28], in the Fiduccia-Mattheyses form [16]. This name of this method is often abbreviated to KLFM, which is the name used in the rest of this chapter. Originally, KLFM was designed for *bi*partitioning *graphs*. However, it is also very commonly applied for bipartitioning hypergraphs, for which it requires very little adaptation. A less common extension is to allow partitioning over more than 2 processors, as most current solutions using recursive bipartitioning instead. In this thesis, a simple extension to $p > 2$ is used, where we simply allow all $\mathtt{flip}(q,i,j)$ functions that obey the load imbalance constraint.

In short, the KLFM method works by repeatedly applying the $\mathtt{flip}(q,i,j)$ function that has the lowest flip cost and obeys the load imbalance constraint. In order to prevent the algorithm flipping a single row and column over and over again, a row or column is *locked* to its new processor index after it has been flipped. After this, we are not allowed to flip this row or column again. In the end, all rows and columns are either locked, or cannot be flipped without violating the load imbalance constraint. Let s_0 be the partitioning at the start of the KLFM method, and s_{end} the partitioning at the end. In between both, we have a list of feasible partitionings $\{s_0, s_1, \ldots, s_{end}\}$, each with an associated partitioning cost. The output of the KLFM method is the s_i partitioning with the *lowest* cost. Note that this could be s_0, the partitioning that was given as input to the method, meaning that the KLFM method was not able to improve the given partitioning.

We can also use the output of the KLFM method as the input to another run of the KLFM method. Since rows and columns that were locked in the previous KLFM run are no longer locked at the start of the new run, it is possible that the second run is able to find an even better solution. After the second run, even more KLFM runs can be applied to further improve the partitioning.

3.3 A COMBINED HEURISTIC

By combining the greedy heuristic of Section 3.1 with the KLFM method of Section 3.2, we get a method for finding good solutions to the matrix partitioning problem. This combined heuristic starts with performing the greedy heuristic to obtain a feasible solution to the matrix partitioning problem. After this, the KLFM method is applied N_{runs} times, improving the solution of the greedy heuristic. The output of the combined heuristic is the partitioning that is obtained after the final KLFM run. The resulting algorithm is shown below.

Algorithm 1 Greedy+KLFM

$s_0 \leftarrow$ GREEDYHEURISTIC
for $q = 1$ to N_{runs} **do**
  $t \leftarrow 1$
  while $\exists$ feasible $\mathtt{flip}(q, i, j)$ **do**
    $\mathtt{bestflip} \leftarrow$ minimum cost feasible $\mathtt{flip}(q, i, j)$
    $s_t \leftarrow \mathtt{bestflip}(s_{t-1})$
    $t \leftarrow t + 1$
  end while
  $s_0 \leftarrow$ lowest cost solution $\in \{s_0, s_1, \ldots, s_{t-1}\}$
end for
return s_0

3.4 MULTILEVEL PARTITIONING

Although Algorithm 1 is able to partition small matrices reasonably well, its computation time and solution quality become much worse for larger matrices. Depending on the actual matrix that is partitioned, Algorithm 1 tends to break down at matrices with around 5000 rows and columns and larger. This problem is not specific to this heuristic, as it is commonly encountered when using the KLFM method.

An efficient solution to this problem is to apply *multilevel* partitioning, which is used by practically every popular partitioner currently available, like PaToH ([38]), hMetis ([27]) and Mondriaan ([40]). In multilevel partitioning, before partitioning, we start by constructing a sequence of increasingly coarse approximations of the given matrix. Each approximation matrix is smaller than the one preceding it, while still retaining certain characteristics of the larger matrix. The key is to coarsen the matrix in such a way, that the smaller matrices are a good representation of the larger matrix.

After coarsening, the smallest matrix is partitioned using the heuristic above. Since we are able to choose at which point we stop coarsening, we can make the final matrix small enough to be efficiently partitioned by Algorithm 1. Because the matrix is small, it can be partitioned relatively fast, and therefore we will partition it N_{init} times and choose the one with the lowest cost. Since the small matrix is only an approximation of the large matrix, this partitioning will not be a good partitioning of the large matrix in general. Therefore, the initial partitioning is *refined* to the larger matrices in the sequence, until a good partitioning of the largest matrix is obtained.

In a refinement step, the partitioning of a small matrix is translated to the same partitioning in the larger matrix it represents. Then, one or more KLFM runs are applied to this partitioning, possibly improving its quality. This new partitioning is then refined to the even larger matrix that precedes

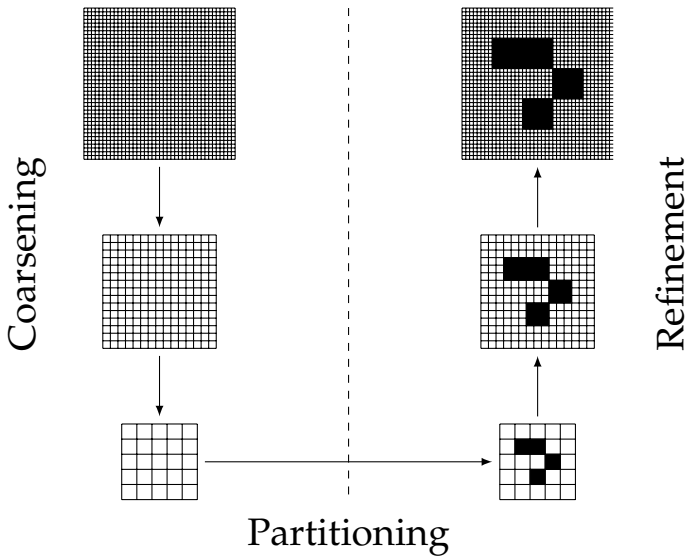


Figure 24.: Schematic representation of the multilevel scheme used in this thesis. The input matrix is first coarsened to a smaller matrix, which is partitioned using the Greedy+KLFM method. The partitioned matrix is refined back to the original matrix, with KLFM runs at each refinement step.

it in the coarsening sequence. After refining each partitioning in the sequence, we end up with a partitioning for the input matrix. This multilevel scheme is shown in Figure 24.

The resulting multilevel matrix partitioning heuristic is given in Algorithm 2.

3.5 COARSENING AND REFINEMENT

The question remains how to coarsen the input matrix to smaller matrices, such that the characteristics of the larger matrix remain present in the smaller matrices. In this thesis, an approach similar to [37] is taken. Although both methods are similar, the context in which they are used is very different, and development of both methods was independent of each other. The basic operation used in the coarsening process is that of row or column *merging*, where two rows or two columns are merged into one. Note that we do not allow merging one row and one column together. In the coarsening process, we try to merge rows and columns that are similar to each other, in order to retain matrix characteristics.

The entire coarsening process consists of different steps, in which we either try to merge all rows or all columns of the matrix. For instance, if we start with merging the columns of an $m \times n$ matrix A_0, we get a matrix A_1 with approximately $\frac{m}{2}$ columns and n rows. Then, if we choose to merge the rows next, we end up with a matrix A_2 with approximately $\frac{m}{2}$ columns and $\frac{n}{2}$ rows. Exactly how the choice is made between merging rows or columns in this report will be explained in Section 3.6. While merging, it is possible that, for a single row or column, no rows or columns are available to merge with: either all other rows and columns have already been merged, or are so dissimilar that merging with them would be disadvantageous. In this case, we simply *insert* that row or column in the smaller matrix, with no alteration. Because of this, it is possible that the number of rows or

Algorithm 2 Multilevel partitioning heuristic

```
A_0 ← input matrix
i ← 0
while A_i dimensions > given bound do
  A_{i+1} ← coarsen(A_i)
  i ← i + 1
end while
i ← i − 1
for q = 1 to N_init do
  s_q ← Greedy+KLFM(A_i)
end for
s ← s_q with lowest cost
while i > 0 do
  s ← translate(s,A_{i−1})
  for q = 1 to N_ref do
    s ← KLFM(s_0)
  end for
  i ← i − 1
end while
return s
```

columns in the coarsened (smaller) matrix is larger than half of the rows or columns of the larger matrix.

For each row and column of matrix A_j, we define a $nr_q(A_j, i)$ variable. The value of $nr_q(A_j, i)$ is defined as the number of rows or columns of the input matrix which are represented by row or column i of matrix A_j. For example, if $nr_c(A_j, 10) = 3$, column 10 of A_j represents 3 columns of the input matrix, and if $nr_r(A_j, 4) = 5$, row 4 of A_j represents 5 rows of the input matrix. It is clear that $nr_q(A_0, i) = 1$ for all rows and columns of the input matrix, since each row and column represents exactly one row or column in the input matrix: itself. During coarsening, the $nr_q(A_j, i)$ values of the smaller matrices will increase, since multiple rows and columns are merged to single rows and columns. In fact, if we merge rows j and k of matrix A_i together to create a new row l in the smaller matrix A_{i+1}, we let:

$$nr_r(A_{i+1}, l) = nr_r(A_i, j) + nr_r(A_i, k) \tag{3.2}$$

The equation above is similar for matrix columns instead of rows. During row or column insertion, $nr_q(A_{i+1}, l) = nr_q(A_i, l)$.

Similar to rows and columns, we define a variable $nz(A_k, i, j)$, which indicates the number of nonzeroes of the input matrix that are represented by the matrix nonzero in column i and row j of matrix A_k. For example, if $nz(A_k, 10, 6) = 4$, the nonzero in column 10 and row 6 of matrix A_k represents 4 nonzeroes of the input matrix. If $nz(A_k, i, j) = 0$, it means that there is no nonzero in column i and row j of matrix A_k. For the input matrix A_0, $nz(A_0, i, j) = 1$ if there is a nonzero at (i, j), and $nz(A_0, i, j) = 0$ otherwise. During row or column merging, we add the $nz(A_k, i, j)$ values of both rows or columns to get the new value for $nz(A_{k+1}, i, j)$. For instance, if we merge rows i and j of matrix A_k together to create a new row l in the smaller matrix A_{k+1}, we let:

$$nz(A_{k+1}, q, l) = nz(A_k, q, i) + nz(A_k, q, j) \qquad \forall \quad 0 \leqslant q \leqslant m \tag{3.3}$$

In the equation above, m is equal to the number of columns of matrix A_i. We can view Equation 3.3 as the definition of the merging process: a merged row has a nonzero in column i if and only if at

least one of its parent rows has a nonzero in column i. The equation, which is similar for columns instead of rows, ensures that the value for $nz(A_k, i, j)$ is always equal to the number of nonzeroes represented by (i, j) of matrix A_k. It follows that the sum of all $nz(A_k, i, j)$ values is always equal to N, the number of nonzeroes of the input matrix. Finally, the $nz(A_k, i, j)$ values remain unchanged during column and row insertion.

An example of the entire coarsening process is given in Figure 25. The 9×9 cage4 matrix from [12] is coarsened in 4 steps to a 3×3 matrix. The rows and columns are alternately merged, starting with the columns. Note that the main characteristics of the cage4 matrix, like its banded nature, are copied to the smaller matrices by the coarsening process.

During refinement, we need to translate a partitioning of a small matrix A_i to a partitioning of the larger matrix A_{i-1}. The translation method used in this thesis is fairly simple. Suppose that during coarsening, columns i and j of matrix A_{k-1} were merged to form column l of matrix A_k. In the partitioning of A_k, column l is assigned to processor $v_c(A_k, l)$. In the translated partitioning of A_{k-1}, columns i and j are defined to be assigned to the *same* processor. In other words: $v_c(A_{k-1}, i) = v_c(A_{k-1}, j) = v_c(A_k, l)$.

The cost of a multilevel partitioning of a matrix is slightly different than in normal partitioning. In normal matrix partitioning, the partitioning cost of a single row or column is equal to the number of different processors in that row or column, minus one. This is usually defined as the $(\lambda - 1)$ metric. In a coarsened matrix A_j, the cost of a single row or column is equal to $nr_q(A_j, i)(\lambda - 1)$. In other words, the cost of each row and column is multiplied by the $nr_q(A_j, i)$ value. The cost of the entire partitioning is still equal to the sum of all column and row costs. Using this definition of partitioning cost, we are guaranteed that the cost of a partitioning of A_i is always larger than or equal to the cost of the same partitioning translated to the larger matrix A_{i-1}. In order to see this, note that if row j is cut in matrix A_i, we interpret this as if all the rows of the input matrix that that row represents are cut. In the larger matrix A_{i-1}, the same translated partitioning either cuts the same input matrix rows that row j of matrix A_i cuts, or *less*. Since the total cost of a partitioning is the sum of all cut rows and columns, the cost of the partitioning of A_i can never be smaller than the cost of the same partitioning translated to matrix A_{i-1}.

Similar to the partitioning cost, the load imbalance constraint is adjusted, such that each nonzero of A_k contributes its number of represented nonzeroes to the subset sizes:

$$\max_q \sum_{(i,j) \in A_q} nz(A_q, i, j) \leqslant (1 + \varepsilon) \frac{N}{p} \tag{3.4}$$

If the nonzero at (i, j) of matrix A_k is assigned to processor 2, we interpret this as if all $nz(A_k, i, j)$ nonzeroes have been assigned to processor 2. In this way, if a partitioning of A_i obeys the load imbalance constraint, it is guaranteed that the same partitioning translated to the larger matrix A_{i-1} also obeys the load imbalance constraint.

3.6 IMPLEMENTATION

We will now discuss how the multilevel algorithm is implemented in this thesis. The main components of the algorithm are the KLFM method and the coarsening steps, which will take the most computation time.

A big advantage of the KLFM method is that it can be implemented such that the computation time of a single run depends linearly on the size of the problem. In order to do this, the method uses an advanced data structure called a *bucket-list*. In this thesis however, a proof-of-concept version was implemented that produces statistically identical results, but is much slower asymptotically.

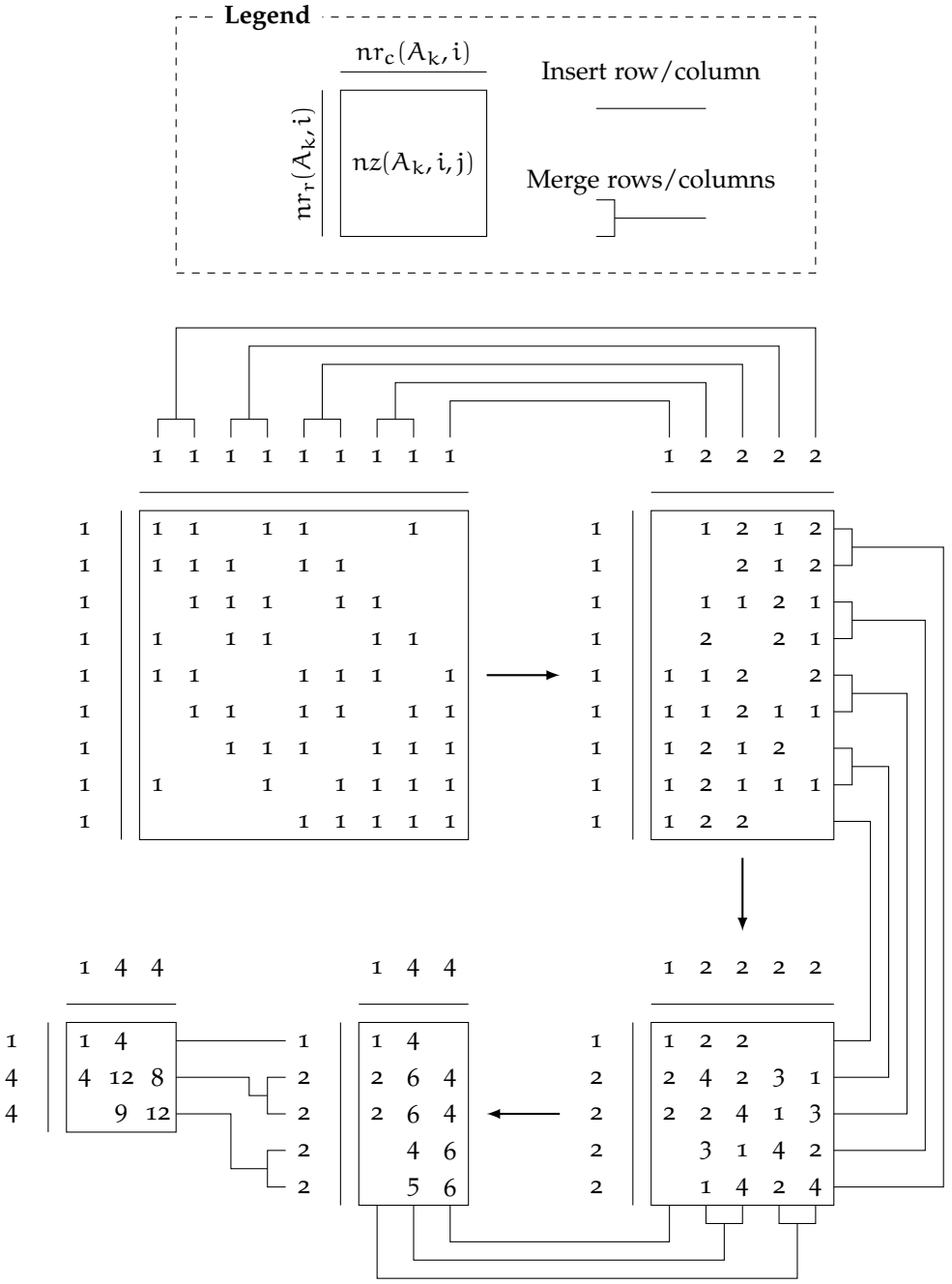


Figure 25.: Example of a coarsening process for the `cage4` matrix with zeroes omitted. Columns and rows are merged with similar ones to create smaller matrices, and the ones that remain are simply inserted. Note that the smaller matrices retain certain characteristics of the larger matrix (although possibly reflected in one or two directions), such as its banded nature. This is the main objective of the coarsening step: to produce smaller matrices that are good representations of the larger matrix.

By simply trying all possible flips and applying the one with the lowest cost, this version has a computation time that depends quadratically on the input size of the problem (matrix dimensions, number of processors). The proof-of-concept version was chosen because the main goal of the implementation was to calculate partitioning costs and compare them with other partitioners, and because it was easier to implement.

The coarsening steps were implemented as follows. First, a choice is made whether to merge the columns or the rows of the current matrix. Let's assume we have chosen to merge the rows, in order to simplify the following explanations. We will pick rows to merge in order of decreasing number of nonzeroes contained in them. If there are multiple rows of the same size, a random row is picked to merge. After picking a row i to merge, we consider all its unmerged *neighbouring* rows as candidates to merge with. Two rows i and j are neighbours if there is at least one column in which both rows have a nonzero. During merging, we exclude neighbours that will result in a merged row that includes more than a $0.2/p$ fraction of all nonzeroes, in order to prevent the appearance of a single dominating row. Out of all its unmerged neighbours, we merge row i with the one with which it shares the most columns. If no unmerged neighbours exist for row i, it is simply inserted in the smaller matrix. In this thesis, a naive version of the coarsening process was implemented, which takes $O(n^2m)$ time to coarsen the rows of a $m \times n$ matrix. The coarsening algorithm simply considers all possible neighbours of the row that is currently merged, and merges it with the one with which it shares the most columns. Apart from speeding up the process, it is also possible to define different ways to choose which columns and rows to merge. Although other methods have not been tested in this thesis, it is possible that better methods than the one used exist.

Finally, we need to define a way to choose the directions in which to merge at each step of the coarsening process. Again, several possibilities are available, and not all have been tested. For example, we can choose to merge rows and columns alternately, starting with the columns. Similarly, it is also possible to merge them alternately, but start with the rows. Furthermore, we can also choose to merge in the direction that is the largest at the beginning of each step. For example, if a matrix has more rows than columns, we merge all rows next. Note that in this case, it is possible that we may also merge all rows in the next step, if the smaller matrix still has more rows than columns. Beforehand, it is hard to predict which of the three methods will produce the best results, since it depends on the actual nonzero pattern of the matrix that is partitioned. Therefore, in the new heuristic, all three methods are tried, and lowest cost out of all three is returned.

The entire coarsening process is stopped if the sum of the number of rows and columns of smallest matrix is smaller than a given bound N_{bound}. In the heuristic used in this thesis, N_{bound} was set to 2000. The other parameters, N_{init}, N_{runs} and N_{ref} were set to $N_{init} = 8$, $N_{runs} = 25$ and $N_{ref} = p$. In other words, the smallest matrix is partitioned 8 times, using 25 KLFM runs each time, and the lowest cost partitioning is used. Furthermore, during each refinement step, p KLFM runs are applied to improve the current solution. These parameters were chosen similar to the default parameters of `Mondriaan 3.0`, in order to make the comparisons of Section 3.7 fair. Although it turned out that the values of these parameters can have a large influence on the resulting partitioning cost, the optimal values are unknown. Therefore, performing a tuning procedure similar to the one that is applied to `Mondriaan` in Appendix A might improve the quality of the new heuristic.

3.7 RESULTS

In order to test the quality of the new heuristic that was developed in this chapter, we compare the communication volumes it produces for a selection of test matrices and number of processors with the communication volumes produced by the state-of-the-art partitioner `Mondriaan 3.0`, using the `hybrid` setting and all other parameters at their default value. The test matrices are chosen

Matrix	Columns	Rows	Nonzeroes	Symmetric
c98a	56243	56274	2075889	no
lp_cre_b	9648	77137	260785	no
polyDFT	46176	46176	3690048	no
lp_dfl001	6071	12230	35632	no
tbdmatlab	19859	5979	430171	no
onetone2	36057	36057	227628	no
tbdlinux	112757	20167	2157675	no
bcsstk30	28924	28924	2043492	yes
cage10	11397	11397	150645	pattern
bcsstk32	44609	44609	2014701	yes
finan512	74752	74752	596992	yes
gemat11	4929	4929	33185	no
lhr34	35152	35152	764014	no
memplus	17758	17758	126150	pattern

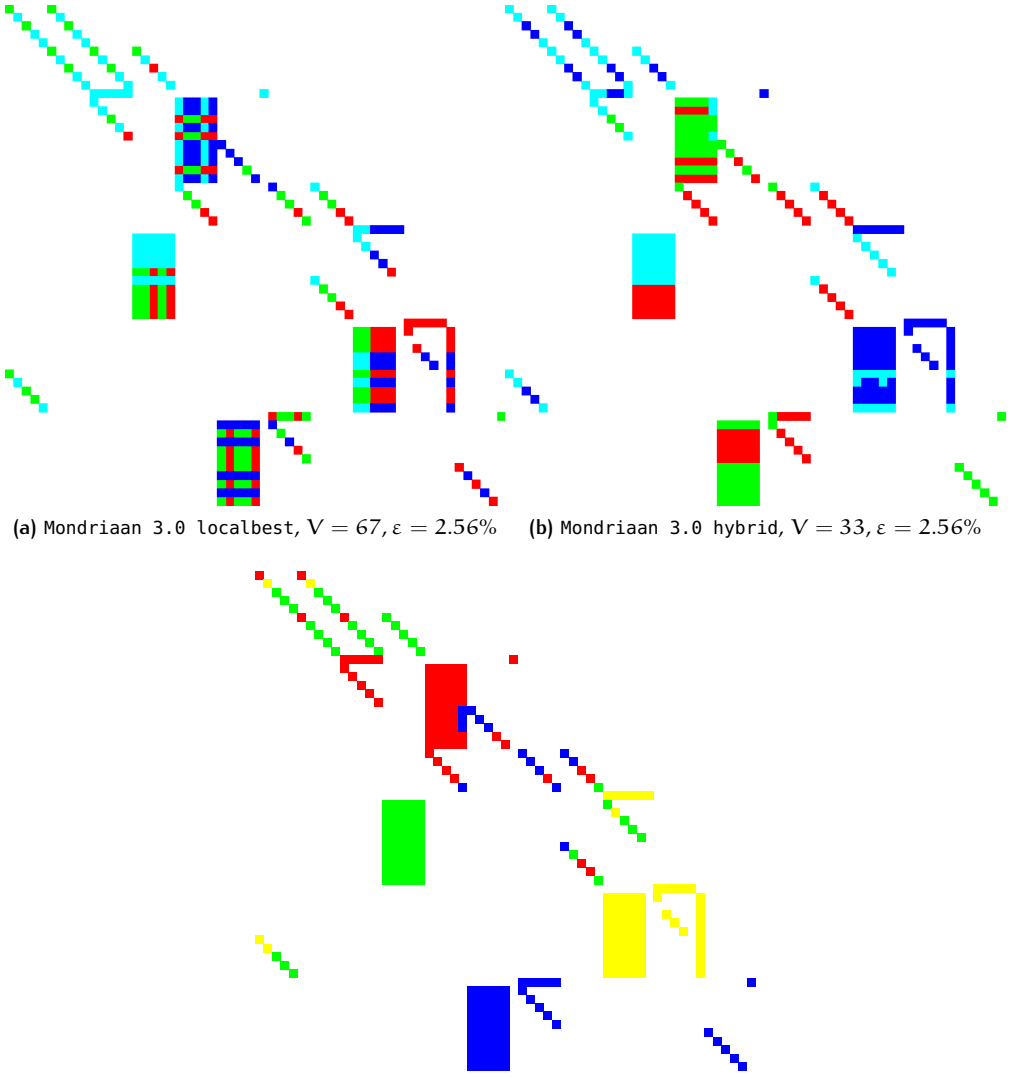
Table 4.: Test matrices used for comparison of the new heuristic with `Mondriaan 3.0`. Pattern symmetry occurs when the nonzero values themselves are not symmetric, but the nonzero pattern is.

such that they represent a wide variety of application areas. The properties of the matrices are given in Table 4. All matrices except the `polyDFT` matrix can be found in the University of Florida Sparse Matrix Collection [12]. The `polyDFT` matrix represents self-assembly of polymers. For a visual comparison of the solutions that both partitioners are able to produce, see Figure 26, where the `impcol_b` matrix is partitioned over 4 processors.

For each matrix of the test set and p value $\in \{2, 4, 8, 16, 32, 64\}$, the new heuristic and `Mondriaan` were run 10 times, and the lowest communication volume was returned. In Table 5, the lowest communication volumes that were found by both methods are shown. Since the new heuristic was implemented in a very inefficient proof-of-concept version, a comparison of computation time would be of no interest, and is therefore not shown in this thesis. A fair comparison of computation time can only be made after the heuristic is implemented in a more efficient way.

For a large fraction of the test matrices, the new heuristic produces similar results to `Mondriaan`. In almost 36% of the cases, the heuristic was actually able to produce solutions with a lower communication volume than `Mondriaan`. An interesting case is the `polyDFT` matrix. For $p \leqslant 4$, `Mondriaan` is able to find a solution with a communication volume smaller than or equal to the new heuristic. For larger p values however, the new heuristic produces much better results. Whether this is an indication of failing recursive bisection, or if there is a different reason for `Mondriaan`'s suboptimal behaviour, is unknown. The `memplus` matrix is interesting as well, since the new heuristic consistently outperforms `Mondriaan` for all p values. For $p = 2$, the heuristic is able to find a solution that has a communication volume that is only 34.7% of the communication volume that `Mondriaan` finds. We hope that the insights of this thesis will improve `Mondriaan`'s behaviour for this matrix, and others.

On the negative side, the new heuristic performs much worse for some matrices. For example, for the `lp_creb` matrix and $p = 2$, `Mondriaan` is able to find a solution that has a communication volume that is 23.6% of the communication volume that the heuristic produces. However, the suggestions of Section 3.8 might be able to improve the heuristic for these matrices.



(a) `Mondriaan 3.0 localbest`, $V = 67, \varepsilon = 2.56\%$ (b) `Mondriaan 3.0 hybrid`, $V = 33, \varepsilon = 2.56\%$

(c) New heuristic, $V = 30, \varepsilon = 2.56\%$

Figure 26.: Solutions of the matrix partitioning problem for the `impcol_b` matrix, which is a 59×59 matrix with 312 nonzeroes. Shown are different partitionings over 4 processors with $\varepsilon = 3\%$, obtained by `Mondriaan 3.0 localbest`, `Mondriaan 3.0 hybrid` and the new heuristic developed in this thesis. Different colours indicate different processor indices. Note that the new heuristic was able to find the lowest communication volume out of all.

Matrix	p	Mondriaan	New	Matrix	p	Mondriaan	New
c98a	2	**44787**	56229	lp_cre_b	2	**548**	2320
	4	100023	**94428**		4	**1244**	4103
	8	**164890**	168073		8	**2370**	5732
	16	**237416**	281691		16	**3967**	7256
	32	**332549**	430842		32	**6235**	10632
	64	**444065**	582561		64	**9121**	18953
polyDFT	2	**3456**	**3456**	lp_dfl001	2	**573**	825
	4	**8472**	8520		4	**1360**	1500
	8	18516	**15818**		8	**2407**	2685
	16	37078	**26499**		16	**3534**	3866
	32	53881	**46779**		32	**4741**	5259
	64	74036	**68104**		64	**5908**	6325
tbdmatlab	2	**4698**	5049	onetone2	2	**204**	256
	4	**10101**	10617		4	**658**	1180
	8	16705	**16585**		8	**1524**	2300
	16	**25062**	27359		16	**2590**	3260
	32	**36900**	41324		32	**3672**	4362
	64	**53598**	68699		64	**5364**	5671
tbdlinux	2	13212	**10297**	bcsstk30	2	535	**489**
	4	33795	**25273**		4	1562	**1452**
	8	54101	**45336**		8	**3566**	3619
	16	**76432**	79331		16	7771	**7260**
	32	**113378**	140592		32	13141	**13068**
	64	162242	200579		64	21326	**21015**
cage10	2	1765	**1738**	bcsstk32	2	**788**	807
	4	**3798**	4142		4	1845	**1609**
	8	**5782**	6859		8	3696	**3273**
	16	**8272**	9620		16	6461	**6296**
	32	**11255**	13527		32	**10780**	10942
	64	**15595**	17038		64	**17102**	18328
finan512	2	106	**100**	gemat11	2	**34**	78
	4	233	**200**		4	**71**	167
	8	486	**450**		8	**163**	298
	16	978	**804**		16	**345**	488
	32	1809	**1620**		32	**588**	754
	64	**7706**	10945		64	**1042**	1192
lhr34	2	**64**	94	memplus	2	554	**192**
	4	**327**	383		4	1296	**674**
	8	**800**	1172		8	2436	**1439**
	16	**1532**	2009		16	3843	**2208**
	32	**3127**	4580		32	5828	**3864**
	64	**5714**	9003		64	7999	**5492**

Table 5.: Comparison of the lowest communication volume found in 10 runs for the new heuristic and `Mondriaan 3.0 hybrid`. Bold values depict the lowest of all communication volumes that were found for that matrix and p value.

3.8 FURTHER RESEARCH

The heuristic method that was introduced in this chapter is meant as a first attempt at moving beyond hypergraph models for solving the matrix partitioning problem. As such, there is a lot of room for improvement, although current results are already very promising. An obvious improvement would be to rewrite the current proof-of-concept implementation to a high-quality one. The main task would be implementing a more efficient KLFM and coarsening method. Since the new heuristic flips matrix rows and columns instead of nonzeroes, the computation time of a good implementation with respect to the matrix dimensions should be comparable to the `localbest` option of `Mondriaan 3.0`.

The heuristic algorithm itself can be improved as well. For example, many different methods to coarsen a matrix are possible. The `Mondriaan 3.0` package has 6 different options for choosing which vertex to merge next, and 5 more that influence with which vertex the chosen vertex is merged. This already creates 30 combinations of coarsening options, and even more parameters related to coarsening exist. In the current version of the new heuristic, only one coarsening method was implemented, and it is possible that other methods can improve the quality of the solutions by a large amount.

Furthermore, several variations of the KLFM method exist that might be worthwhile to investigate. In one variation the flip cost function is redefined to include a lookahead, where we measure the maximum improvement that is possible in several flips after the current one. In other words, instead of finding the cost of flipping only one vertex, we find the cost of flipping several vertices at once, but apply only one of them. The advantage of this method will be that the heuristic can make a better choice of which vertex to flip next, at the cost of an increased computation time.

In another variation, we allow flips that exceed the load imbalance constraint, but restrict the algorithm to flips that improve balance afterwards, until the load imbalance constraint is obeyed. Using this variation, we prevent that the algorithm is confined to a small part of all possible solutions, although it is possible that solution quality will suffer if the algorithm spends a lot of time restoring load balance.

Instead of partitioning the matrix in p parts directly, the new heuristic can also be implemented using recursive bipartitioning. An advantage of this approach will be that the computation time of the resulting algorithm will scale much better with the number of processors p, although the effect on solution quality is hard to predict. In this form, the new heuristic can be integrated in the `Mondriaan` software, possibly creating a new `hybrid` option where the `localbest`, `finegrain` *and* the new heuristic are tried and the best solution is returned. As the results from this chapter have shown, the current implementation of the new heuristic is already able to produce better solutions than `Mondriaan` for some matrices. Therefore, the new `hybrid` option will probably produce higher quality solutions than the current one, with a small added computational cost, since the new heuristic will perform similar to the `localbest` option. It is also interesting to see whether a combination of the `localbest` option and the new heuristic will produce better solutions than the current `hybrid` option, which is a combination of the `localbest` option and the `finegrain` option.

As a final note, parallelizing the current heuristic will lower its computation time, and enable it to partitioning larger matrices in reasonable time. However, since the KLFM method works in sequential steps (it flips one single row/column after another), it might be hard to parallelize that part of the algorithm. A parallelization that is easier to implement would be to divide the computation of all $N_{\mathtt{init}}$ initial partitioning calculations over the different available processors. Furthermore, it is also easy to simply run the heuristic on each available processor with a different random seed, and return the lowest communication volume that was found by all processors. Although the computation time of this parallel heuristic will be identical to that of the sequential algorithm, the quality of

the solutions it produces will be higher, since it returns the lowest cost out of several runs of the sequential algorithm.

A MONDRIAAN 3.0 PARAMETER TUNING

When using the Mondriaan 3.0 matrix partitioning software [40, 6], it is possible to change several parameters that influence the execution of the program. The `Mondriaan.defaults` file, in which the values of the parameters are defined, lists 33 different parameters that can be changed. Explanations and possible values for all parameters are given in the User Guide, which is shipped along with the software. Although the parameters can heavily influence computation time *and* solution quality (as was experienced while using the software in this thesis), it is not easily understood how they interact, and what the *best* parameter settings are. In this chapter, an attempt is made to derive settings that produce the best matrix partitionings in reasonable time.

Some parameters can be easily set to the best option, either because they do not influence the matrix partitioning or because it is trivial to show that one option is the best. The following parameters are set in this way:

- `SplitStrategy`: Will be set to `hybrid` during the parameter tuning, as it produces the best results in practice. The `hybrid` option actually tries both the `finegrain` and `localbest` setting and returns the best out of these two.
- `SplitMethod`: The `simple` setting is only included for debugging purposes, so we will use the `KLFM` setting.
- `Partitioner`: Since we are only interested in the `Mondriaan` partitioner in this section, this parameter will be set to `mondriaan`.
- `Metric`: The $\lambda - 1$ metric is used throughout this thesis, so this parameter is set to `lambda1`.
- `DiscardFreeNets`: According to the User Guide, this option should be enabled whenever the cut-net metric is used.
- `Coarsening_MatchingStrategy`: The default setting `inprod` produces better solutions than the `random` setting.
- `VectorPartition_Step3`, `-_MaxNrLoops` and `-_MaxNrGreedyImproves`: These parameters influence the vector partitioning step, which is performed *after* the matrix partitioning. Since these parameters are not related to the communication cost of the matrix partitioning, we can leave them at their default settings.
- `Seed`: In order to obtain statistically relevant results, this parameter should be set to a random value for each run.
- `Permute`: This parameter dictates how the output matrix is permuted, according to its partitioning. As such, it has no influence on the communication cost, so we may as well leave it at its default setting.

After excluding these, 22 parameters remain that can possibly influence the quality of `Mondriaan 3.0`.

In principle, it is possible to find the optimal parameter settings by trying all possibilities of the *test domain* and remembering the best one. Since the parameters can interact with each other, we need

to try every *combination* of them. Let's label each parameter with an integer $\in \{0, 1, \ldots, N_{par} - 1\}$, where N_{par} is the number of parameters. If $q(i)$ is the number of different possible settings for parameter i, the total number of combinations N_{comb} is equal to:

$$N_{comb} = \prod_{i=0}^{N_{par}-1} q(i) \tag{A.1}$$

In the case of `Mondriaan`, even if every parameter can only be set to two different settings, this would result in 2^{22} combinations, which is more than 4 million possibilities. If one possibility would take 1 second to process, it would take almost 2 months to check them all. In practice, things are much worse, since processing one possibility will take much longer, as `Mondriaan` has to partition a set of large matrices over different numbers of processors. Furthermore, most parameters have more than 2 possible settings, so the actual value of N_{comb} will be much larger than 2^{22}. Therefore, simply trying every possible parameter combination will take an extremely long time, and a different method has to be used.

A.1 ORTHOGONAL ARRAY TESTING

Since not all possibilities will be checked, as explained above, it is impossible to find the optimal parameter setting. Therefore, the goal of this section is simply to find settings that produce higher quality solutions than the default settings. The main problem in parameter tuning is to choose which combinations of the test domain to check. A systemic way to generate a set of combinations to check is called *orthogonal array testing*. When using this method, combinations are generated orthogonally to each other, in the sense that they are statistically independent, and arranged in an array. An advantage of orthogonal array testing compared to other testing methods is that the entire test domain is uniformly covered by the generated combinations. Furthermore, all possible pairwise combinations of parameters are checked at least once by the orthogonal array testing method.

At [30], a large number of orthogonal arrays are given for various system arrangements. In this thesis, the $2^8 3^8 4^1 6^5 - 72$ array was chosen, which is a collection of 72 orthogonal combinations that can be used for testing a system of 22 parameters, with eight parameters of $q(i) = 2$, eight of $q(i) = 3$, one of $q(i) = 4$ and five of $q(i) = 6$. Although this array does not fit the `Mondriaan` system exactly, it is easily adaptable to it. The way the parameters of `Mondriaan` are implemented in the orthogonal array is shown in Table 6. Some parameters of `Mondriaan` are numerical, which means that they could be set to a large number of different settings. For these parameters, $q(i)$ is shown as # in the Table 6. Note that we are still able to implement these numerical parameters in the orthogonal array by defining a small number of different settings to test. In the case of `Coarsening_InprodScaling`, the default setting of `min` is duplicated in the orthogonal array in order to increase $q(i)$ from 5 to 6. The resulting orthogonal array is shown in Table 7. By using Table 7, we only have to check 72 combinations to obtain good information on what the optimal setting for `Mondriaan` may be.

For each combination of parameter settings, we run `Mondriaan 3.0` on several matrices, for $p \in \{2, 4, 8, 16, 32, 64\}$. The *score* of a single run is defined as the communication volume of that run, relative to the average communication volume of that run using the default settings of `Mondriaan 3.0`:

$$\text{score(matrix, } p \text{)} = \frac{V_{run}}{\langle V_{default} \rangle} \tag{A.2}$$

The total score of a parameter combination is equal to the average score over all matrices and p values. The best setting for the parameters is defined as the one with with the lowest total score.

Parameter	q(i)	Number of possibilities in orthogonal array
LoadBalanceStrategy	3	3
LoadBalanceAdjust	2	2
Alternate_FirstDirection	3	3
SquareMatrix_DistributeVectorsEqual	2	2
SquareMatrix_DistributeVectorsEqual_AddDummies	2	2
SymmetricMatrix_UseSingleEntry	2	2
SymmetricMatrix_SingleEntryType	2	2
Coarsening_NrVertices	#	6
Coarsening_MaxCoarsenings	#	3
Coarsening_MaxNrVtxInMatch	#	4
Coarsening_StopRatio	#	6
Coarsening_VtxMaxFractionOfWeight	#	6
Coarsening_InprodMatchingOrder	6	6
Coarsening_Netscaling	2	2
Coarsening_InprodScaling	5	6
Coarsening_MatchIdenticalFirst	2	2
Coarsening_FineSwitchLevel	2	2
KLFM_InitPart_NrRestarts	#	3
KLFM_InitPart_MaxNrLoops	#	3
KLFM_InitPart_MaxNrNoGainMoves	#	3
KLFM_Refine_MaxNrLoops	#	3
KLFM_Refine_MaxNrNoGainMoves	#	3

Table 6.: Implementation of the 22 Mondriaan settings in the $2^8 3^8 4^1 6^5 - 72$ orthogonal array from [30].

Test #	LoadBalanceAdjust	SquareMatrix_DistributeVectorsEqual	SquareMatrix_DistributeVectorsEqual_AddDummies	SymmetricMatrix_UseSingleEntry	SymmetricMatrix_SingleEntryType	Coarsening_Netscaling	Coarsening_MatchIdenticalFirst	Coarsening_FineSwitchLevel	LoadBalanceStrategy	Alternate_FirstDirection	Coarsening_MaxCoarsenings	KLFM_InitPart_NrRestarts	KLFM_InitPart_MaxNrLoops	KLFM_InitPart_MaxNrNoGainMoves	KLFM_Refine_MaxNrLoops	KLFM_Refine_MaxNrNoGainMoves	Coarsening_MaxNrVtxInMatch	Coarsening_NrVertices	Coarsening_StopRatio	Coarsening_VtxMaxFractionOfWeight	Coarsening_InprodMatchingOrder	Coarsening_InprodScaling
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	4	4	4	4	4	4
3	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	0	2	2	2	2	2
4	0	0	0	0	0	1	0	0	1	1	0	1	1	0	2	2	1	3	3	3	4	2	5
5	0	0	0	0	0	1	0	0	1	1	1	2	2	1	0	0	2	3	1	1	2	0	3
6	0	0	0	0	0	1	0	0	1	1	2	0	0	2	1	1	0	3	5	5	0	4	1
7	0	0	0	1	0	0	0	1	1	0	0	2	1	2	1	1	2	1	3	0	5	1	3
8	0	0	0	1	0	0	0	1	1	0	1	0	2	0	2	2	0	1	1	4	3	5	1
9	0	0	0	1	0	0	1	1	1	0	2	1	0	1	0	0	1	1	5	2	1	3	5
10	0	0	1	0	1	1	0	1	0	0	0	1	1	1	0	0	3	4	2	5	5	5	2
11	0	0	1	0	1	1	0	1	0	1	1	2	2	2	1	1	3	2	0	3	3	0	0
12	0	0	1	0	1	1	0	1	0	2	2	0	0	0	2	2	3	0	4	1	1	4	4
13	0	0	1	1	0	0	1	1	1	0	0	2	2	0	2	1	2	4	5	4	1	3	3
14	0	0	1	1	0	0	1	1	1	1	0	0	1	0	2	2	2	3	2	5	5	1	1
15	0	0	1	1	0	0	1	1	1	2	2	1	1	2	1	0	2	0	1	0	3	5	5
16	0	0	1	1	1	1	1	1	1	2	0	2	0	0	2	1	1	0	1	5	5	0	4
17	0	0	1	1	1	1	1	1	1	2	1	0	1	1	0	2	2	0	5	3	3	4	2
18	0	0	1	1	1	1	1	1	1	2	2	1	2	2	1	0	0	0	3	1	1	2	0
19	0	1	0	0	0	1	1	0	0	0	2	1	2	0	0	1	2	5	1	3	5	4	4
20	0	1	0	0	0	1	1	0	0	1	0	2	0	1	1	2	2	3	5	1	3	2	2
21	0	1	0	0	0	1	1	0	0	2	1	0	1	2	2	0	2	1	3	5	1	0	0
22	0	1	0	1	1	0	0	0	2	0	1	2	0	0	1	0	2	2	4	5	4	5	5
23	0	1	0	1	1	0	0	0	2	1	2	0	1	1	2	1	2	0	2	3	2	3	3
24	0	1	0	1	1	0	0	0	2	2	0	1	2	2	0	2	2	4	0	1	0	1	1
25	0	1	0	1	1	1	0	1	2	0	0	0	1	2	1	2	1	2	1	4	5	0	0
26	0	1	0	1	1	1	0	1	2	1	1	1	2	0	2	0	1	0	5	2	3	4	4
27	0	1	0	1	1	1	0	1	2	2	2	2	0	1	0	1	1	4	3	0	1	2	2
28	0	1	1	0	0	1	0	0	1	0	1	0	2	1	2	2	1	4	1	3	0	5	5
29	0	1	1	0	0	1	0	0	1	1	2	1	0	2	0	0	1	2	5	1	4	3	3
30	0	1	1	0	0	1	0	0	1	2	0	2	1	0	1	1	1	0	3	5	2	1	1
31	0	1	1	0	1	1	1	0	1	0	2	2	1	1	2	0	0	5	0	4	3	1	1
32	0	1	1	0	1	1	1	0	1	1	0	0	2	2	0	1	0	3	4	2	1	5	5
33	0	1	1	0	1	1	1	0	1	2	1	1	0	0	1	2	0	1	2	0	5	3	3
34	0	1	1	1	0	0	1	0	2	0	1	2	1	2	0	2	3	3	2	0	4	4	4
35	0	1	1	1	0	0	1	0	2	1	2	0	2	0	1	0	3	1	0	4	2	2	2
36	0	1	1	1	0	0	1	0	2	2	0	1	0	1	2	1	3	5	4	2	0	0	0

(a)

Test #	LoadBalanceAdjust	SquareMatrix_DistributeVectorsEqual	SquareMatrix_DistributeVectorsEqual_AddDummies	SymmetricMatrix_UseSingleEntry	SymmetricMatrix_SingleEntryType	Coarsening_Netscaling	Coarsening_MatchIdenticalFirst	Coarsening_FineSwitchLevel	LoadBalanceStrategy	Alternate_FirstDirection	Coarsening_MaxCoarsenings	KLFM_InitPart_NrRestarts	KLFM_InitPart_MaxNrLoops	KLFM_InitPart_MaxNrNoGainMoves	KLFM_Refine_MaxNrLoops	KLFM_Refine_MaxNrNoGainMoves	Coarsening_MaxNrVtxInMatch	Coarsening_NrVertices	Coarsening_StopRatio	Coarsening_VtxMaxFractionOfWeight	Coarsening_InprodMatchingOrder	Coarsening_InprodScaling
37	1	0	0	0	1	0	1	0	2	0	1	2	1	2	0	2	0	0	5	3	1	1	1
38	1	0	0	0	1	0	1	0	2	1	2	0	2	0	1	0	0	4	3	1	5	5	5
39	1	0	0	0	1	0	1	0	2	2	0	1	0	1	2	1	0	2	1	5	3	3	3
40	1	0	0	1	0	1	1	0	1	0	2	2	1	1	2	0	3	2	3	1	0	4	4
41	1	0	0	1	0	1	1	0	1	1	0	0	2	2	0	1	3	0	1	5	4	2	2
42	1	0	0	1	0	1	1	0	1	2	1	1	0	0	1	2	3	4	5	3	2	0	0
43	1	0	0	1	1	1	0	0	1	0	1	0	2	1	2	2	2	1	4	0	3	2	2
44	1	0	0	1	1	1	0	0	1	1	2	1	0	2	0	0	2	5	2	4	1	0	0
45	1	0	0	1	1	1	0	0	1	2	0	2	1	0	1	1	2	3	0	2	5	4	4
46	1	0	1	0	0	0	0	0	2	0	1	2	0	0	1	0	1	5	1	2	1	2	2
47	1	0	1	0	0	0	0	0	2	1	2	0	1	1	2	1	1	3	5	0	5	0	0
48	1	0	1	0	0	0	0	0	2	2	0	1	2	2	0	2	1	1	3	4	3	4	4
49	1	0	1	0	0	1	0	1	2	0	0	0	1	2	1	2	2	5	4	1	2	3	3
50	1	0	1	0	0	1	0	1	2	1	1	1	2	0	2	0	2	3	2	5	0	1	1
51	1	0	1	0	0	1	0	1	2	2	2	2	0	1	0	1	2	1	0	3	4	5	5
52	1	0	1	1	1	1	1	1	0	0	0	2	1	2	0	0	1	1	2	4	0	2	1
53	1	0	1	1	1	1	1	1	0	0	1	0	2	0	1	1	2	1	0	2	4	0	5
54	1	0	1	1	1	1	1	1	0	0	2	1	0	1	2	2	0	1	4	0	2	4	3
55	1	1	0	0	0	1	1	1	2	0	2	0	0	2	1	1	3	4	2	2	3	1	1
56	1	1	0	0	0	1	1	1	2	1	0	1	1	0	2	2	3	2	0	0	1	5	5
57	1	1	0	0	0	1	1	1	2	2	1	2	2	1	0	0	3	0	4	4	5	3	3
58	1	1	0	0	1	0	1	1	1	0	0	2	2	0	2	1	1	1	2	1	4	0	0
59	1	1	0	0	1	0	1	1	1	1	1	0	0	1	0	2	1	5	0	5	2	4	4
60	1	1	0	0	1	0	1	1	1	2	2	1	1	2	1	0	1	3	4	3	0	2	2
61	1	1	0	1	0	1	0	1	0	0	0	1	1	1	0	0	0	1	5	2	2	5	5
62	1	1	0	1	0	1	0	1	0	1	1	2	2	2	1	1	0	5	3	0	0	3	3
63	1	1	0	1	0	1	0	1	0	2	2	0	0	0	2	2	0	3	1	4	4	1	1
64	1	1	1	0	1	0	1	1	0	0	2	1	2	1	1	2	2	0	3	2	4	0	0
65	1	1	1	0	1	0	1	1	0	1	0	2	0	2	2	0	2	4	1	0	2	4	4
66	1	1	1	0	1	0	1	1	0	2	1	0	1	0	0	1	2	2	5	4	0	2	2
67	1	1	1	1	0	0	0	1	1	0	1	1	0	2	2	1	0	0	0	1	5	2	2
68	1	1	1	1	0	0	0	1	1	1	2	2	1	0	0	2	0	4	4	5	3	0	0
69	1	1	1	1	0	0	0	1	1	2	0	0	2	1	1	0	0	2	2	3	1	4	4
70	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	3	3	3	3	3	3	3
71	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	3	1	1	1	1	1	1
72	1	1	1	1	1	0	0	0	0	2	2	2	2	2	2	2	3	5	5	5	5	5	5

(b)

Table 7.: The orthogonal testing array used in Appendix A.

Matrix	Columns	Rows	Nonzeroes	Symmetric
lp_cre_b	9648	77137	260785	no
lp_nug30	52260	379350	1567800	no
lp_dfl001	6071	12230	35632	no
tbdmatlab	19859	5979	430171	no
onetone2	36057	36057	227628	no
bcsstk30	28924	28924	2043492	yes
cage10	11397	11397	150645	pattern
bcsstk32	44609	44609	2014701	yes
finan512	74752	74752	596992	yes
gemat11	4929	4929	33185	no
lhr34	35152	35152	764014	no
memplus	17758	17758	126150	pattern

Table 8.: Test matrices used for tuning the parameters of Mondriaan 3.0. Pattern symmetry occurs when the nonzero values themselves are not symmetric, but the nonzero pattern is.

After running the orthogonal array testing, a final *local improvement* step was applied. In this step, we take the best settings from the orthogonal array testing, change one parameter to a different setting, and measure the total score. After doing this for all parameters and possible settings, we take the change that had the best total score and use this as the new best parameter setting. This process is repeated until no improving changes are found. Note that this can also happen in the first iteration, implying that the orthogonal array solution was already locally optimal.

A.2 RESULTS

The parameter tuning procedure described above was applied to a selection of test matrices, which is shown in Table 8. This test set is almost identical to that of Section 3.7, except for the removal of the polyDFT, tbdlinux and c98a matrix and the addition of lp_nug30. These changes were made in order to reduce the computation time of a single run. It should be noted that the results of the parameter tuning can depend on which matrices are included in the test set. Therefore, the best settings that are found should be interpreted as the best settings for solving the matrices of this test set, although we hope that these settings are good for solving other matrices as well. During orthogonal array testing, it was noted that most parameter setting combinations produced results that were much worse than the default results ($\langle score \rangle \gg 1$), indicating that the default settings are already close to optimal.

The best parameter settings that were found are shown in Table 9. The average score of these settings is equal to 0.896. This means that, on average, these settings produce a solution that has a communication volume that is 10.4% smaller than that of the default settings. In the non-numerical parameters, the biggest deviation from the default values is that SymmetricMatrix_UseSingleEntry is set to yes, with the related parameter SymmetricMatrix_SingleEntryType set to lower. With these parameters enabled, Mondriaan uses only the lower triangular part of a symmetric matrix during partitioning. For the matrices of the test set, this turned out to lower the resulting communication volumes significantly.

For the numerical parameters, it was noticed that increasing the values of KLFM_InitPart_NrRestarts and KLFM_InitPart_MaxNrNoGainMoves had a large positive impact on the resulting

solution quality. Apparently, it is important to obtain a high quality initial partitioning of the coarsened matrix in order to end up with a good partitioning of the input matrix. Therefore, it might be wise to invest more time in finding good initial partitionings in future versions of `Mondriaan`. For example, a greedy heuristic similar to the one used in the heuristic of Chapter 3 could improve the solution quality of `Mondriaan`.

In Table 10, results from the new settings for `Mondriaan 3.0` are compared with the default settings, on the same matrix test set as in Section 3.7. For each matrix and p value, `Mondriaan` was run 10 times, and the lowest volume out of all runs is shown. The new settings were able to produce the best solution in 75% of all cases. For some matrices, the improvement on the default settings was significant, such as the 31.2% improvement for the `memplus` matrix and $p = 2$.

Parameter	Value
LoadbalanceAdjust	yes
SquareMatrix_DistributeVectorsEqual	no
SquareMatrix_DistributeVectorsEqual_AddDummies	no
SymmetricMatrix_UseSingleEntry	yes
SymmetricMatrix_SingleEntryType	lower
Coarsening_NetScaling	linear
Coarsening_MatchIdenticalFirst	yes
Coarsening_FineSwitchLevel	0
LoadbalanceStrategy	constant
Alternate_FirstDirection	col
Coarsening_MaxCoarsenings	256
KLFM_InitPart_NrRestarts	32
KLFM_InitPart_MaxNrLoops	25
KLFM_InitPart_MaxNrNoGainMoves	800
KLFM_Refine_MaxNrLoops	100
KLFM_Refine_MaxNrNoGainMoves	800
Coarsening_MaxNrVtxInMatch	2
Coarsening_NrVertices	400
Coarsening_StopRatio	0.025
Coarsening_VtxMaxFractionOfWeight	0.3
Coarsening_InprodMatchingOrder	natural
Coarsening_InprodScaling	min
SplitStrategy	hybrid
SplitMethod	KLFM
Partitioner	mondriaan
Metric	lambda1
Coarsening_MatchingStrategy	inproduct
VectorPartition_Step3	random
VectorPartition_MaxNrLoops	10
VectorPartition_MaxNrGreedyImproves	10
Permute	none
Seed	random

Table 9.: Best `Mondriaan 3.0` parameter settings, obtained by the tuning procedure of Appendix A.

Matrix	p	Default	Tuned	Matrix	p	Default	Tuned
c98a	2	44787	**44615**	lp_cre_b	2	548	**438**
	4	100023	**95082**		4	1244	**1062**
	8	164890	**163675**		8	2370	**2090**
	16	**237416**	243086		16	3967	**3444**
	32	**332549**	346014		32	6235	**5539**
	64	**444065**	458919		64	9121	**8182**
polyDFT	2	**3456**	**3456**	lp_dfl001	2	**573**	588
	4	8472	**8448**		4	1360	**1349**
	8	18516	**18432**		8	**2407**	2444
	16	37078	**36096**		16	**3534**	3540
	32	53881	**52002**		32	4741	**4703**
	64	74036	**70879**		64	5908	**5853**
tbdmatlab	2	4698	**4694**	onetone2	2	**204**	256
	4	**10101**	10701		4	**658**	812
	8	**16705**	16956		8	**1524**	1638
	16	**25062**	26066		16	2590	**2554**
	32	**36900**	44274		32	3672	**3519**
	64	**53598**	60103		64	5364	**4777**
tbdlinux	2	13212	**12637**	bcsstk30	2	**535**	658
	4	33795	**32899**		4	1562	**1536**
	8	**54101**	5843		8	3566	**3448**
	16	**76432**	83658		16	7771	**6826**
	32	**113378**	134407		32	13141	**11840**
	64	**162242**	178157		64	21326	**18870**
cage10	2	**1765**	1787	bcsstk32	2	**788**	828
	4	3798	**3683**		4	1845	**1662**
	8	5782	**5572**		8	3696	**3198**
	16	8272	**7969**		16	6461	**5708**
	32	11255	**11040**		32	10780	**9801**
	64	15595	**14756**		64	17102	**16120**
finan512	2	106	**100**	gemat11	2	34	**32**
	4	233	**200**		4	71	**68**
	8	486	**400**		8	163	**150**
	16	978	**800**		16	345	**302**
	32	1809	**1600**		32	588	**529**
	64	7706	**6400**		64	1042	**918**
lhr34	2	**64**	**64**	memplus	2	554	**381**
	4	327	**321**		4	1296	**859**
	8	800	**731**		8	2436	**1581**
	16	1532	**1389**		16	3843	**2624**
	32	3127	**2857**		32	5828	**4169**
	64	5714	**5388**		64	7999	**5973**

Table 10.: Comparison of the lowest communication volume found in 10 runs for `Mondriaan 3.0 hybrid`, with the default settings and the best settings found by the tuning procedure of Appendix A. Bold values depict the lowest of all communication volumes that were found for that matrix and p value.

B BENCHMARK RESULTS

Table 11.: Benchmark results for all matrices of the University of Florida Matrix Collection [12] with no more than 200 rows or columns. Given are the matrix name and dimensions, and the communication volume of the optimal two-way partitioning, obeying the strict load imbalance equation 2.1 with $\varepsilon = 0.03$. For some matrices, computation time of the optimal communication volume exceeded 1 day, and for these matrices, the best known communication volume is given instead, and a star (*) is added to its volume to indicate that it may not be optimal.

Matrix	Rows	Columns	Nonzeroes	Optimal volume
Trec3	1	2	1	0
Trec4	2	3	3	1
GL7d10	1	60	8	1
Trec5	3	7	12	2
b1_ss	7	7	15	3
ch3-3-b2	6	18	18	0
rel3	12	5	18	3
cage3	5	5	19	4
lpi_galenet	8	14	22	2
relat3	12	5	24	3
lpi_itest2	9	13	26	3
lpi_itest6	11	17	29	2
Tina_AskCal	11	11	29	3
n3c4-b1	15	6	30	5
n3c4-b4	6	15	30	5
ch3-3-b1	18	9	36	5
Tina_AskCog	11	11	36	4
GD01_b	18	18	37	1
Trec6	6	15	40	5
farm	7	17	41	4
Tina_DisCal	11	11	41	5
kleemin	8	16	44	6
LFAT5	14	14	46	4
Tina_DisCog	11	11	48	6
bcsstm01	48	48	48	0
cage4	9	9	49	9
jgl009	9	9	50	5
GD98_a	38	38	50	0
GD95_a	36	36	57	1
n3c4-b2	20	15	60	9
n3c4-b3	15	20	60	9
klein-b1	30	10	60	5

Continued on next page

Table 11 – continued from previous page

Matrix	Rows	Columns	Nonzeroes	Optimal volume
klein-b2	20	30	60	6
Ragusa18	23	23	64	5
bcsstm02	66	66	66	0
lpi_bgprtr	20	40	70	4
wheel_3_1	21	25	74	8
jgl011	11	11	76	7
rgg010	10	10	76	9
Ragusa16	24	24	81	7
LF10	18	18	82	4
problem	12	46	86	2
GD02_a	23	23	87	7
Stranke94	10	10	90	10
n3c5-b1	45	10	90	8
GD95_b	73	73	96	2
ch4-4-b3	24	96	96	0
Hamrle1	32	32	98	5
lp_afiro	27	51	102	5
rel4	66	12	104	5
bcsstm03	112	112	112	0
p0033	15	48	113	5
football	35	35	118	8
n4c5-b11	10	120	120	0
wheel_4_1	36	41	122	12
GlossGT	72	72	122	5
ibm32	32	32	126	13
bcspwr01	39	39	131	6
bcsstm04	132	132	132	0
p0040	23	63	133	3
GD01_c	33	33	135	7
bcsstm22	138	138	138	0
lpi_woodinfe	35	89	140	0
ch4-4-b1	72	16	144	11
Trefethen_20b	19	19	147	16
Trec7	11	36	147	8
lp_sc50b	50	78	148	5
d_ss	53	53	149	4
GD99_c	105	105	149	0
refine	29	62	153	3
bcsstm05	153	153	153	0
Trefethen_20	20	20	158	17
can_24	24	24	160	8
lp_sc50a	50	78	160	5
bcspwr02	49	49	167	4
lap_25	25	25	169	10
relat4	66	12	172	4

Continued on next page

Table 11 – continued from previous page

Matrix	Rows	Columns	Nonzeroes	Optimal volume
pores_1	30	30	180	9
wheel_5_1	57	61	182	13*
GD96_b	111	111	193	3
GD98_b	121	121	207	0
n3c6-b1	105	105	210	11
n2c6-b1	105	15	210	11
n4c5-b1	105	15	210	11
can_62	62	62	218	6
dwt_72	72	72	222	4
divorce	50	9	225	8
GD96_d	180	180	229	0
GD02_b	80	80	232	5
cage5	37	37	233	14
Maragal_1	32	14	234	14
d_dyn1	87	87	238	5
d_dyn	87	87	238	5
n3c5-b7	30	120	240	14
lpi_forest6	66	131	246	5
Sandi_authors	86	86	248	4
GD96_c	65	65	250	5
IG5-6	30	77	251	18
GD99_b	64	64	252	20*
bibd_9_3	36	84	252	22*
cat_ears_2_1	85	85	254	14*
wheel_6_1	83	85	254	15*
GD97_b	47	47	264	11
dwt_59	59	59	267	8
ex5	27	27	279	10
will57	57	57	281	4
GD95_c	62	62	287	6
ch4-4-b2	96	72	288	24*
curtis54	54	54	291	7
west0067	67	67	294	12
mesh1e1	48	48	306	18
mesh1em1	48	48	306	18
mesh1em6	48	48	306	18
pivtol	102	102	306	4
impcol_b	59	59	312	10
lp_kb2	43	68	313	14
dwt_66	66	66	320	4
GD97_a	84	84	332	24*
GD98_c	112	112	336	16*
wheel_7_1	114	113	338	17*
lp_sc105	105	163	340	5
bfwb62	62	62	342	8

Continued on next page

Table 11 – continued from previous page

Matrix	Rows	Columns	Nonzeroes	Optimal volume
n3c5-b2	120	45	360	25*
west0156	156	156	371	5
can_73	73	73	377	18
GD06_theory	101	101	380	0
flower_4_1	121	129	386	23*
olm100	100	100	396	2
tub100	100	100	396	4
bcsstk01	48	48	400	24
ch5-5-b1	200	25	400	19*
gams10am	114	171	407	2
gams10a	114	171	407	2
impcol_c	137	137	411	14*
west0132	132	132	414	6
gre_115	115	115	421	22*
lp_adlittle	56	138	424	9
sphere2	66	66	450	24*
bfwa62	62	62	450	11
lp_scagr7	129	185	465	6
bcspwr03	118	118	476	8
rw136	136	136	479	19*
lp_stocfor1	117	165	501	9
west0167	167	167	507	7
lp_blend	74	114	522	11
ash85	85	85	523	14
lns_131	131	131	536	22*
lnsp_131	131	131	536	22*
dwt_87	87	87	541	12
Trec8	23	84	549	17
IG5-7	62	150	549	26
can_61	61	61	557	10
nos4	100	100	594	14*
TF10	99	107	622	33*
bcsstk03	112	112	640	0
gent113	113	113	655	17
zed	116	142	666	10
bcsstk22	138	138	696	8*
will199	199	199	701	14*
rotor1	100	100	708	14
lpi_klein1	54	108	750	33
can_96	96	96	768	24*
lp_share2b	96	162	777	9*
cage6	93	93	785	38*
bwm200	200	200	796	4
rajat11	135	135	812	8
robot	120	120	870	12

Continued on next page

Table 11 – continued from previous page

Matrix	Rows	Columns	Nonzeroes	Optimal volume
steam3	80	80	928	8
lop163	163	163	935	16*
ck104	104	104	992	0
gre_185	185	185	1005	37*
CAG_mat72	72	72	1012	13
fs_183_1	183	183	1069	24*
fs_183_3	183	183	1069	23*
fs_183_4	183	183	1069	24*
fs_183_6	183	183	1069	24*
rdb200l	200	200	1120	38*
rdb200	200	200	1120	38*
dwt_162	162	162	1182	14*
bibd_9_5	36	126	1260	34*
arc130	130	130	1282	13
can_144	144	144	1296	12
Cities	55	46	1342	38
can_161	161	161	1377	32*
dwt_198	198	198	1392	12*
can_187	187	187	1491	22*
rajat14	180	180	1503	10
tols90	90	90	1746	18
Trefethen_150	150	150	2040	114*
bcsstk05	153	153	2423	24*
lund_b	147	147	2441	41*
lund_a	147	147	2449	41*
mcca	180	180	2659	38*
Trefethen_200b	199	199	2873	145*
Trefethen_200	200	200	2890	146*
dwt_193	193	193	3493	40*
bcsstk04	132	132	3648	48
bcsstk02	66	66	4356	66*
Journals	124	124	12068	124*

BIBLIOGRAPHY

[1] IBM ILOG CPLEX 12.1. http://www.ilog.com/products/cplex/. (Cited on pages 21 and 26.)

[2] GNU Compiler Collection, April 2010. http://gcc.gnu.org/. (Cited on pages 18 and 38.)

[3] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: a survey. *Integration, the VLSI Journal*, 19, 1995. (Cited on page 5.)

[4] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica*, 1998. (Cited on page 21.)

[5] S. Areibi and A. Vannelli. Advanced search techniques for circuit partitioning. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 77–98, 1993. (Cited on page 25.)

[6] R. H. Bisseling. Mondriaan 2.01 software. http://www.math.uu.nl/people/bisseling/Mondriaan/mondriaan.html. (Cited on page 55.)

[7] R. H. Bisseling. *Parallel Scientific Computing - A structured approach using BSP and MPI*. Oxford University Press, 2004. (Cited on pages 1 and 2.)

[8] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999. (Cited on pages 3 and 4.)

[9] Ü. V. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *Proceedings Eighth International Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 2001)*, page 118. IEEE Press, Los Alamitos, CA, 2001. (Cited on pages 3 and 4.)

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992. (Cited on pages 1 and 15.)

[11] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963. (Cited on page 20.)

[12] T. A. Davis. The University of Florida Sparse Matrix Collection. Submitted to ACM Transactions on Mathematical Software, March 2010. http://www.cise.ufl.edu/research/sparse/matrices/. (Cited on pages iii, 1, 6, 8, 27, 28, 40, 47, 50, and 63.)

[13] W. E. Donath and A. J. Hoffman. Lower Bounds for the Partitioning of Graphs. *IBM Journal of Research and Development*, 17(5):420–425, September 1973. (Cited on page 36.)

[14] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, 1989. (Cited on page 6.)

[15] J. Falkner, F. Rendl, and H. Wolkowicz. A computational study of graph partitioning. *Mathematical Programming*, 66(1-3):211–239, August 1994. (Cited on pages 35 and 36.)

[16] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference, IEEE*, pages 175–181, 1982. (Cited on pages 5 and 43.)

[17] E. Gabriel et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI users' group meeting*, 2004. (Cited on page 18.)

[18] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. (Cited on page 5.)

[19] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958. (Cited on page 21.)

[20] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1999. (Cited on page 18.)

[21] M. Grötschel and O. Holland. Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming*, 51(1-3):141–202, 1991. (Cited on page 21.)

[22] S. W. Hadley, B. L. Mark, and A. Vannelli. An Efficient Eigenvector Approach for Finding Netlist Partitions. *IEEE Transactions on Computer-aided Design*, 11(1), July 1992. (Cited on pages 33 and 34.)

[23] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 28, New York, NY, USA, 1995. ACM. (Cited on page 6.)

[24] Y. Hu. Efficient, high-Quality force-Directed graph drawing. *Mathematica Journal*, 2006. (Cited on page 40.)

[25] W. N. A. Jr. and T. D. Morley. Eigenvalues of the laplacian of a graph. *Linear and Multilinear Algebra*, 18:141–145, October 1985. (Cited on page 36.)

[26] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984. (Cited on page 20.)

[27] G. Karypis and V. Kumar. hMetis 1.5: A hypergraph partitioning package, 1998. Technical report, Department of Computer Science, University of Minnesota. (Cited on pages 5, 41, and 44.)

[28] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49:291–307, 1970. (Cited on pages 5 and 43.)

[29] J. B. Kruskal. On the shortest spanning subtree of a graph and the travelling salesman problem. In *Proceedings of the American Mathematical Society*, volume 7, pages 48–50, February 1956. (Cited on page 43.)

[30] W. F. Kuhfeld. Orthogonal arrays. `http://support.sas.com/techsup/technote/ts723.html`. (Cited on pages 56 and 57.)

[31] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, 1998. (Cited on page 38.)

[32] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*. Wiley, 1988. (Cited on page 21.)

[33] P. M. Pardalos and J. B. Rosen. Methods for global concave minimization: A bibliographic survey. *SIAM Review*, 28(3):367–379, 1986. (Cited on page 38.)

[34] H. D. Simon and S.-H. Teng. How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, September 1997. (Cited on page 6.)

[35] E.-G. Talbi, editor. *Parallel Combinatorial Optimization*, chapter 1. Wiley-Interscience, 2006. (Cited on page 16.)

[36] P. Toth and D. Vigo. *The Vehicle Routing Problem*. Society for Industrial and Applied Mathematics, 2001. (Cited on page 21.)

[37] B. Uçar, Ümit V. Çatalyürek, and C. Aykanat. A matrix partitioning interface to PaToH in MATLAB. *Parallel Computing*, 36:254–272, 2010. (Cited on page 45.)

[38] Ümit V. Çatalyürek and C. Aykanat. PaToH: Partitioning tool for hypergraphs, version 3.0, 1999. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. (Cited on pages 5, 41, and 44.)

[39] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–11, 1990a. (Cited on pages 1 and 18.)

[40] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005. (Cited on pages iii, 5, 33, 41, 44, and 55.)

[41] M. G. Wrighton and A. M. DeHon. Sat-based optimal hypergraph partitioning with replication. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 789 – 795, 2006. (Cited on page 5.)