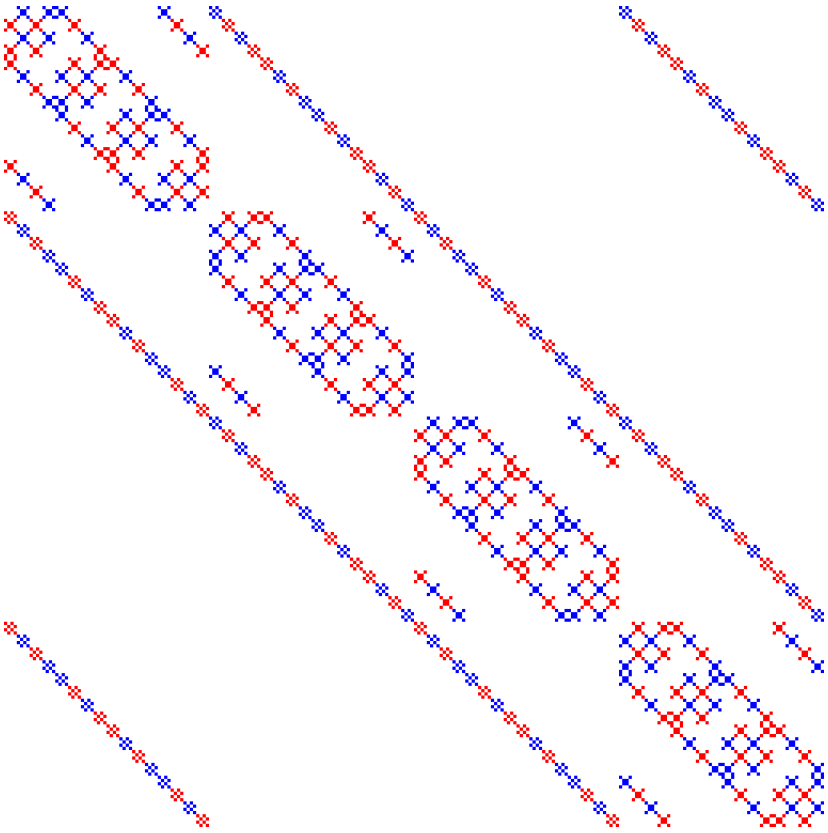
Accelerating the Mondriaan sparse matrix partitioning package

Author: Marco van Oort
Supervisor: Rob H. Bisseling



Utrecht University

May 24, 2017

Abstract

We consider a number of modifications to the Mondriaan package for sparse matrix partitioning, with as main goal to improve run times of the software. Two new algorithms are considered, one of which aims at reducing load imbalance to enable Mondriaan to find partitionings with less communication volume, while the other aims at finding solutions with zero communication volume more quickly than Mondriaan currently does. The other modifications include an improved sorting function, an improvement in a data structure used in a core component of Mondriaan, and two modifications in the PGA matching algorithm implementation used in Mondriaan. We compare the performance of each modification against the previous version, Mondriaan 4.1, and we will discuss which modifications are suitable to include into a new version, Mondriaan 4.2. Experiments with a candidate version for Mondriaan 4.2 show a mean run time improvement of 25% compared to Mondriaan 4.1, while also significantly reducing the mean load imbalance and slightly reducing the mean communication volume.

I would like to thank Rob Bisseling for his guidance on this thesis.

Titlepage figure: zero volume bipartitioning (see Section 5) of the top-left 768×768-submatrix of the `QCD/conf5_0-4x4-14` matrix[8].

Contents

1 Introduction

Matrices play a key role in many mathematical areas. In particular considering large sparse matrices, one wants to use efficient methods to perform calculations on these matrices. Most of these methods, for example for calculating eigenvalues, eigenvectors or inverse mappings, involve iterative procedures using successive multiplications of vectors with a mostly fixed matrix. In other words, the matrix-vector multiplication is at the heart of many matrix manipulations.

When matrices tend to become really large, in a way they cannot be stored on a single physical computer any more, one has to make a switch from sequential to parallel algorithms on matrices, spreading the matrix over multiple processing units. While sequential algorithms are relatively portable by their nature, parallel algorithms are more dependent on a systems specific architecture, like the number of processing units it can use and how fast communication is between these processing units. To abstract this machine architecture, Valiant [27] proposed the BSP (Bulk Synchronous Parallel) model, providing a framework upon which a parallel algorithm can be defined.

In [29], Bisseling and Vastenhouw developed a software package named Mondriaan [4], which takes as input a matrix and a number of processors, and computes a partitioning of the nonzeros of this matrix over this number of processors. This is done in such a way that, in a matrix-vector multiplication based on the BSP model, each processor has to perform an approximately equal amount of work (scalar multiplications), and the overall communication between processors is kept at a minimum.

Since the initial release in 2002, many improvements have been made to the Mondriaan software. In 2005, improvements were made in the vector partitioning part of the algorithm [6], and in 2006 [28] column scaling was introduced to improve the matching in the coarsening phase and the fine-grain method from [33] was implemented to facilitate a hybrid method [5] between this method and the original Mondriaan method. In 2009 and 2011 improvements were made by permuting the input matrix such that the resulting matrix-vector multiplication makes better use of caching [31][32]. In 2013, a new metric was introduced for use with finite element triangulations [13] and in 2014 a new medium-grain method was introduced, which currently is the default setting [21]. Also in 2014, new methods were investigated for matching and neighbour-finding in the coarsening phase of the algorithm [2]. Last but not least, in 2015 an algorithm was developed for solving the bipartitioning problem exactly, under the name MondriaanOpt [22].

It is the purpose of this thesis to elaborate on these developments by accelerating some of the core parts of the software package, and include some other improvements that may either speed up the algorithm or improve the solution quality.

This introduction consists of three parts. First, I will describe what the exact problem is we are looking at, secondly I will describe some of the earlier work and achievements within this field and thirdly we look at what the current state of the Mondriaan software is. Then the subsequent sections of this thesis will deal with the considered potential improvements to the package. In order, these are:

- improving the sorting algorithms used in Mondriaan, by removing recursion and using 3-way quicksort instead of the 2-way variant,
- a new implementation of the `GainBucket` structure in Mondriaan which is used in the Kernighan-Lin / Fiduccia-Mattheyses (KLFM) algorithm,
- detecting connected components in the matrix, in order to find potential partitionings with zero volume faster,
- using free nonzeros to improve the balance of work of a partitioning, to enlarge the search subspace in subsequent iterations which may in turn lead to reduced amounts of communication,
- a speed improvement to the Path Growing Algorithm (PGA) used in Mondriaan,
- and generalizing the PGA algorithm to coarsen hypergraphs faster.

1.1 A parallel sparse matrix-vector multiplication algorithm

Consider the multiplication $u = Av$ of an $m \times n$ matrix A with a vector v, with u, A and v distributed over P processors. The particular algorithm we will be using is the following.

1. Each component v_j is known to exactly one processor. In this phase, each processor determines which components v_j it will be needing in the next phase (i.e. it determines in which columns of A it owns elements), and gathers these components v_j from the processor that owns them. This phase is called the fanout phase.

2. Each processor $s \in [0, P-1]$ determines its contribution u_{is} to the vector component u_i, which is the sum of $a_{ij}v_j$ over all nonzeros a_{ij} that processor s owns in row i.

3. Each component u_i belongs to exactly one processor, hence each processor s sends its contributions u_{is} to the processor owning u_i. Note that it only does that whenever $u_{is} \neq 0$. This phase is called the fanin phase.

4. Each processor determines the final value of their components u_i from all received contributions, $u_i = \sum_{t=0}^{P-1} u_{it}$.

To derive a formula for the total cost of this algorithm, consider the following. In the BSP model, each of the phases above is called a superstep. Phases one and three are communication supersteps, phases two and four are computation supersteps. After each superstep a synchronization between processors takes place, leading to some waiting time (latency) l. In the above algorithm, this sums up to $4l$ costs. Furthermore, each communicated value incurs a communication cost g. In the fanout phase, denote the maximum number of received components v_j per processor by $n_{rec,out}$ and the maximum number of sent components by $n_{sent,out}$. Likewise, denote the maximum number of received and sent components u_{is} in the fanin phase with $n_{rec,in}$ and $n_{sent,in}$. Then the total communication cost is given by $(\max\{n_{rec,out}, n_{sent,out}\} + \max\{n_{rec,in}, n_{sent,in}\})g$. Lastly, let the maximum number of nonzeros of a processor be given by nnz_{max}, then the cost of phase two can be bounded by $2nnz_{max}$, composed of one multiplication and one addition per nonzero. The cost of phase four can be bounded by $n_{rec,in}$ as each processor performs an addition for each of the $n_{rec,in}$ received contributions. In total, the costs are bounded by

$$T \leq 2nnz_{max} + n_{rec,in} + (\max\{n_{rec,out}, n_{sent,out}\} + \max\{n_{rec,in}, n_{sent,in}\})g + 4l. \tag{1}$$

From this, we can deduce that an optimal partitioning should obey the following.

1. It should minimize nnz_{max}, the maximum number of nonzeros per part. In other words, it should divide the work evenly.

2. It should minimize $\max\{n_{rec,out}, n_{sent,out}\}$ and $\max\{n_{rec,in}, n_{sent,in}\}$, in other words, it should minimize the communication (and the number of additions in phase four, but these are the least of our concerns here, as additions are relatively cheap).

The following definitions and theorem will be important.

Definition 1.1 ([29, Def. 2.1])**.** *Let A be an $m \times n$ sparse matrix and let $A_0, \ldots, A_{k-1}$ be mutually disjoint subsets of A, where $k \geq 1$. Define λ_i as the number of subsets that have a nonzero in row $i \in [0, m-1]$ of A:*

$$\lambda_i = \lambda_i(A_0, \ldots, A_{k-1}) = |\{r \in [0, k-1] : \exists j \in [0, n-1] \text{ s.t. } a_{ij} = 1 \wedge (i,j) \in A_r\}|, \tag{2}$$

and define μ_j as the number of subsets that have a nonzero in column $j \in [0, n-1]$ of A:

$$\mu_j = \mu_j(A_0, \ldots, A_{k-1}) = |\{r \in [0, k-1] : \exists i \in [0, m-1] \text{ s.t. } a_{ij} = 1 \wedge (i,j) \in A_r\}|. \tag{3}$$

Define $\lambda'_i = \max(\lambda_i - 1, 0)$ *and* $\mu'_j = \max(\mu_j - 1, 0)$. *Then the communication volume for the subsets* $A_0, \ldots, A_{k-1}$ *is defined as*

$$V(A_0, \ldots, A_{k-1}) = \sum_{i=0}^{m-1} \lambda'_i + \sum_{j=0}^{n-1} \mu'_j. \tag{4}$$

We can interpret this definition as follows. Each component v_j needs to be sent to each processor (subset) that owns a part of column j, hence this introduces μ_j communication. However, note that sending a value from a processor to itself does not incur communication. Hence in a good partitioning, we may assume that v_j is assigned to one of the processors that contains an element in column j, because if not, it would introduce one extra communication. Hence the total communication volume of the fan-out is $\sum \mu'_j$, and a similar reasoning holds for the fan-in. This definition leads to the following theorem:

Theorem 1.2 ([29, Thm. 2.2])**.** *Let* A *be an* $m \times n$ *matrix and let* $A_0, \ldots, A_{k-1}$ *be mutually disjoint subsets of* A, *where* $k \geq 2$. *Then*

$$V(A_0, \ldots, A_{k-1}) = V(A_0, \ldots, A_{k-3}, A_{k-2} \cup A_{k-1}) + V(A_{k-2}, A_{k-1}). \tag{5}$$

This theorem tells us that when partitioning, the extra volume incurred by splitting a subset can be calculated from the bipartitioning of this subset only: we do not have to reconsider all other subsets.

We denote the number of nonzeros in a matrix A, or a subset of nonzeros A_r thereof, by $nnz(A)$ and $nnz(A_r)$, respectively.

Definition 1.3 ([29, Def. 2.3])**.** *Let* A *be an* $m \times n$ *matrix and let* $A_0, \ldots, A_{k-1}$ *be mutually disjoint subsets of* A, *where* $k \geq 1$. *Then the maximum amount of computation work for the subsets* $A_0, \ldots, A_{k-1}$ *is*

$$W(A_0, \ldots, A_{k-1}) = \max_{r \in [0,k-1]} nnz(A_r). \tag{6}$$

Now we are ready to formulate the problem at hand:

Problem 1.4. *Given an* $m \times n$ *matrix* A *with* $nnz(A)$ *nonzero elements, a number of processors (or parts)* P *and a load imbalance parameter* $\epsilon \geq 0$, *determine a partitioning* $A_0, \ldots, A_{P-1}$ *of the nonzeros of* A *over the* P *parts that minimizes the total communication volume* $V(A_0, \ldots, A_{P-1})$ *among all partitionings that satisfy*

$$W(A_0, \ldots, A_{P-1}) \leq (1 + \epsilon) \frac{nnz(A)}{P}. \tag{7}$$

Note that this problem does indeed try to keep load imbalance low, as we wished, but it does not exactly minimize the maximum communication per processor in the fan-out and fan-in phases. Instead, it minimizes the total communication volume. While a (near-)optimal total communication volume may not imply a (near-)optimal maximum communication cost in either the fan-in or fan-out phase, it suffices to say that a small total communication volume does impose an upper bound on the maximum communication cost. As calculations on the total volume are easy by the use of Theorem 1.2, this is the objective function of our choice here.

Lastly we consider the following lemma:

Lemma 1.5. *Given a matrix* A, *Problem 1.4 is feasible if and only if*

$$\left\lceil \frac{nnz(A)}{P} \right\rceil \leq \left\lfloor (1 + \epsilon) \frac{nnz(A)}{P} \right\rfloor. \tag{8}$$

Proof. Note that Equation (7) is equivalent with

$$W(A_0, \ldots, A_{P-1}) \leq \left\lfloor (1+\epsilon)\frac{nnz(A)}{P} \right\rfloor, \tag{9}$$

as partitions can not contain a fractional number of nonzeros. Hence the problem is feasible if and only if Equation (9) is true.

Assume we have a feasible partitioning. Some partitions may contain $\lfloor \frac{nnz(A)}{P} \rfloor$ nonzeros or less, but at least one partition should contain $\lceil \frac{nnz(A)}{P} \rceil$ nonzeros or more, hence $W(A_0, \ldots, A_{P-1}) \geq \lceil \frac{nnz(A)}{P} \rceil$. Combining this with Equation (9), we obtain Equation (8).

Now assume Equation (8) holds. Then we can distribute all nonzeros over the partitions such that we have

$$W(A_i) \leq \left\lceil \frac{nnz(A)}{P} \right\rceil \qquad \forall i \in [0, P-1]. \tag{10}$$

Note that indeed, all nonzeros can be assigned to some partition this way, as the maximum number of nonzeros the matrix may consist of to be able to do this equals $\lceil \frac{nnz(A)}{P} \rceil \cdot P$, and we certainly have $nnz(A) \leq \lceil \frac{nnz(A)}{P} \rceil \cdot P$. By combining Equations (10) and (8), we obtain Equation (9), hence the constructed partitioning is feasible. □

Before expanding on the Mondriaan package, we first briefly discuss some of the other matrix partitioners. While all matrix partitioners have the same goal of partitioning the matrix, the specific details may vary. For instance, one may choose to minimize different objectives, either maximum or total number of sent messages, maximum or total communication volume, or some other measure. One may choose to use a graph or a hypergraph representation of the matrix, and one may try to either split this graph directly in the desired number of parts, or recursively by repeatedly bipartitioning sets of nonzeros. In the next section, we will explore some of the available algorithms and software packages along with their properties.

1.2 Related work

We will now discuss some of the work that has already been done regarding matrix partitioning, and their related software packages. In the early ages, matrix partitioning was done using graph representations, such as in METIS [18] and ParMETIS [19]. However, it was experienced that such graph models could not reflect the desired quantities (e.g. total communication volume) correctly [14]. Hence attention started to shift towards using hypergraph models instead of graph models to model the nonzero structure of matrices, as with hypergraphs it is possible to model the desired quantities exactly. Research has been done developing parallel hypergraph partitioners, like Par$k$way [25] and Zoltan [9], while PaToH [36], hMETIS [17] and Mondriaan are serial hypergraph partitioners.

At the largest scale, two different approaches exist when partitioning a hypergraph. One can either partition the hypergraph in k parts all at once, or one can bipartition the hypergraph recursively until k partitions have been formed. Amongst others, hMETIS includes a successful attempt at the former, partitioning the hypergraph in k parts at once, by first coarsening the hypergraph, then applying recursive bipartitioning to the coarse hypergraph, resulting in a coarse k-way partitioning which is then uncoarsened to the original size. Also the jagged-like and checkerboard partitionings in PaToH use a direct approach, both working on a $k = P \times Q$ processor mesh, first partitioning into columns and then partitioning the columns separately into blocks. Mondriaan, Zoltan and KaHyPar [23] all include a recursive bipartitioning algorithm. Mondriaan in particular, thanks its name to the alternating-direction strategy, which produces both a good partitioning and appealing images when visualized. In this strategy, each recursion of the recursive bipartitioning changes direction. If in one recursion the matrix is bipartitioned in the column direction, then in the next recursion it will bipartition in the row direction and vice versa.

There are many different partitioning strategies available, and hence it may be difficult to determine which method to use when given the task to partition a matrix. In 2010, to this end, Çatalyürek, Aykanat

and Uçar [7] proposed a recipe for matrix partitioning with PaToH, to choose automatically between different partitioning strategies therein included, namely row-wise and column-wise partitioning, fine-grain [33] partitioning, jagged-like [35] partitioning and a version of checkerboard [34] partitioning. In their research, they found that when minimizing the total communication volume, the fine grain method produces the best results, but also takes the longest time to complete. Their method recipe allowed them to obtain slightly better results on average, using on average less computing time by choosing more efficient methods than fine-grain when possible.

As mentioned before, the Mondriaan matrix partitioning package does not explicitly reduce the maximum communication cost per processor, but rather the total communication cost. In [24], Selvakkumaran and Karypis make an attempt to include a post-processing step in hMetis after having obtained a good solution with low number of cut hyperedges. Note that this is a different objective than Mondriaan uses, as in [24] only the number of cut hyperedges is counted, while Mondriaan also counts the number of partitions a hyperedge is distributed over. In their post-processing step, they try to minimize the maximum partition degree using some different techniques. Their results showed a significant improvement in the solution quality based on this secondary measure.

The latest development within Mondriaan was due to Pelt and Bisseling [21], who developed a medium-grain method for 2D bipartitioning. This method divides the input matrix A into two matrices A_c, A_r, such that $A = A_c + A_r$, and computes a bipartitioning of A_c and A_r simultaneously, partitioning A_c column-wise and A_r row-wise. This method improved over the then best method, localbest, which is an adaption of the alternating-direction strategy, in which in each recursion the split direction is not predetermined, but both directions are tried and the best is chosen. With $P = 2$, the medium-grain method with iterative refinement (IR) produces solutions with 27% lower communication volume using 28% less time compared to localbest without IR. The medium-grain method with IR also improved over the fine-grain method without IR, producing 22% lower volumes in 45% less time with $P = 2$. Hence within Mondriaan, the medium-grain method with IR is the current default method.

Most implementations of multilevel bipartitioning follow the same recipe, first iteratively matching similar vertices and merging them, then computing an *initial partitioning*, which is then iteratively expanded and refined until the original size is reached. In KaHyPar, an alternative is proposed, where in the coarsening phase not all (not even nearly all) vertices are matched, but just two. Additionally, in the uncoarsening phase less vertices need to be considered for refinement as most vertices stay unchanged between iterations. While this approach requires more iterations to be done, the authors found that their approach is faster than other, traditional multilevel bipartitionings.

1.3 Current state of the Mondriaan partitioning package

The Mondriaan package provides a heuristic to calculate a feasible solution to problem 1.4. In doing so, it tries to achieve a low communication volume, but no guarantees are given about the quality of the solution. To do this, Mondriaan makes extensive use of hypergraphs. A hypergraph $H = (V, N)$ is a generalization of a graph, where edges are replaced by hyperedges (or nets). While in a graph an edge connects two vertices, a hyperedge may connect any number of vertices. In other words, a hyperedge can be any subset of V.

Mondriaan partitions a matrix by iteratively bipartitioning nonzeros of the matrix until all parts have a sufficiently low weight, i.e. until (7) is satisfied. In each iteration, the algorithm searches for a part with too much weight, and upon finding such a part it initiates a bipartitioning on this part. This bipartitioning consists of a few steps.

The first step is to translate the matrix into a hypergraph. This can be achieved in multiple ways, and one can choose from:

- A row-net model, in which each row corresponds to a net and each column corresponds to a vertex. A net contains a vertex if and only if the corresponding row has a nonzero in the corresponding column.
- A column-net model, which is the same as above, with columns and rows interchanged.

- A fine-grain model, in which each nonzero corresponds to a vertex and the rows and columns correspond to nets.
- A medium-grain model, where the matrix A is divided into two parts A_r, A_c such that $A = A_r + A_c$, and the row-net model is applied to the new matrix

$$B = \begin{pmatrix} I_n & A_r^T \\ A_c & I_m \end{pmatrix}. \tag{11}$$

Combinations of the above are also possible; Mondriaan provides the following combinations:

- Alternate: Force splits to alternate between row-net and column-net models in each recursion. Colourings of partitionings produced by this strategy are the origin of the name 'Mondriaan'.
- Localbest: Like alternate, but in each recursion try both the row-net and column-net method and use the best one.
- Localratio: Like alternate, but in each recursion decide which model to choose based on the ratio between submatrix height and width.
- Hybrid: In each recursion, try row-net, column-net and fine-grain, and use the best.

The second step is the coarsening. Here, a matching of the vertices is computed, where the aim is to match vertices that have many nets in common. In doing this, we halve the number of vertices, while the overall structure of the hypergraph is largely maintained. After coarsening the hypergraph sufficiently many times, we continue to the third step. The third step is the actual bipartitioning, using the Kernighan-Lin / Fiduccia-Mattheyses (KLFM) algorithm. This KLFM algorithm returns a bipartitioning, of which one partition we assign to one part (processor), and the other partition we assign to the other. The fourth step takes the computed bipartitioning back to the original problem size, also called the uncoarsening. For each pair of vertices $\{i, j\}$ that were matched in the coarsening into one vertex k, we now assign both i and j to the same part as k is assigned to. After each uncoarsening stage, a single run of the KLFM algorithm is done to find any cheap improvements on the partitioning. Once we carried out all uncoarsenings, the fifth step is to translate the obtained bipartitioned hypergraph back into a matrix. Effectively, we obtain the same matrix we started with, only now with nonzeros assigned to either part 0 or part 1. This concludes the bipartitioning. Optionally, just before the fifth step, we may carry out iterative refinement, an idea taken from the medium grain method.

In some of the steps we just discussed, we can choose between several different approaches. We take a look at some options in the coarsening step. Here, the matching of vertices is subdivided into two main tasks. The first task is, given a vertex, to search for another vertex which has many nets in common with the given vertex (also called a neighbour). When looking at the row-net model, this amounts to searching for pairs of columns in the matrix with high inner products. The second task is to obtain a good matching, using this neighbour-finding function. Fagginger Auer and Bisseling [2] describe a few different methods for both neighbour-finding and for the matching. The neighbour-finding functions discussed there are:

1. the Mondriaan method, which calculates the inner products between columns exactly,
2. the MondriaanTB method, which uses a smart tie-breaking rule,
3. the Stairway method, which computes lower bounds on the inner products, performing faster than the Mondriaan method,
4. the StairwayM method, which combines the Stairway and Mondriaan method to obtain better results than Stairway with lower running time than Mondriaan.

Furthermore, the matching functions provided are (see [2] and references cited therein):

1. the Greedy method, which matches a column j with another column k that has highest inner product with j,
2. the PGA' method, a $\frac{1}{2}$-approximation algorithm,
3. the GPA method, another $\frac{1}{2}$-approximation algorithm,
4. the ROMA method, based on a $(\frac{2}{3} - \epsilon)$-approximation algorithm.

The ROMA method produced the best results but took too long computing them, and GPA produced similar results as PGA', while also being slower than PGA'. Hence, in Mondriaan the Greedy and PGA' methods are currently available, and the GPA and ROMA methods are not included. Regarding the neighbour-finding functions, the MondriaanTB method produced better results than the Mondriaan method, and StairwayM produced better results than Stairway, hence MondriaanTB is available as an option named 'inproduct', and StairwayM is available as 'stairway'.

After choosing a neighbour-finding function and matching algorithm, another option that may be given is whether the computed inner products should be scaled or not. This is only of effect if the inproduct neighbour-finding is chosen. Most of the options are discussed in [28]. Denoting the number of nonzeros in columns c_i and c_j by ν_i and ν_j respectively, one can choose between no scaling, scaling by $(\min(\nu_i, \nu_j))^{-1}$ (min), $(\max(\nu_i, \nu_j))^{-1}$ (max), $(\sqrt{\nu_i \nu_j})^{-1}$ (cos) or $(\nu_i + \nu_j - < c_i, c_j >)^{-1}$ (Jaccard). Of these methods, the min() option was observed to work best for the matrices tested, hence this is the default in Mondriaan. Another related option is InprodMatchingOrder, which defaults to decreasing-weight, next to increasing-weight, decreasing-degree, increasing-degree, natural (given order) and random. Here, weight refers to the number of nonzeros that have been merged into a column, and degree refers to just the amount of nonzeros that are in a column. Lastly there is the NetScaling option, which defaults to linear, but can also be set to 'no'. This option allows to scale all inner products with the net size of the net they are in.

These are not all options available in the package, but it does give some background on what settings are available.

2 Preliminary notes

2.1 Notation

Before moving on to the different considered modifications, we first define some of the notation and terminology used in this thesis. We denote a nonzero of a matrix as an ordered pair of indices (i, j), where i and j are the row index and column index respectively. The nonzeros of a matrix A are a set of such pairs $nz(A) = \{(i_0, j_0), \ldots, (i_{nnz-1}, j_{nnz-1})\}$, where $nnz = nnz(A) = |nz(A)|$ is the number of nonzeros of A. When partitioning a matrix, we denote the number of processors to partition over with P and the maximum imbalance ratio is given by ϵ as defined in 1.4. The achieved imbalance after partitioning is denoted by ϵ'.

When recursively bipartitioning a matrix, the entire matrix is first bipartitioned, after which each of the created partitions is recursively bipartitioned. These recursive bipartitionings can be interpreted as a binary tree, each node representing a bipartitioning and the root being the bipartitioning of the original matrix. The depth d of a bipartitioning is then defined as the depth of the bipartitioning within this tree, taking $d = 0$ for the root of the tree. For example, for $P = 4$ one bipartitioning is performed at depth 0 and two bipartitionings are performed at depth 1. Note that given a number of processors P being a power of 2, at depth d the number of bipartitionings performed equals 2^d.

During the recursive bipartitioning of a matrix, a partition may either belong to a single processor or to multiple processors. In the latter case, the partition will be further bipartitioned in later recursive bipartitionings. Note the use of terminology here: we compute a *partitioning* of a matrix into P partitions, to in the end assign each partition to a *processor*. During the algorithm, a partition may contain as many nonzeros as its number of processors allows.

Throughout this thesis at various places means and standard deviations are presented. The means are often called performance ratios, which are given by the expression

$$\frac{1}{n_m} \sum_{i=1}^{n_m} \frac{x_i^{new}}{x_i^{old}}.$$

Here, n_m is the number of matrices and x_i^{old} and x_i^{new} are the averages of a certain quantity computed on matrix i in an old and a new situation, respectively, where the average is taken over a number of runs (often 5 or 10 runs). The quantity x often represents either run time, communication volume or load imbalance. In most cases, the standard deviations should be interpreted as spreads rather than uncertainties: the means are computed from data points that should not be expected to be clustered around one true value. Also, any log() functions used in this thesis implicitly have base 2, unless specified otherwise.

2.2 Test machines

We performs calculations with Mondriaan on a few different machines:

1. Intel(R) Quad Core Q8200 @ 2.33GHz with 2048 KiB cache, 3 GiB RAM, Ubuntu Linux 16.04
2. Intel(R) Dual Core E2160 @ 1.80GHz with 1024 KiB cache, 4 GiB RAM, Ubuntu Linux 16.04
3. Quad-Core AMD Opteron(tm) Processor 2378 @ 2.4 GHz with 512 KiB cache, 32 GiB RAM, Scientific Linux SL release 5.5
4. Pentium(R) Dual Core E5200 @ 2.50GHz with 2048 KiB cache, 2 GiB RAM, Ubuntu Linux 14.04

Machines 1, 2 and 4 are home desktop systems, set to specifically run only Mondriaan to minimize noise in timing measurements. The third machine is the science server of Utrecht University, which may be used by all students. Hence, timing data from this server is unreliable as different usage patterns may result in different amounts of cache misses between multiple runs. Results obtained from this server are thus primarily used to analyze solution quality, while machines 1, 2 and 4 are used for both solution quality and timing measurements.

2.3 Test matrices

To test our proposed modifications, we perform multiple calculations of Mondriaan using a wide variety of different matrices. All matrices are taken from the SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection) [8]. The matrices used are the matrices obeying the following rules:

- We take all matrices with $10^3 \leq nnz \leq 10^7$ and an even matrix ID, the maximum ID being 2540.
- Of these, we remove any structurally duplicate matrices. As Mondriaan only uses the structure of matrices and not their values, any two matrices that are structurally the same need only to be considered once.
- We perform a single run of calculations with Mondriaan using default options, to check how long calculations take to perform. Any matrices taking longer than one hour on machine 1 are removed from the list.

This leads to a list of 955 matrices, upon which we will test our modifications to Mondriaan. We may sometimes deviate a little from this list, in these cases we will describe which matrices are added or left out. Sometimes with higher P, Mondriaan does not always succeed in finding a feasible solution. For example, for $P = 16$, Mondriaan often does not find a solution for the matrix (`HB/ash608`). In these cases, we will without further notice leave out the concerned matrices from the results, as there are enough other matrices to base results upon. Only with a high number of failures we will make a note in the results. All calculations following are performed with the default options of Mondriaan 4.1, unless specified otherwise.

3 Three-way quicksort

In Mondriaan, often different kinds of sets need to be sorted. For instance, a set of weights may be sorted because we want to process items in decreasing order of weight. Mondriaan includes its own implementation of quicksort, the prime reason for this being independence from library function implementations. The question treated here is whether we can, and should, improve the performance of this sorting function.

3.1 Mondriaan's sorting function

The sorting function included in Mondriaan, from now on called QSort() and presented in Algorithm 1, has the following characteristics:

1. it is recursive: QSort() calls itself recursively,
2. it uses random pivots,
3. it is unstable (i.e., it does not preserve the order of identical elements).

Algorithm 1 QSort()

```
1:  Sort list using quicksort. Input of initial call should be the set to be sorted along with the identity permutation.
2:  Input: List of n numbers list, current permutation with respect to start index.
3:  Output: L is sorted, with index containing the applied permutation
4:  procedure QSort
5:      s ← median({list[random(0, n − 1)], list[random(0, n − 1)], list[random(0, n − 1)]})
6:      i ← 0, j ← n − 1
7:      while i ≤ j do
8:          while list[i] > s and i < n − 1 do
9:              i ← i + 1
10:         while list[j] < s and i > 0 do
11:             j ← j − 1
12:         if i < j then
13:             swap(list[i], list[j])
14:             swap(index[i], index[j])
15:             i ← i + 1
16:             j ← j − 1
17:         else if i = j then
18:             if list[i] ≥ s then
19:                 i ← i + 1
20:             else
21:                 j ← j − 1
22:     QSort(list[0, j], index[0, j])
23:     QSort(list[i, n − 1], index[i, n − 1])
```

We are not interested in creating a stable sorting algorithm, but in improving the run time of QSort(). The first inefficiency to be noticed is the recursive character of QSort(): recursion is relatively slow in C, compared to methods using and maintaining their own call stack. Moreover, one can argue that a recursive algorithm is limited by the stack size, as there is a fixed (but configurable) number of maximum recursive function calls, imposed by the limited stack size of a process. However, as the expected recursion depth of quicksort is $O(\log n)$, a common stack size of a few megabytes would impose a limit on the list size of $n \leq O(2^{1000000})$, which is a limit we will never reach. Notice that at depth k in the recursion tree, we expect

to call QSort() 2^k times. With a maximum depth of $m = O(\log n)$, we obtain a total number of QSort() calls equal to

$$\sum_{k=0}^{m} 2^k = 2^{m+1} - 1 = O(n). \tag{12}$$

Hence, by removing the recursion from the QSort() algorithm, we may remove $O(n)$ function calls, thereby potentially improving performance.

A second inefficiency in the algorithm is the use of $random(a, b)$, a function that returns a (pseudo-)random number from the set $[a, b]$. In each recursion, the pivot is determined as the median of three random list entries. As in each recursion of QSort() we determine one pivot, we have $O(n)$ total calls to $random()$ by our argumentation above. To circumvent these calls, one other approach is choosing a pivot as the median of either three or nine list entries at fixed positions, as explained further below and based upon the approach taken in [3]. One important note is that the expected running time of $O(n \log n)$ relies on the assumption that the chosen pivots are random samples from the lists to be sorted[15]. This assumption is justified whenever we use pivots at random positions or whenever the input data is randomly permuted. In [3], it was indeed experimentally found that the median-3 and median-9 rules admitted $O(n \log n)$ run time on random data, with good complexity constants. We will be experimenting with both random and deterministic pivots, and determine which has our preference.

A third point of attention is the concept of 3-way quicksort. Instead of only partitioning elements into a 'lower' and 'higher' list, we also maintain a list with elements equal to the pivot, hence creating 3 partitions. In cases where the lists to be sorted contain many equal elements, this may provide a significant speed-up. As it is expected that Mondriaan typically sorts these kinds of lists, we will look into the effects of using 3-way quicksort in Mondriaan.

In [15], it is opted to stop sorting whenever an input array has size smaller than or equal to some threshold T. After quick sort finishes, the array is nearly sorted, and each element is at most $T-1$ positions away from its correct position. Then insertion sort is used to put all elements into the right position. While insertion sort has a running time of $O(n^2)$ in the input size, for low n this may provide a slight performance gain. Indeed, in the C library glibc [1] this approach is taken with $T = 5$. We will investigate whether this feature will improve sorting performance in Mondriaan.

3.2 New quicksort implementation

3.2.1 Pivot determination

We consider two different algorithms to determine a pivot s from a list L of n elements. The first is a random procedure used in the original Mondriaan sorting algorithm: we choose three random positions $i, j, k = random(0, n-1)$, and choose the median from these three elements $s = median(\{L_i, L_j, L_k\})$ as pivot. The second method is a deterministic method used in [3], that distinguishes two regimes. If $n \leq \tau$ for some $\tau \in \mathbb{N}$, we choose the pivot to be the median of three values $s = median(\{L_0, L_{n/2}, L_{n-1}\})$. Otherwise if $n > \tau$, we choose the pivot as a median of medians $s = median(\{x, y, z\})$, where

$$\begin{aligned} x &= median(\{L_0, \quad L_\Delta, \quad L_{2\Delta} \quad\}), \\ y &= median(\{L_{n/2-\Delta}, \; L_{n/2}, \quad L_{n/2+\Delta}\}), \\ z &= median(\{L_{n-1-2\Delta}, L_{n-1-\Delta}, L_{n-1} \quad\}), \end{aligned}$$

where $\Delta = n/8$, of course all in integer arithmetic. Following [3], we choose $\tau = 40$.

3.2.2 Non-recursive two-way quicksort

To start off with building an algorithm, we implement a non-recursive variant of quicksort, based on the implementation of quicksort in the glibc library [1] and the paper [3].

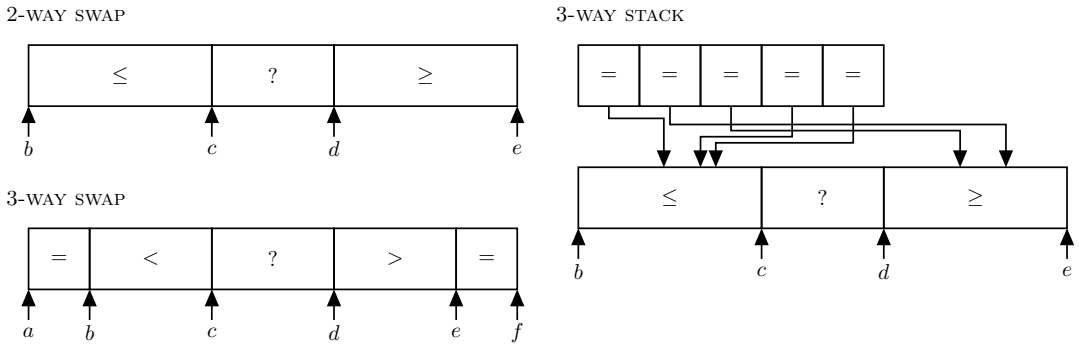


Figure 1: Illustration of pointer variables in the main loop of the different quicksort methods. For the 3-way stack method, the figure illustrates a case where five elements equal to the pivot have already been found.

From the glibc library implementation of quicksort, we borrow the stack construction. The stack contains pairs of indices (lo, hi) which represent ranges within the list that still have to be sorted, eliminating the need for recursion. By always pushing the larger of two ranges onto the stack and dealing with the smaller range right away in the next iteration, we guarantee that we never need a stack size larger than $\lceil \log(n) \rceil$ [15][1]. Apart from some bookkeeping variables, this is the only large memory space required in this implementation, hence we may conclude that the space complexity is $O(\log n)$.

The algorithm can be found in Algorithm 2.

3.2.3 Non-recursive three-way quicksort

Three-way quicksort is a variation on two-way quicksort, where apart from two partitions containing elements respectively lower and higher than the pivot, we also keep track of a third partition containing elements equal to the pivot. For sets with many equal elements, this promises performance improvement over two-way quicksort, as the algorithm can deal with the chunk of equal elements immediately and does not need to recurse into them. Starting from Algorithm 2, only a few adaptions need to be made, as shown in Algorithm 3. At three places, we check for elements equal to the pivot element s. We choose two different approaches here. In the first approach, we create an extra stack of indices to equal elements, to which we add an index of an element whenever it equals the pivot (*HandleEqual()*). In the end, we use this stack of indices to swap equal elements to the correct position (*ProcessEquals()*). In the second approach, we swap the equal elements to their own partition just as we do with the elements lower and higher than the pivot (*HandleEqual()*), swapping them back to their correct positions afterwards (*ProcessEquals()*). An illustration of the methods is shown in Figure 1, and the modified algorithm can be found in Algorithm 3.

An advantage of the first method is that it reduces the amount of swaps required, while it suffers from the disadvantage that it requires more memory, which is in the order of the maximum number of equal elements in the set to be sorted. While the second method may require more swaps, its space complexity remains the same as in the two-way algorithm.

Algorithm 2&3: Non-recursive quicksort algorithms. QSort2w() defines the 2-way quicksort algorithm, QSort3w() defines the modifications done to QSort2w() to establish a 3-way quicksort algorithm. Note that the 2-way algorithm is highly based on [1].

Algorithm 2: QSort2w()

```
1: Sort list using quicksort. Input of initial call should be the set
   to be sorted along with the identity permutation.
2: Input: List of n numbers list, current permutation with respect
   to start index.
3: Output: L is sorted, with index containing the applied per-
   mutation
4: procedure QSort2w
5:     b ← 0, e ← n − 1, stack ← {}
6:     push(stack, (NULL, NULL))
7:     while size(stack) > 0 do
8:         c ← b, d ← e
9:         if d − c > 1 then
10:            s ← determinePivot(list[c..d])
11:            while c ≤ d do
12:                while list[c] < s do
13:                    c ← c + 1
14:                while s < list[d] do
15:                    d ← d − 1
16:                if c < d then
17:                    swap(list[c], list[d]), swap(index[c], index[d])
18:                    c ← c + 1, d ← d − 1
19:                else if c = d then
20:                    c ← c + 1, d ← d − 1
21:                    break
22:        else
23:            % Only two elements
24:            if list[d] < list[c] then
25:                swap(list[c], list[d]), swap(index[c], index[d])
26:            c ← c + 1, d ← d − 1
27:        % Determine which part to partition next
28:        if d − b + 1 ≤ T ∧ e − c + 1 ≤ T then
29:            (b, e) ← pop(stack)
30:        else if d − b + 1 ≤ T then
31:            b ← c
32:        else if e − c + 1 ≤ T then
33:            e ← d
34:        else if d − b > e − c then
35:            push(stack, (b, d))
36:            b ← c
37:        else
38:            push(stack, (c, e))
39:            e ← d
```

Algorithm 3: QSort3w()

```
1: (...)
2: if d − c > 1 then
3:     s ← determinePivot(list[c..d])
4:     % Check equal elements
5:     while c ≤ d ∧ list[c] = s do
6:         HandleEqual(c)
7:         c ← c + 1
8:     while c ≤ d ∧ list[d] = s do
9:         HandleEqual(d)
10:        d ← d − 1
11:
12:    while c ≤ d do
13:        % Check equal elements
14:        while c ≤ d ∧ list[c] = s do
15:            HandleEqual(c)
16:            c ← c + 1
17:        while c ≤ d ∧ list[d] = s do
18:            HandleEqual(d)
19:            d ← d − 1
20:
21:        % Partition
22:        while c ≤ d ∧ list[c] < s do
23:            c ← c + 1
24:        while c ≤ d ∧ s < list[d] do
25:            d ← d − 1
26:
27:        if c < d then
28:            % Swap
29:            swap(list[c], list[d])
30:            swap(index[c], index[d])
31:            if list[c] = s then
32:                HandleEqual(c)
33:            if list[d] = s then
34:                HandleEqual(d)
35:            c ← c + 1, d ← d − 1
36:        else if c = d then
37:            c ← c + 1, d ← d − 1
38:            break
39:    ProcessEquals()
40: (...)
```

3.3 Results

We performs tests on machine 1, using Mondriaan with our default test set of 955 matrices to assess how Mondriaan performs with the mentioned variations on quicksort. In addition, we construct synthetic test sets, based on the sets used in [3, figure 1], see Algorithm 4.

The five different sorting algorithms we present here are the current recursive Mondriaan sorting algorithm, the non-recursive 2-way algorithm and the 3-way algorithm with stacks and using swaps, all three with deterministic pivots, and lastly 3-way sorting using swaps and random pivots.

To obtain timing results for test sets within Mondriaan, we interrupt each sorting call to evaluate each algorithm on the given set of data. For each matrix, all contributions are summed to provide a grand total of sorting time in the end. This is done 5 times for all matrices after which the results are averaged per

Algorithm 4 The synthetic test sets used. Note that most of these sets are copied from [3], where these are claimed to be some of the hardest cases for sorting available.

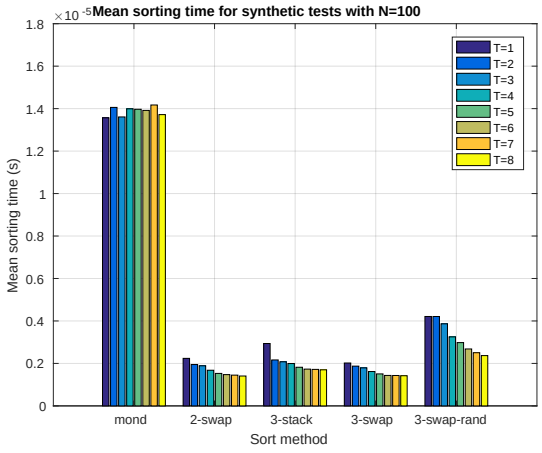
```
1:  for n ∈ {100, 1000, 10000, 100000} do
2:      for (m = 1; m < 2*n; m *= 2) do
3:          for dist in { increasing, sawtooth, rand, stagger, plateau, shuffle } do
4:              for (i = j = 0, k = 1; i < n; i++) do
5:                  switch (dist)
6:                  case increasing: x[i] = x[i-1]+random(1,m)          ▷ x[-1] = 0
7:                  case sawtooth: x[i] = i % m
8:                  case rand: x[i] = rand(0, m-1)
9:                  case stagger: x[i] = (i*m + i) % n
10:                 case plateau: x[i] = min(i, m)
11:                 case shuffle: x[i] = rand()%m? (j+=2): (k+=2)
12:             test copy(x)                                            ▷ test x
13:             test reverse(x, 0, n)                                   ▷ test x reversed
14:             test reverse(x, 0, n/2)                                 ▷ test x with front half reversed
15:             test reverse(x, n/2, n)                                 ▷ test x with back half reversed
16:             test sort(x)                                            ▷ test an ordered copy of x
17:             test dither(x)                                          ▷ test x with noise: add i%5 to x[i]
```

matrix. For the synthetic tests, the timing results are averaged over 20 runs of Algorithm 4.

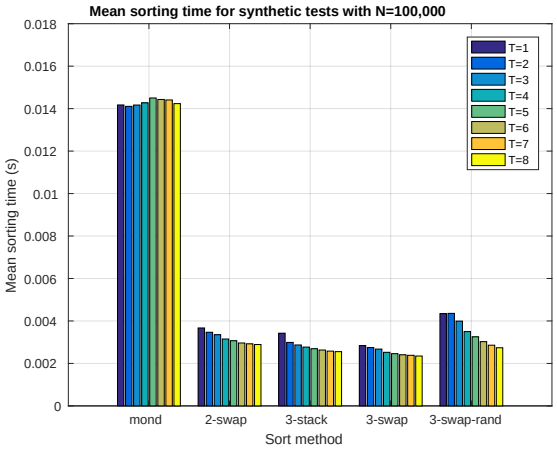
It is immediately clear from Figure 2 that the new sorting functions all perform better than the current sorting function, with speed-ups exceeding 75%. We see that in Mondriaan, 3-way sorting works better than 2-way sorting, while this difference is less prominent in the synthetic tests. An explanation for this is that the synthetic tests include increasing sets and other sets with no duplicate entries, for which 3-way sorting performs worse due to the extra overhead. Indeed, looking at the synthetic tests for random sets (not shown here), we see behaviour comparable with the results from the Mondriaan tests, in particular for lower values for m. Note that lower m signifies the presence of many equal elements, hence we can also conclude that Mondriaan sorts sets that show similar sorting behaviour as sets containing many equal elements.

Another observation is that the 3-way swap implementation outperforms the 3-way stack implementation. We do not have a solid explanation for this, apart from the suggestion that the swap implementation may be more cache-friendly. Indeed, Figure 1 illustrates that the stack implementation will require a lot of jumping through memory when moving the elements equal to the pivot to the correct positions.

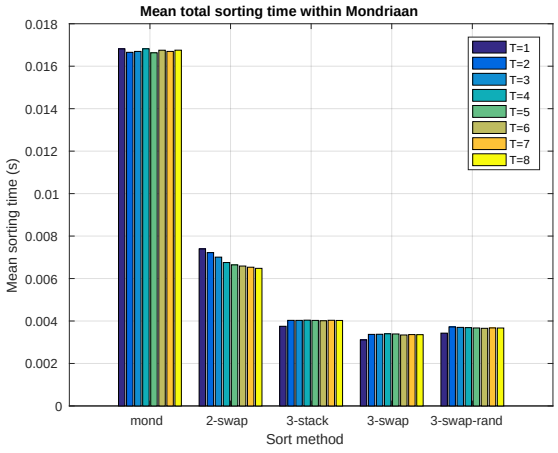
A third observation from Figure 2 is that in the Mondriaan tests, the 3-way methods do not seem to profit from values for T higher than one. To the contrary, the cases $T = 2$ to $T = 8$ have equal performance, while $T = 1$ has better performance compared to the others. This may be explained as follows. Suppose some set in an iteration of 3-way qsort consists of three different elements a, b and c, and we happen to select the middle one (say, b) as pivot (other cases follow a similar argumentation). Then 3-way sorting will generate one partition for elements equal to b, and two for elements respectively equal to a and c. Consider the partition A with elements equal to a, note that these are already sorted as they are all equal. This partition has either size $|A| > T$, in which case the 3-way sort will in the next iteration mark all elements as sorted, or $|A| \leq T$ in which case the algorithm does nothing. In both cases, insertion sort is not needed any more to sort the elements: the partition under consideration is already sorted before we arrive at size below T. Note that if the sets A as defined here often have a large size $|A| \geq T_{\max} = 8$, this behaviour is independent of T, in which cases choosing $T > 1$ will result in performing an insertion sort that does not do much sorting. This is exactly what we see in Figure 2c: when $T = 1$, we do not use insertion sort, and when $T > 1$ we do use insertion sort and this imposes a time penalty independent of T. These observations are supported by the results of 2-way sorting: as this algorithm does not have any way to quit early (i.e., at



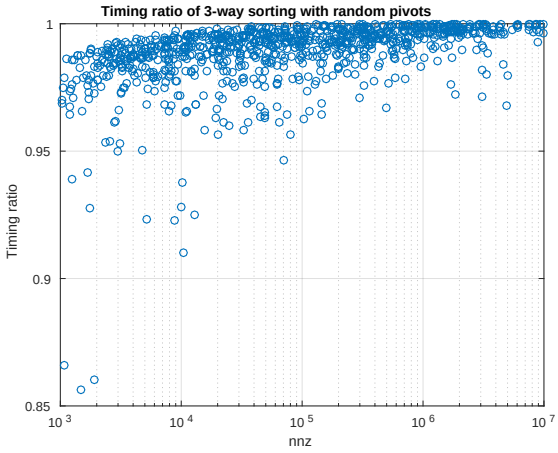
(a) Average of all synthetic tests as defined in Algorithm 4 for $n = 100$.



(b) Same as (a), but for $n = 100{,}000$.



(c) Total sorting time in Mondriaan averaged over 5 runs.



(d) Mean timing ratios.

Figure 2: ***(a,b,c)*** *Mean sorting times for all five sorting methods. For all new methods, each bar represents a different threshold value, from $T = 1$ (blue/left) to $T = 8$ (yellow/right). For the current* `mond` *method, T has no effect and hence each bar represents the same quantity, any fluctuations being caused by noise and different sequences of RNG.* ***(d)*** *Mean timing ratios, comparing total Mondriaan run times when using 3-way quicksort with random pivots versus the current sorting method. Each dot represents a matrix, and its value equals the mean run time with 3-way quicksort divided by the run time with the old quicksort, both averaged over 5 runs. In particular for small matrices some performance is gained, but for larger matrices the relative performance gain is small. The mean of all ratios is 0.9887 ± 0.0138, indicating a mean performance gain of just over 1%.*

sizes greater than T), it does profit from larger values of T.

Note that additionally, the fact that in the graphs the running time still tends to be decreasing after $T = 8$ does not mean that $T > 8$ is a good threshold value, as these results may be biased due to sets that are easily sorted. More intensive research should be done to determine such an optimal value if such a value is desired.

Our last observation is that for larger synthetic sets, the three deterministic algorithms have a running time closer to the running time of the random 3-way algorithm than in the case for smaller sets. While we are not sure this is the reason, this is probably due to the fact that the total costs of using the random number generator (RNG) are linear in the input size, as we have a fixed number of random() calls per iteration, and we perform $O(n)$ iterations. Compared to the $O(n \log n)$ cost of the total algorithm, the costs of calling the RNG will become relatively small with increasing n.

While the 3-way random algorithm chooses three random elements, we also experimented with choosing three *distinct* random elements. As no duplicate positions are chosen in this case, this may theoretically improve the quality of the pivots. However, timing results indicated that any possible pivot quality improvement was undone by the additional logic that was required, hence the method with distinct random elements was inferior to the reported 3-way algorithm.

3.4 Conclusion

When considering the synthetic tests, we do not see a significant advantage of 3-way sort over 2-way sort. This is not surprising however, because 3-way sorting can only be expected to perform better whenever there are many duplicate entries. In Mondriaan, these are present and hence 3-way sorting seems to pay off in comparison with 2-way sorting, with a performance gain of at least 30%.

Compared to the current sorting algorithm, the non-recursive 2-way algorithm performs already much better, and the 3-way sorting variations improve the old algorithm with speed-ups exceeding 75%. This is a significant speed-up. However, it must be noted that sorting does not take up a large part of the running time of the complete Mondriaan algorithm. Hence, the speed-ups in the sorting algorithm are only slightly visible in the total running times.

As we will be working on other sections of Mondriaan, improving their running times and hence decreasing total running times of Mondriaan, the relative time spent at sorting will only increase, in accordance with the principle *When improving code execution speed, you actually shift bottlenecks*. Hence, despite the fact that our new algorithms show only a marginal improvement, including one of the new algorithms is still advised. Due to the nature of the sets to be sorted in Mondriaan, 3-way sorting proves to work well, and hence this is the algorithm of choice. Looking only at running times, the deterministic 3-way swap method proves to work best among the three variations considered, hence the advice would be to include this algorithm in the new version of Mondriaan, of course with $T = 1$.

However, we must also take into account that Mondriaan is a heuristic, and it may profit from more randomness. Indeed, with more randomness introduced, multiple runs of Mondriaan are more likely to explore a larger search space, potentially increasing the chance of obtaining good results. As the current sorting algorithm does include random pivots, it would in this regard be a regression to throw away this randomness. As Figure 2 shows that including random pivots does not introduce a large penalty, the best advice would be to include the random 3-way swap sorting algorithm in the new version of Mondriaan, again with $T = 1$ of course.

4 Gain bucket data structure

In the multilevel approach used in Mondriaan, the coarse hypergraph is bipartitioned using a simple greedy method, which is then iteratively improved using the Kernighan-Lin / Fiduccia-Mattheyses (KLFM) [12][20] algorithm. In this section, we will discuss an improvement in the implementation of this KLFM algorithm that is used in Mondriaan, that will significantly improve running times. The improvement is based on the gain bucket data structure used in [30].

4.1 The algorithm

The initial partitioning is created using a greedy method, in which the vertices of the hypergraph are assigned one by one, in decreasing order of vertex weight, to the partition which has the least weight. Here, the weight of a vertex is defined to be the number of nonzeros that it consists of, or that it has been merged from.

After the greedy method, the KLFM algorithm is used to improve the generated bipartitioning. This algorithm visits each vertex in decreasing order of their 'gains', and moves the vertex to the other partition if this is allowed by the imbalance constraint. The gain of a vertex is defined to be the reduction in total communication volume if we would move this vertex. After moving a vertex, it is marked to not be eligible for moving any more in the current iteration. After all vertices are moved, we choose the solution that had the least amount of communication that we came across. After the initial partitioning, this KLFM algorithm is run multiple times to iteratively improve the partitioning.

By default, multiple initial partitionings are created and improved with KLFM, and the best among these is chosen. In addition, the KLFM algorithm is used in the uncoarsening phase, to fine-tune the coarsened partitioning in each uncoarsening iteration. Hence, we may conclude that KLFM is used intensively throughout the algorithm.

One of the core parts of the KLFM algorithms is keeping track of the gains. Consider a vertex v we want to move from partition P_0 to P_1. If it is in a net that solely contains vertices on partition P_0, the net will be cut by moving v to P_1, hence the total communication will increase by 1. Otherwise, if vertex v is in a net that contains one vertex (just v) in P_0 and all others in P_1, the net will be uncut by moving v and hence the total communication will decrease by 1. Similar results apply with P_0 and P_1 interchanged, while in all other situations the gain will be 0. The total gain of vertex v is defined to be the sum of all gains over all nets. Note that the total gain may either by positive, in which case it will be advantageous to move v, or negative, in which case it will not be. From now on, we will refer to total gains as just gains.

4.2 The improvement

The gains of all vertices are tracked in a data structure called the gain bucket. Once a vertex is moved and locked by the KLFM algorithm, it is removed from the gain bucket. The gain bucket is a list of buckets containing vertices, each bucket representing the gain value of the vertices it contains, see Figure 3.

The gain bucket structure can be manipulated with the following methods:

- `BucketInsert`: Insert a vertex v_i with gain value g_i into the gain bucket. This amounts to a linear search for the right bucket to put v_i in, or if this bucket does not exist, the right position to create this bucket (Alg. 5, line 2). After this a new bucket entry for v_i is added to the bucket in constant time (Alg. 5, line 6).

- `BucketMove`: Change the value of a vertex v_i, which implies we will have to move the corresponding bucket entry between buckets. This is implemented by removing the bucket entry (Alg. 5, line 14), and calling `BucketInsert` to insert it again at the right position (Alg. 5, line 20).

- `BucketGetMax`: Obtain a vertex with maximum gain and optionally remove it from the gain bucket structure. This can be done in constant time because of the use of linked lists.

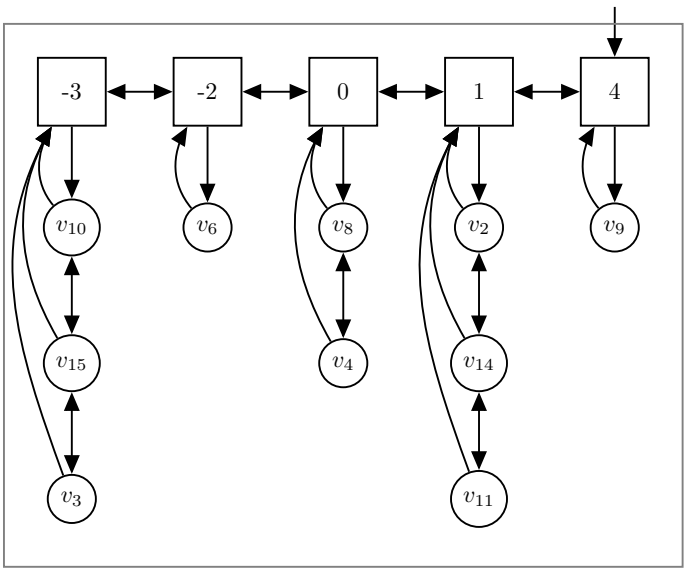


Figure 3: Gain bucket structure as implemented with a doubly linked list. Both the buckets (squares) and the bucket entries (circles) are connected as doubly linked lists, with pointers in both directions. A pointer is always maintained to the bucket containing the maximum gain value, here this is the value 4.

Whenever a vertex is moved between partitions, it may cause the gains of other vertices to change, because they may be left as single vertex in a net or a net the vertex is in may not be cut any more. Hence in each KLFM iteration, whenever a vertex is moved, all neighbouring vertices are visited and if required their gain values are updated, in which case `BucketMove` needs to be called. In contrast, `BucketInsert` is only called when the gain bucket structure is initiated, besides of course the call in `BucketMove`. Lastly, `BucketGetMax` is only called a constant number of times per KLFM iteration, one call of which deletes the vertex from the gain bucket.

It is clear that because of the method complexities and amount of function calls, any speed ups must be sought in `BucketMove` and, by extent, in `BucketInsert`. We will do this by firstly removing excessive memory allocations and secondly by saving the buckets in an array instead of a linked list, see also Figure 4. These modifications lead to the following concrete changes:

- `BucketInsert`: A linear search for the bucket to insert into is not required any more, hence the linear search is eliminated. Some constant time bookkeeping is required to keep track of the maximum gain bucket.
- `BucketMove`: The call to `BucketInsert` is eliminated by not deleting but editing the bucket entry, thereby also eliminating the mentioned memory allocation per `BucketMove` call (note the (un)link method instead of add/delete (Alg. 6, lines 14, 20)). However, whenever we move the last entry in the maximum gain bucket to a gain bucket with lower value, we will have to perform a linear search for the new maximum gain bucket (Alg. 6, line 16). This is less costly than the linear search formerly required in `BucketInsert` however, as fewer elements will need to be traversed and the data is less fragmented.
- `BucketGetMax`: Whenever we delete the last entry from the maximum gain bucket, we will need to perform the same linear search for the new maximum gain bucket as now required for `BucketMove` (Alg. 6, line 32).

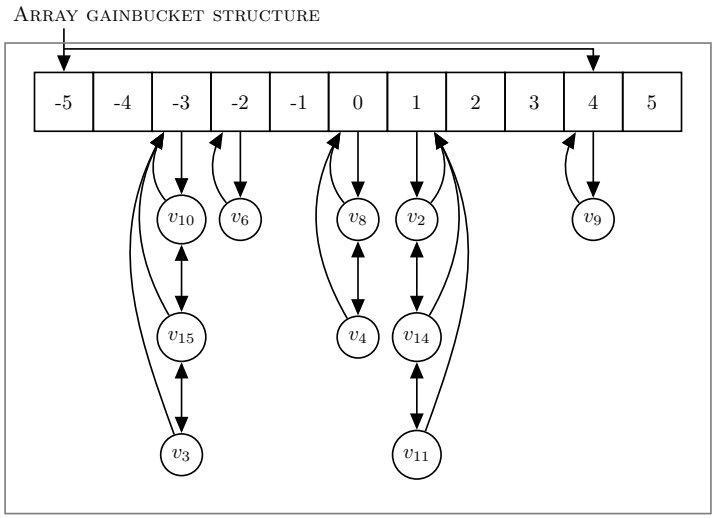


Figure 4: Gain bucket structure as implemented with an array, representing the exact same structure as Figure 3. The buckets (squares) are implemented using an array and the bucket entries (circles) are connected using doubly linked lists. A pointer is available to the first element in the list, and also the value of the maximum gain value is maintained such that the maximum element can be retrieved directly.

One disadvantage we need to consider, is that an array has a higher amount of fixed memory requirements compared to the more flexible linked lists. For a linked list, we only need to reserve memory for gain values that are actually in use by gain entries. For an array however, we need to reserve memory for all possible gain values, regardless whether we use them or not. Note that each net may contribute -1, 0 or 1 to the gain of a vertex, hence the number of gain buckets needed is $2n_{nets} + 1$, where n_{nets} is the maximum number of nets a vertex is in. Memory allocations of order $O(n_{nets})$ are common in Mondriaan, hence one additional such allocation is not considered a bad trade-off.

As we have seen, we have reduced `BucketInsert` from linear time to a constant time method. Also, though the complexity of `BucketMove` remains the same we have potentially improved its performance significantly. Lastly, the complexity of `BucketGetMax` has increased, but in the larger picture this should be outweighed by the achieved improvements.

Algorithm 5&6: Gain bucket structure algorithms: comparison of the old and new algorithm. We use v for vertex-numbers and e for bucket entries containing a vertex and a reference to the bucket it belongs to. A bucket b has the methods $addVertex(v)$ to create an entry for vertex v, $linkEntry(e)$ to add entry e, $unlinkEntry(e)$ to remove an entry, $deleteEntry(e)$ to unlink and destroy the entry and $getFirstEntry()$ to return the first entry in the list. The bucket list l is assumed to be always available. In the array implementation $l(g)$ is an array-indexed lookup of gain bucket with value g in bucket list l, and $|l(g)|$ is the number of gain entries this bucket contains. Furthermore in the array implementation, $g_{\max}$ is the maximum present gain value, which is also assumed to be always available (in the code, it is a property of l). g_b denotes the gain of bucket b.

Algorithm 5: Gain bucket structure using a linked list for the buckets

```
 1: procedure BucketInsert(Vertex v, Gain value g)
 2:     b ← arg max_{b'}{g_{b'} | g_{b'} ≤ g}
 3:     if g_b ≠ g then
 4:         b ← New bucket with value g
 5:     e ← b.addVertex(v)
 6:     return e
 7:
 8: procedure BucketMove(Gain entry e, New value g)
 9:
10:     b ← e.getBucket()
11:     v ← e.getVertex()
12:     b.deleteEntry(e)
13:     if b.isEmpty() then
14:         Delete bucket b
15:
16:
17:
18:     e ← BucketInsert(v, g)
19:
20:     return e
21:
22: procedure BucketGetMax(del)
23:     b ← l.firstBucket()
24:     e ← b.getFirstEntry()
25:     v ← e.getVertex()
26:     if del then
27:         b.deleteEntry(e)
28:         if b.isEmpty() then
29:             Delete bucket b
30:     return v
```

Algorithm 6: Gain bucket structure using an array for the buckets

```
 1: procedure BucketInsert(Vertex v, Gain value g)
 2:     b ← l(g)
 3:     g_max ← max{g, g_max}
 4:
 5:
 6:     e ← b.addVertex(v)
 7:     return e
 8:
 9: procedure BucketMove(Gain entry e, New value g)
10:                          ▷ Remove e from current bucket
11:     b ← e.getBucket()
12:     v ← e.getVertex()
13:     b.unlinkEntry(e)
14:     if b.isEmpty() ∧ g_b = g_max ∧ g < g_b then
15:         g_max ← max{g' | g ≤ g' < g_max ∧ |l(g')| > 0}
16:                                  ▷ Add e to new bucket
17:     b ← l(g)
18:     b.linkEntry(e)
19:     g_max ← max{g, g_max}
20:     return e
21:
22: procedure BucketGetMax(del)
23:     b ← l(g_max)
24:     e ← b.getFirstEntry()
25:     v ← e.getVertex()
26:     if del then
27:         b.deleteEntry(e)
28:         if b.isEmpty() then
29:             g_max ← max{g' | g' < g_max ∧ |l(g')| > 0}
30:     return v
```

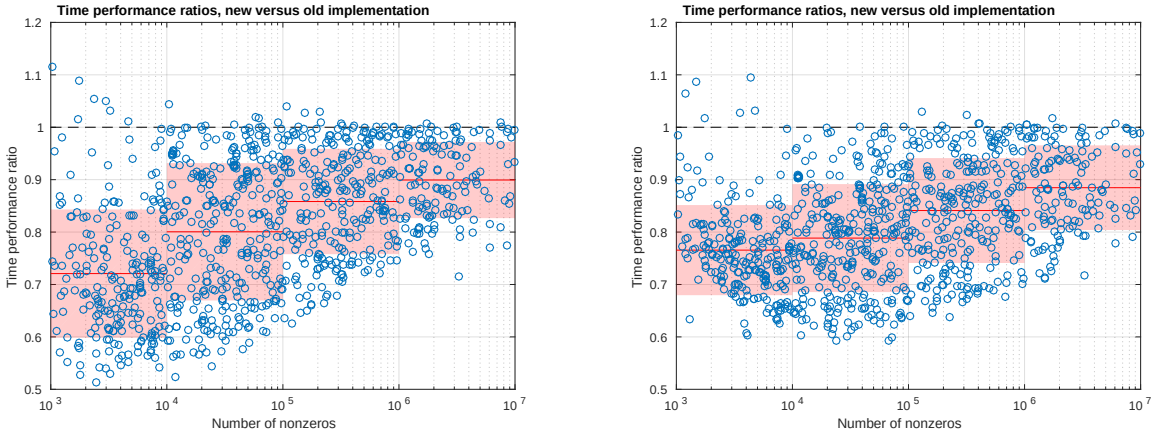
4.3 Results

We perform tests using our standard set of test matrices, the results of which can be found in Figure 5. These results show the improvements gained by using the array implementation and by preventing excessive memory allocations. One thing that is obvious from Figure 5a, is that the relative improvement decreases with the number of nonzeros. However, the absolute improvement (not shown here) does scale linearly with the number of nonzeros. This phenomenon can be explained by the fact that as the number of nonzeros increases, Mondriaan will relatively spend increasingly more time in the coarsening phase. Hence any improvements in the KLFM algorithm, which is used in the initial partitioning and uncoarsening, are of less influence. For $P = 16$ this difference becomes less pronounced, presumably because other parts of the software lead to more overhead.

In Figure 6, the results for $P = 2$ are broken down into separate results for each of the two improvements. To do this, we modified the original code to reuse the existing allocation in `BucketMove`, instead of freeing it and allocating new space. The results of the allocation improvement compare the two linked list implementations, one with the allocation improvement and the other being the original algorithm. The

$\log_{10}(nnz)$	Mean time performance ratio			
	$P = 2$	$P = 16$	Allocation impr.	Data structure impr.
$3 - 4$	0.7208 ± 0.1223	0.7656 ± 0.0860	0.8423 ± 0.0801	0.8527 ± 0.0925
$4 - 5$	0.8005 ± 0.1310	0.7885 ± 0.1030	0.8921 ± 0.0610	0.8919 ± 0.0917
$5 - 6$	0.8583 ± 0.1005	0.8411 ± 0.1001	0.9395 ± 0.0381	0.9110 ± 0.0758
$6 - 7$	0.8993 ± 0.0727	0.8847 ± 0.0808	0.9647 ± 0.0244	0.9312 ± 0.0566
$3 - 7$	0.8126 ± 0.1289	0.8123 ± 0.1038	0.9045 ± 0.0715	0.8939 ± 0.0871

Table 1: Timing ratio means belonging to Figures 5 and 6. The leftmost two columns belong to Figure 5, and indicate improvements by the combined action of the allocation improvement and the data structure improvement. The rightmost two columns belong to Figure 6, decomposing the results for $P = 2$ into separate means for each of the two improvements.



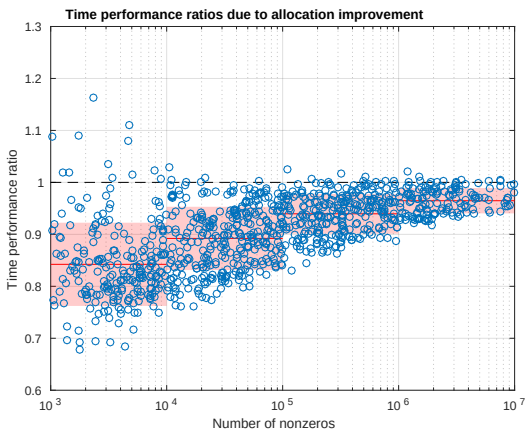
(a) $P = 2$, $\epsilon = 0.03$ *(b)* $P = 16$, $\epsilon = 0.03$

Figure 5: Timing ratios for computations using arrays versus linked lists. The results are averaged over 5 runs using the standard set of test matrices.

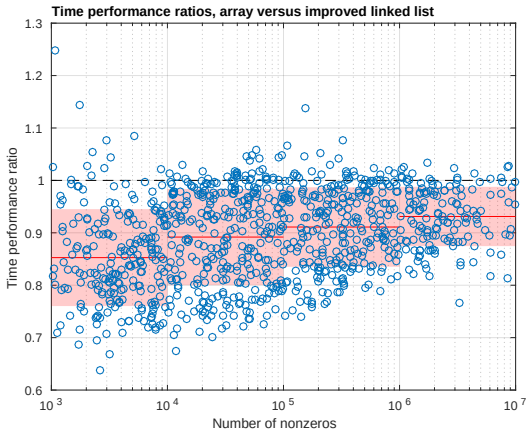
results of the data structure improvement compare the new array implementation against the linked list implementation with improved allocations. Care must be taken when comparing the numbers shown in Table 1, as the allocation improvement ratios are based on improvements from the original algorithm, and the data structure improvement ratios are based on improvements on an already improved algorithm. Improvements based on the algorithm may be calculated by taking the difference of the total improvement for $P = 2$ and the allocation improvement. Using either way of comparing the results for the individual improvements, we observe that on average both improvements perform well. For smaller matrices, slightly more than half of the improvement is due to the allocation improvement, while for larger matrices the data structure improvement takes account for most of the improvement. Presumably, this is due to the fact that for larger matrices n_{nets} will be larger and hence more buckets may exist. As we omitted a linear search through these in the new data structure, the performance gain due to the new data structure can indeed be expected to be higher for larger matrices.

4.4 Conclusion

Based on the reported results, we may conclude that for almost all tests matrices we gain speed performance by applying the new algorithm. Any matrices that take longer with the new algorithm are likely to be cases that coincidentally benefit from specific code execution combinations provided by the old algorithm, which



(a) Time performance ratios, comparing the linked list implementation with improved allocations with the original linked list implementation.



(b) Time performance ratios, comparing the array implementation with the linked list implementation with improved allocations.

Figure 6: Time performance ratios of Figure 5a broken down into improvements due to improved allocations and improvements due to the improved data structure, using the array implementation.

are cases we are not interested in. The trade-off of having fixed allocation costs of $O(n_{nets})$ is acceptable, however we do have to note that this upper bound is derived under the assumption that each net may contribute either -1, 0 or 1 to the total gain of a vertex. As this assumption is true for all current use-cases of Mondriaan, our recommendation is to include this new algorithm as default choice in Mondriaan, but retain the old algorithm using linked lists to accommodate any future features for which the assumption is not true. Lastly, it is advised to update the linked list implementation such that also there, excessive memory allocations are prohibited.

5 Zero volume search

Experience dictates that when bipartitioning a matrix, it can take quite a while for the partitioner to complete, only to decide that the matrix can be bipartitioned with zero communication volume. In that particular case, we could have come to this conclusion far earlier by first checking whether a zero volume bipartitioning (ZVBP) is possible by means of an alternative method. By applying this method in each recursion of Mondriaan, we may detect ZVBPs during the entire algorithm while they otherwise might have been left unnoticed.

The zero volume search (ZVS) algorithm we consider here consists of two separate parts. The first part searches for **connected components** within the matrix. Connected components are often defined on graphs, but as we will develop an algorithm that directly acts on a matrix, we will give a consistent definition for matrices.

Definition 5.1 (Connected component of a matrix)**.** *Two nonzeros are said to be **neighbours** if they are either in the same row or in the same column. A nonzero nz_A is said to be **reachable** from another nonzero nz_B if a sequence $(nz_1, \ldots, nz_k)$ of k nonzeros can be constructed such that $nz_1 = nz_A$, $nz_k = nz_B$, and for each $i = 1, \ldots, k-1$, we have that nz_i and nz_{i+1} are neighbours. Now, a connected component C of a matrix A is a maximal subset of nonzeros ($C \subseteq nz(A)$) with the property that every nonzero in C is reachable from any other nonzero in C. We may refer to a connected component simply as a **component**; the **weight** of a component C is defined as the number of nonzeros $|C|$ it contains.*

Note that every non-trivial matrix contains at least one connected component. In this section, we are interested in matrices containing more than one connected component. For example, suppose we have a matrix consisting of two connected components, each containing half of the nonzeros of the entire matrix. When bipartitioning this matrix into two parts we may assign each connected component to a different part. In doing so, we have constructed a partitioning of the matrix with zero communication volume, which we will prove shortly.

Unfortunately, whenever a matrix contains more than one component, chances are the matrix consists of many components, possibly each with a different weight. Hence, we have to come up with an algorithm that distributes the different components over the two partitions we want to create. Suppose for the moment $\epsilon = 0$, and consider the problem of distributing a matrix A over P processors. In a bipartitioning, we then will want to create a partition with q nonzeros and a partition with $nnz(A) - q$ nonzeros, where

$$q = \left\lfloor \frac{nnz(A)}{P} \cdot \left\lceil \frac{P}{2} \right\rceil \right\rfloor.$$

Here, the ceiling operator makes sure we generate partitions of appropriate sizes: when $P = 3$ for instance, we first want to generate a bipartitioning with partitions containing respectively one third and two thirds of the nonzeros of A. The floor operator makes sure q is defined as an integer. The most important point of attention here is that these divisions should be in integer arithmetic; the specific choice for either floor or ceiling is of less importance.

Suppose our algorithm finds c components $C_l \subseteq nz(A)$ $(l = 1, \ldots, c)$, each component C_l consisting of $w_l = |C_l|$ nonzeros. Then our goal is to find two disjoint sets of component numbers S_1, S_2 such that $S_1 \cup S_2 = \{1, \ldots, c\}$ and

$$\sum_{l \in S_1} w_l = q,$$
$$\sum_{l \in S_2} w_l = nnz(A) - q.$$

This problem is exactly the Subset Sum problem, which is a well known NP-complete problem. If we now relax the assumption we made such that we may have $\epsilon \geq 0$, we recognize the resulting problem as a relaxation of the Subset Sum problem, or equivalently, as an instance of the optimization formulation

of the Subset Sum problem with a lower bound unequal to the upper bound. Still, the problem remains NP-complete, hence whenever we have a large number of components we will have to use an approximation algorithm to solve large instances of this problem.

Lemma 5.2. *Suppose an $m \times n$ matrix A has c connected components with weights $w_1, \ldots, w_c$, and the components have been partitioned into S_1 and S_2, i.e. $S_1 \cup S_2 = \{1, \ldots, c\}$ and $S_1 \cap S_2 = \emptyset$. When assigning the nonzeros of S_1 to one processor and the nonzeros of S_2 to another, no communication volume is incurred.*

Proof. We are considering a bipartitioning here. Whenever the nonzeros of a row belong to different processors, we say this row is **cut**. Similarly, a column can also be cut. Whenever a row or column is not cut, it is said to be **uncut**. Note that each cut row or column incurs one communication volume when bipartitioning, and an uncut column incurs no communication volume. By construction, the bipartitioning created by connected components contains uncut rows and columns only, hence the total communication volume is zero. □

5.1 Finding connected components

Algorithm 7 provides a procedure that searches for connected components in a matrix. It requires the matrix A along with a maximum component weight $w_{\max}$. Setting $w_{\max} = nnz(A)$ will work, but setting a tighter $w_{\max}$ allows to terminate the algorithm early if we conclude no feasible partitioning can be found. More precisely, if some component has weight larger than the maximum weight of a partition, there is no way of fitting the component in either of the partitions. In practice this means we can terminate the algorithm halfway whenever the matrix exists of only one connected component.

The algorithm outputs the weights w_l of the components and a mapping from row number to component number, $R_r = l$ if row r is assigned to component l. Note that indeed, we do not assign individual nonzeros but entire rows, primarily because this requires less memory.

Finding the connected components of an $m \times n$ matrix A can be done in time and space linear in m, n and $nnz(A)$. Indeed, all rows are considered once, all nonempty columns are considered once and all nonzeros are considered twice, so the run time is $O(m) + O(n) + O(nnz)$. To see this, note that whenever a row or column enters the stack it is also assigned to a component, and a row or column can only enter the stack if it is not assigned to a component yet. Therefore, each row and nonempty column is considered exactly once. Furthermore, each nonzero is considered twice: once when going through its row and once when going through its column. This is also the reason we add $\frac{1}{2}$ to w_l each time we consider a nonzero. The memory space required for the algorithm is the space for R_r and C_c, which sums to $O(m) + O(n)$, and the space for the row and column stacks, which is at most $O(m)$ and $O(n)$, respectively.

We may relate this algorithm with a breadth-first-search on graphs. Given a matrix, consider the graph with a vertex for each row and for each column in the matrix. Moreover, for each nonzero (i, j) we draw an edge between the vertices corresponding to row i and column j. Note that this graph is bipartite. Now indeed, Algorithm 7 corresponds exactly to a breadth-first-search on this graph, which due to the bipartite nature, will go back and forth adding row vertices and column vertices to components.

5.2 Subset Sum

Once we have concluded that a matrix has $c > 1$ connected components with weights w_l, $l = 1, \ldots, c$, we want to distribute these components such that the balance constraint (7) is satisfied. As noted, this boils down to be a variant of the NP-complete Subset Sum problem. We implement two methods to solve this Subset Sum problem. The first is a direct approach that will try every possible combination of components to achieve two partitions obeying the balance constraint. The second method is an adaption of the Karmarkar-Karp algorithm, which is a heuristical approach.

Algorithm 7 DetectConnectedComponents

Search for connected components in A with weight at most $w_{\max}$.
Input: $m \times n$-matrix A, max weight w_{max}.
Output: Mapping R_r from rows to components, component weights w_l
Return: False if a component exists with weight greater that $w_{\max}$, true otherwise
procedure DetectConnectedComponents
    Initialize an empty row stack and column stack
    $R_r \leftarrow -1 \quad \forall r = 1, \ldots, m$
    $C_c \leftarrow -1 \quad \forall c = 1, \ldots, n$
    $i \leftarrow 0, l \leftarrow 0$
    while $i < m$ **do**
        if row stack empty **then**
            if $R_i > -1$ **then** ▷ Row is already assigned, skip it
                $i \leftarrow i + 1$
                continue
            if current component nonempty **then**
                Start new component: $l \leftarrow l + 1$
            Assign row i to current component: $R_i \leftarrow l$
            Add row i to row stack

        while row stack non-empty **do** ▷ Process all newly assigned rows
            $r \leftarrow$ pop(row stack)
            for all nonzeros (r, c) in row r **do**
                if $C_c = -1$ **then**
                    Assign column c to current component: $C_c \leftarrow l$
                    Add column c to column stack
                $w_l \leftarrow w_l + \frac{1}{2}$

        while column stack non-empty **do** ▷ Process all newly assigned columns
            $c \leftarrow$ pop(column stack)
            for all nonzeros (r, c) in column c **do**
                if $R_r = -1$ **then**
                    Assign row r to current component: $R_r \leftarrow l$
                    Add row r to row stack
                $w_l \leftarrow w_l + \frac{1}{2}$

        if current component weight w_l exceeds $w_{\max}$ **then**
            return false
    return true

5.2.1 Direct approach

The direct approach we implement is a recursive, exponential time algorithm, and can be found in Algorithm 8. It incorporates some checks to ensure we do not consider combinations of components of which we can a priori decide they will not lead to a valid weight. However, in the worst case it may still need $O(2^n)$ recursive calls when called with n weights. In an implementation, we accumulate the different sums, such that the sums in the procedure will not have to be calculated explicitly. This way, each recursion runs in $O(1)$ time. Note that it is assumed that the weights w_i are sorted, which leads to an extra term $O(n \log(n))$, hence the total running time is $O(2^n) + O(n \log n) = O(2^n)$. The main advantage of this algorithm is that whenever a feasible solution exists, it will be found.

Algorithm 8 SubsetSumExp. Search for a vector $x \in \{0,1\}^n$ such that $w_{\min} \leq \sum_{i=1}^{n} w_i x_i \leq w_{\max}$. Initial call is SubsetSumExp(w, $x = \vec{0}$, $j = 1$, $w_{\min}$, $w_{\max}$), with w sorted ascendingly.

```
1:  Input: Weights w_i (i = 1, ..., n), decision vector x ∈ {0,1}^n, current item j, min subset weight w_min,
    max subset weight w_max
2:  Output: Altered decision vector x
3:  Return: True if a feasible x exists, false otherwise
4:  procedure SubsetSumExp
5:      { Invariant(x, j − 1) }
6:      if Σ_{i=1}^{j−1} w_i x_i ≥ w_min then
7:          return true
8:
9:      if j ≥ n then
10:         return false
11:     if Σ_{i=1}^{j−1} w_i x_i + w_j > w_max then
12:         return false
13:     if Σ_{i=1}^{j−1} w_i x_i + Σ_{i=j}^{n} w_i < w_min then
14:         return false
15:
16:     x_j ← 1
17:     { Invariant(x, j) }
18:     if SubsetSumExp(w, x, j + 1, w_min, w_max) then
19:         return true
20:
21:     x_j ← 0
22:     { Invariant(x, j) }
23:     if SubsetSumExp(w, x, j + 1, w_min, w_max) then
24:         return true
25:     return false
```

To help understanding the algorithm, we introduce an invariant:

$$\text{Invariant}(x, k): \quad \sum_{i=1}^{k} w_i x_i \leq w_{\max}.$$

In the initial call, Invariant(x, 0) holds, as the sum over an empty set may be taken to be zero.

The algorithm assumes the weights w_i to be sorted ascendingly. Referring to Algorithm 8, the following checks are included:

1) By the invariant we know that $\sum_{i=1}^{j-1} w_i x_i \leq w_{\max}$. If in addition $\sum_{i=1}^{j-1} w_i x_i \geq w_{\min}$, we have found a feasible solution.

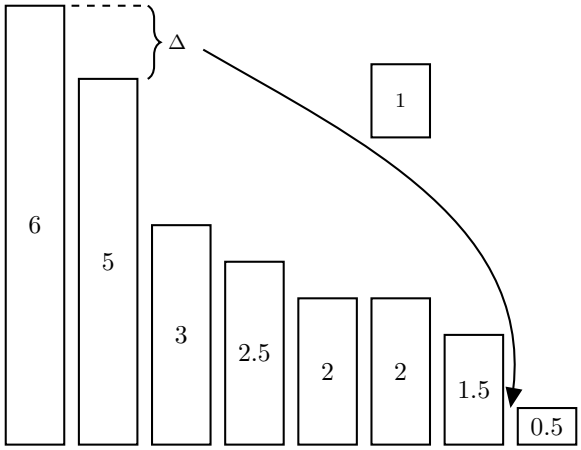


Figure 7: Illustration of one iteration of the differencing loop in the Karmarkar-Karp algorithm.

2) Of course, we should stop whenever we have reached the end of our list. If we reached it and 1) does not hold, we do not currently have a feasible solution.

3) If 1) does not hold and adding a new weight causes the selected weight to exceed the maximum weight, we may avoid considering any weights w_i with $i \geq j$, as $w_i \geq w_j$ for $i \geq j$. This check ensures that Invariant(x, j) will be true for both $x_j = 1$ and $x_j = 0$.

4) The last check computes whether there is still enough weight left to be added to reach $w_{\min}$. The maximum weight that can be selected, given what we have selected so far, equals the total weight currently selected, $\sum_{i=1}^{j-1} w_i x_i$, plus the weight that may still be included, $\sum_{i=j}^{n} w_i$. If this maximum weight does not exceed the minimum target weight, we may discard the current branch.

5.2.2 Karmarkar-Karp adaption

The Karmarkar-Karp algorithm [16] is an algorithm designed to solve the Partition problem, a special case of the Subset Sum problem where the target weight is exactly half of the total weight. It makes use of a max-heap and a graph to keep track of the internal state. A heap is a data structure of which we use two main methods. The first method is adding an item to the heap, which we call `add`; the second method is removing an item from the heap, also called a `pop` operation. The heap always keeps track of the maximum element in the list it is working on. The `pop` operation removes and returns this maximum element, and makes sure it keeps track of the next maximum element. The `add` method adds the new element and makes sure the list is updated accordingly.

An adaption of the Karmarkar-Karp algorithm is given in Algorithm 9 and a visualization of its core part, the differencing loop, is given in Figure 7. Note that the algorithm provided here is a modified version of the original Karmarkar-Karp algorithm. Firstly, it is able to handle the Subset-Sum problem rather than only the Partition problem. Secondly, it takes into account that the partitions are allowed to be within a prespecified range of weights, rather than imposing a single target weight. We will go through the algorithm and expand on its workings step by step.

Converting Subset-Sum to Partition Whenever P is even, we want to create two partitions of nonzeros with (nearly-)equal weight. However, when P is odd one partition will have to be larger than the other, i.e. we want to create two partitions with weights $w_{s(mall)}, w_{l(arge)}$ such that we have $w_l \geq w_s$. For each partition, we define a *target weight* w_s^t, w_l^t that we want w_s and w_l to be approximately equal to. How large

Algorithm 9 SubsetSumKK. Search for a vector $x \in \{0,1\}^n$ such that $w_s^{\min} \leq \sum_{i=1}^n w_i x_i \leq w_s^{\max}$. $w_s^{\min}$ and $w_s^{\max}$ are assumed to define the bounds for the weight of the smallest desired partition.

Input: Weights w_i $(i = 1, \ldots, n)$, min subset weight $w_s^{\min}$, max subset weight $w_s^{\max}$
Output: Decision vector $x \in \{0,1\}^n$, with $x_i = 1$ if weight i belongs to the smallest partition.
Return: True if a feasible x has been found, false otherwise
procedure SUBSETSUMKK
    Add (i, w_i) to a max-heap H, for all $i = 1, \ldots, n$
    Create a graph G with n vertices, each vertex $i = 1, \ldots, n$ will hold a colour value v_i (to be determined)
    $W \leftarrow \sum_{i=1}^n w_i$
    if $w_s^{\min} + w_s^{\max} \neq W$ **then**
        Add dummy weight $(n+1, w_{n+1})$ to H, where $w_{n+1} = W - (w_s^{\min} + w_s^{\max})$
        Add another vertex $n+1$ to G
    while Heap H contains more than 1 element **do**
        $(i, w_a) \leftarrow pop(H)$
        $(j, w_b) \leftarrow pop(H)$
        Add $(i, w_a - w_b)$ to H
        Add an edge between nodes i and j in G
    $(i, w_{res}) \leftarrow pop(H)$
    if $w_{res} > w_s^{\max} - w_s^{\min}$ **then**
        return false
    TwoColorTree(G) ▷ Assign 0 or 1 to all v_i such that whenever edge (i, j) exists in G, $v_i \neq v_j$
    if Dummy weight $n+1$ was added and $v_{n+1} = 0$ **then**
        $x_i \leftarrow 1 - v_i \quad \forall i = 1, \ldots, n$
    else
        $x_i \leftarrow v_i \quad \forall i = 1, \ldots, n$
    return true

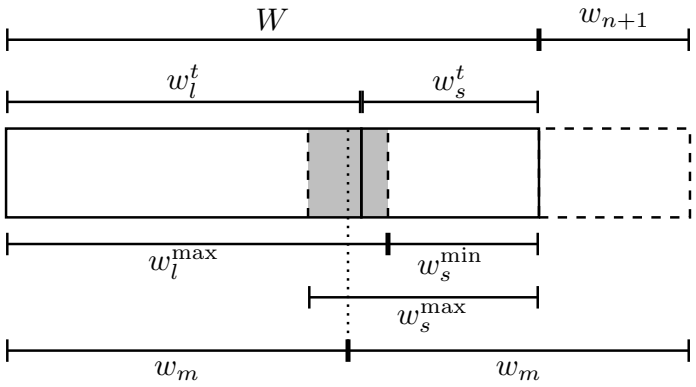


Figure 8: Illustration of the different weights used in the Karmarkar-Karp algorithm. The weight magnitudes come from the first iteration of Mondriaan with $P = 3$ and $\epsilon = 0.3$, using the `Increase` *load balance strategy. The target weights w_s^t, w_l^t and maximum weights $w_s^{\max}$, $w_l^{\max}$ for the partition weights $w_{s(mall)}$, $w_{l(arge)}$ are shown. Note that $w_s^t + w_l^t = w_s + w_l = w_l^{\max} + w_s^{\min} = W$, and that w_s^t equals one third of the total weight W, whereas w_l^t equals two thirds. The* `Increase` *load balance strategy determines that in this first split, $w_s^{\max} = (1+\epsilon)w_s^t = 1.3w_s^t$ and $w_l^{\max} = (1+0.25\epsilon)w_l^t = 1.075w_l^t$. The shaded area indicates at what values the interval may be cut, i.e. feasible values for w_l $(= W - w_s)$. Also the dummy weight w_{n+1} is shown. A dotted line is drawn at $w_m = \frac{1}{2}(W + w_{n+1})$, indicating the two partitions that Karmarkar-Karp aims to create. Note that w_m not necessarily equal to w_l^t. (Indeed, the* `Increase` *load balance strategy was chosen as it leads to $w_l^t \neq w_m$ in this case.)*

$|w_s - w_s^t|$ and $|w_l - w_l^t|$ may be is ultimately defined by the imbalance parameter ϵ, but here we assume that maximum weights $w_s^{\max}$ and $w_l^{\max}$ are given, such that a feasible partitioning must have $w_s \leq w_s^{\max}$ and $w_l \leq w_l^{\max}$.

Note that in the algorithm, another bounding variable $w_s^{\min}$ is used. This minimal weight for w_s equals $w_s^{\min} = W - w_l^{\max}$, where W denotes the total weight $W = \sum_{i=1}^{n} w_i$. We will be using $w_l^{\max}$ in the theory, but $w_s^{\min}$ in the algorithm to be consistent with the previous section. For an illustration of the introduced variables, refer to Figure 8.

As the Karmarkar-Karp algorithm is designed for the Partition problem, our adaption adds a dummy weight w_{n+1} whenever the two desired partitions have different maximum weights $w_s^{\max} \neq w_l^{\max}$. The magnitude of this dummy weight will be taken to be $w_{n+1} = w_l^{\max} - w_s^{\max}$. Note that $w_{n+1} = 0$ whenever the two maximum weights are equal. The expression for w_{n+1} in Algorithm 9 can be derived as follows:

$$w_{n+1} = w_l^{\max} - w_s^{\max} = W - (W - w_l^{\max} + w_s^{\max}) = W - (w_s^{\min} + w_s^{\max}).$$

We will use w_s (and derived quantities) for the weight of the small partition without w_{n+1}, and $\widetilde{w}_s$ (and derived quantities) for the small partition including the dummy w_{n+1}.

The differencing loop After we have set up our problem, the differencing loop commences, as depicted in Figure 7.

Lemma 5.3. *After the while loop in Algorithm 9, the graph G is a tree.*

Proof. Suppose the graph has n nodes. The graph is a tree if and only if it has $n - 1$ edges and it contains no cycles.[1] In each iteration of the while loop, two vertices are removed from the heap and one is added.

[1] These two requirements automatically imply the graph will be connected.

Hence the loop runs $n-1$ times before only one vertex is left and so $n-1$ edges are created. To prove there are no cycles, we proceed by induction. Note that initially, there are no edges hence there are no cycles. Suppose now we have added k edges and we want to add another edge. Whenever an edge was added, one of its incident nodes was removed from the heap and one was removed and reinserted into the heap. Hence whenever a path exists in the graph, at most one of the nodes in it can also be present in the heap. Suppose now that adding a certain edge between nodes i and j would form a cycle. As we add an edge between i and j, both i and j should still be in the heap. However, because edge (i,j) creates a cycle, in the graph a path should exist between nodes i and j. In this path at most one node can be present in the heap, hence i and j cannot both be present in the heap. This is a contradiction and hence our assumption that adding (i,j) creates a cycle cannot be true. As there were no cycles after k edges were added and in iteration $k+1$ no cycles were created, we also have no cycles after $k+1$ edges are added. This concludes the proof that G is a tree. □

Now we have decided that G in the algorithm is a tree, we can continue analyzing the algorithm. An edge (i,j) between two weights signifies the fact we wish to place weights i and j in different partitions, however we postpone the decision which weight will be assigned to which partition. Adding $w_a - w_b$ back to the heap signifies that we add the weight w_b to both partitions, and will add an additional weight $w_a - w_b$ to only one of the partitions. In the end, this amounts to assigning w_b and $w_b + (w_a - w_b) = w_a$ to the two partitions.

Checking the solution When the while loop finishes, the heap consists of a single element w_{res}, the weight of which equals the difference in weight between the two partitions that will be generated, i.e. $w_{res} = |w_l - \widetilde{w}_s|$. In the following lemma, this residual weight w_{res} plays a central role.

Lemma 5.4. *Using dummy weight $w_{n+1} = w_l^{\max} - w_s^{\max}$, Algorithm 9 creates partitions with weights $w_s \leq w_s^{\max}$ and $w_l \leq w_l^{\max}$ if and only if $w_{res} \leq w_s^{\max} - w_s^{\min}$.*

Proof. We start with the condition $w_s \leq w_s^{\max} \wedge w_l \leq w_l^{\max}$, and work our way to the other condition by using a series of equivalences.

First off, note that the small partition is taken to be the partition that w_{n+1} is in if $w_{n+1} > 0$, or either of the partitions if $w_{n+1} = 0$. In either case, we have $\widetilde{w}_s = w_s + w_{n+1} = w_s + w_l^{\max} - w_s^{\max}$. This equality implies that the condition $w_s \leq w_s^{\max}$ is equivalent to $\widetilde{w}_s \leq w_l^{\max}$.

Secondly, the target weight the Karmarkar-Karp algorithm works towards is

$$w_m = \frac{W + w_{n+1}}{2} = \frac{w_l + \widetilde{w}_s}{2}.$$

Define $\Delta w = \frac{w_{res}}{2} \geq 0$ and note that

$$|w_l - w_m| = \frac{|w_l - \widetilde{w}_s|}{2} = \frac{w_{res}}{2} = \Delta w,$$

and the same holds for $|w_m - \widetilde{w}_s|$. Hence we either have

$$\begin{cases} \widetilde{w}_s &= w_m + \Delta w \\ w_l &= w_m - \Delta w \end{cases} \quad \text{or} \quad \begin{cases} \widetilde{w}_s &= w_m - \Delta w \\ w_l &= w_m + \Delta w \end{cases}.$$

Note that $\widetilde{w}_s \leq w_l^{\max} \wedge w_l \leq w_l^{\max}$ is equivalent to $\max\{\widetilde{w}_s, w_l\} \leq w_l^{\max}$, and as $\max\{\widetilde{w}_s, w_l\} = w_m + \Delta w$ in both alternatives above, this condition is equivalent to $w_m + \Delta w \leq w_l^{\max}$.

Lastly, we rewrite this last condition $w_m + \Delta w \leq w_l^{\max}$ into the following:

$$\Delta w \leq w_l^{\max} - w_m \;=\; w_l^{\max} - \frac{W + w_l^{\max} - w_s^{\max}}{2} \;=\; \frac{w_l^{\max} + w_s^{\max} - W}{2} \;=\; \frac{w_s^{\max} - w_s^{\min}}{2},$$

which is equivalent to $w_{res} \leq w_s^{\max} - w_s^{\min}$. □

Building the solution If we conclude that the partitions we can generate obey the maximum weights, we continue building the actual partitioning. The decision which weight goes to which partition is dictated by the tree that has been built. Whenever an edge exists between nodes i and j, we have decided to put them in different partitions. Hence we should assign numbers 0 and 1 to the nodes such that whenever an edge (i, j) exists, the node values v_i and v_j are unequal. This is exactly what the procedure `TwoColorTree` does. In the end the algorithm does some post processing to make sure $x_i = 1$ for the weights in the small partition and $x_i = 0$ for the weights in the large partition.

One can verify that each of the steps in the algorithm runs in $O(n)$ time, except for the heap-related operations which together take $O(n \log n)$ time. The memory space required is of order $O(n)$. The algorithm works well in cases where the generated differencing weights become small enough to fine tune the solution. Whenever the weights remain large throughout the entire algorithm, it may not succeed in finding a feasible solution.

5.3 Results

To test the ZVS algorithm, we use our standard set of test matrices with a number of additional matrices. For $P = 2$ and $P = 3$, we determined by hand for all odd- and even-numbered matrices whether a zero volume partitioning is theoretically possible. All odd-numbered matrices for which a zero volume partitioning is possible are added to the set of test matrices. These matrices are added to obtain more results from matrices that admit a zero volume partitioning, as the number of matrices in the standard set is relatively low. Note however that some of the results presented here may be a little biased due to these added matrices.

Test runs in this section are performed on machine 1 for timing and machine 3 for quality. Whenever we want to perform a Subset Sum algorithm with 16 weights or less we perform the exponential algorithm, for cases with more than 16 weights Karmarkar-Karp is used. For 16 weights, it was experimentally determined that the exponential algorithm still has acceptable performance, but a higher threshold would be inadvisable.

5.3.1 Applying ZVS to bipartitioning

We first take a look at bipartitioning. As we mentioned we perform test runs on the standard test set and any additional matrices that admit a zero volume bipartitioning (ZVBP), in total leading to a set of 994 matrices. Of these matrices, 213 contain at least two connected components. For 79 of these matrices, the `DetectConnectedComponents` procedure returns true, signifying that there might be a zero volume partitioning. In 76 of these matrices, a zero volume partitioning is actually possible. In all of these 76 cases, our zero volume search implementation finds such an optimal partitioning obeying the maximum load imbalance of 0.03. For 69 of these matrices, Mondriaan without ZVS also found a zero volume partitioning in all performed calculations, though taking more time to do so. Hence in the other 7 cases, ZVS improves on both quality and speed.

We present results for the case a zero volume partitioning is possible and the case it is not possible in Figure 9. From the left figure, we can conclude that Mondriaan with ZVS is much faster than Mondriaan without ZVS, when matrices with feasible zero volume partitionings are considered. Speed-ups range from a factor 2 to a factor 30. From the right figure, we can conclude that the run time penalty of including ZVS is only 1.4% in cases where the algorithm does not yield a result. We can also observe that the relative time taken in ZVS increases slightly with the number of nonzeros. Whether this correlation really exists is not clear however due to the relatively low total number of runs. Hence more runs would have to be performed if we wish to establish whether this is a true correlation.

5.3.2 Applying ZVS to tripartitioning

To test tripartitioning, we take the same matrix set as for $P = 2$, but with in addition any matrices that either allow a zero volume tripartitioning or of which it is known that the first Mondriaan recursion should find a ZVBP, taking the number of matrices to 1004. Of these matrices, 37 allow a zero volume tripartitioning,

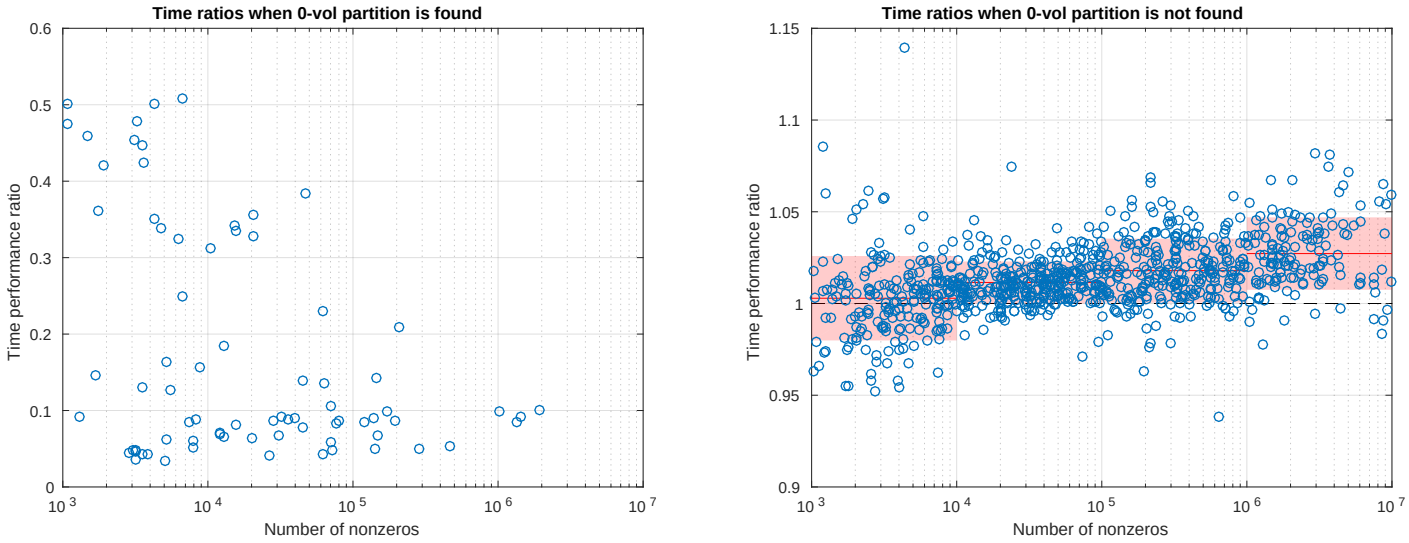


Figure 9: Ratio of running times when a ZVBP is possible (left) and when it is not (right), for the case $P = 2$ and $\epsilon = 0.03$, averaged over 5 runs. The ratios are calculated by dividing the running times of Mondriaan with ZVS by the running times of Mondriaan without ZVS. The mean of all ratios in the left figure is 0.1732. Other means are presented in Table 2.

$\log_{10}(nnz)$	Mean time performance ratio		
	$P = 2$	$P = 3$	$P = 16$
$3 - 4$	1.0029 ± 0.0230	1.0193 ± 0.0216	1.0201 ± 0.0162
$4 - 5$	1.0115 ± 0.0114	1.0220 ± 0.0115	1.0249 ± 0.0096
$5 - 6$	1.0179 ± 0.0175	1.0255 ± 0.0198	1.0311 ± 0.0163
$6 - 7$	1.0272 ± 0.0197	1.0296 ± 0.0236	1.0352 ± 0.0233
$3 - 7$	1.0138 ± 0.0193	1.0235 ± 0.0189	1.0273 ± 0.0165

Table 2: Timing ratio means belonging to Figures 9, 10 and 11, for the cases no zero volume partitionings are found.

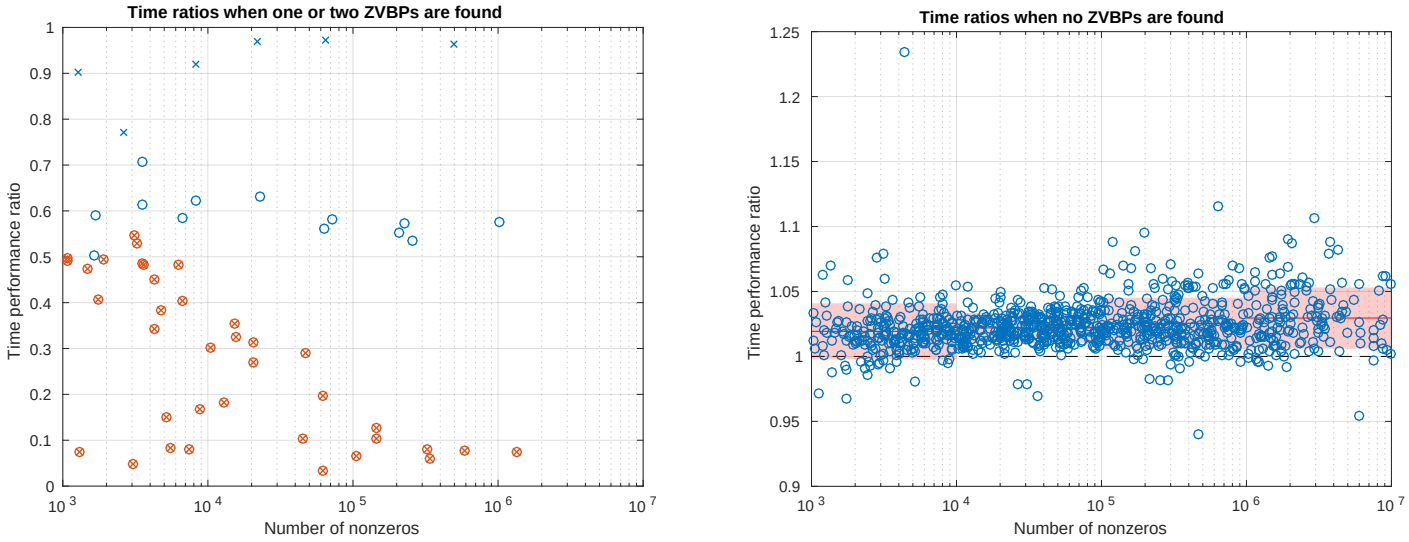


Figure 10: Ratio of running times when a ZVBP is possible (left) and when it is not (right), for the case $P = 3$ and $\epsilon = 0.03$, averaged over 5 runs. In the left figure, circles represent matrices for which in the first iteration a zero volume partition is found (mean 0.5869), crosses represent the same for the second iteration (mean 0.9173), and a combined cross and circle represent runs where in both iterations a ZVBP was found (mean 0.2712), leading to a zero volume tripartitioning. Means for the right figure are given in Table 2.

and for all of these Mondriaan with ZVS finds such a zero volume tripartitioning, opposed to 32 when ZVS is not used. Note that for these matrices, our ZVS algorithm finds a ZVBP twice, as Mondriaan performs two iterations to obtain $P = 3$.

For 15 other matrices, it is possible to construct a tripartitioning where one partition does not have any communication with the other partitions, while the other partitions do need to communicate with each other. In other words, Mondriaan should be able to find a zero volume partitioning once, but not twice. For 12 of these 15 matrices Mondriaan indeed does so, while for the other three it does not (`Rothberg/struct3`) or not always[2] (`Meszaros/gams30am` and `Meszaros/gams60am`) find a ZVBP. Remarkable is that these failures are actually not due to the Subset Sum procedure not succeeding, but due to Mondriaan imposing a more strict ϵ in the first iteration, in order to maintain feasibility in later iterations. Indeed, if we increase ϵ to 0.054, 0.049, and 0.047 for the three respective matrices, Mondriaan with ZVS does succeed in finding the present ZVBP. Also, the few times the mentioned `Meszaros` matrices were successfully bipartitioned once, this bipartitioning occurs in the second iteration and not in the first, indicating that the first iteration created a feasible ZVBP for the second iteration.

Timing results are presented in Figure 10. The 37 matrices for which Mondriaan with ZVS finds a zero volume tripartitioning run within a factor 2 to 30 faster just as with $P = 2$, with a mean of 73% speed improvement. When a ZVBP is only found in one iteration, the speed improvement is less but still present. When no zero volume partition is found, the running time increases on average with 2.4%.

The speed improvement is less when only in the second iteration a ZVBP is found compared to when it is found in the first iteration. This can be explained by the fact that in the second iteration, the submatrix to be partitioned is smaller and hence there is less improvement to be gained.

[2]Depending on the random number generator (RNG) seeds used.

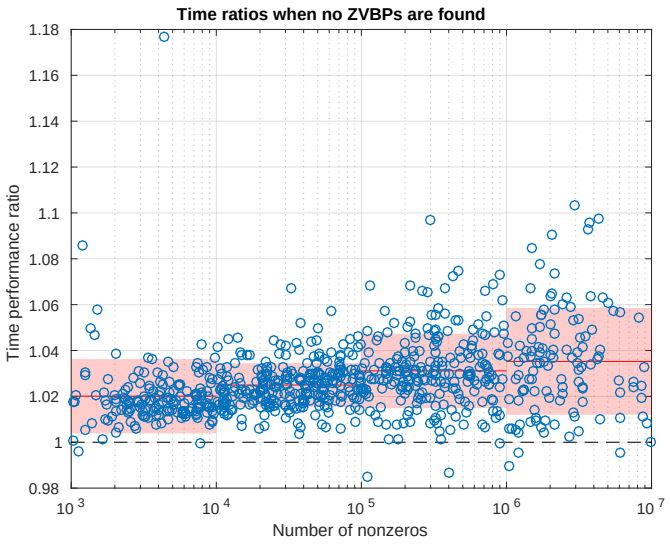


Figure 11: Ratio of running times when no ZVBPs are found, for the case $P = 16$ and $\epsilon = 0.03$, averaged over 5 runs. Numeric values for the means are given in Table 2.

5.3.3 Applying ZVS when $P = 16$

In Figure 11, we present results on the time performance ratio when no ZVBPs are found, using the same set of matrices as in Section 5.3.2 but for $P = 16$. Note that the time performance ratios are higher than for $P = 2$ and $P = 3$, and seem to tend to increase with P. This may be explained by the fact that the `DetectConnectedComponents` procedure has a time complexity linear in the number of rows m, which does not decrease with the depth of the bipartitioning.

In Section 5.3.2, we have seen that for $P = 3$ the running times depend on whether a ZVBP has been found in the first or in the second iteration. For $P = 16$, this is even more the case, as there are $\log_2(16) = 4$ distinct depths at which Mondriaan works. In each of the 15 bipartitionings, we either may or may not find a ZVBP, leading to a total of $\prod_{d=0}^{3}(2^d + 1) = 270$ different combinations of numbers of ZVBPs found per depth. As we do not have enough matrices to cover all these combinations in a sufficient matter, we do not present timing results for the case ZVBPs are found. Instead, in the next section we will be looking in more detail into solution quality ratios for $P = 16$ and $P = 64$.

5.3.4 Solution quality

We have already seen that the ZVS algorithm finds ZVBPs faster than Mondriaan's multilevel hypergraph method. In this section, we will be looking into solution quality rather than running times. Recall that in Section 5.3.2, we have seen that ZVBPs may be found in later recursions, even if there were no feasible ZVBPs in earlier recursions. Here we will look further into this phenomenon.

Let the matrix A be given by

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}.$$

When bipartitioning this matrix with $\epsilon = 0$, an optimal bipartitioning has a communication volume of 2. This is easily seen from the matrix[3], but can also be verified with MondriaanOpt. Suppose we partition the

[3]Both partitions must contain 4 nonzeros. Suppose that the first or second column is assigned entirely to one partition, say A_0, then still 4 nonzeros have to be assigned to the other (A_1). These nonzeros must be in both rows as there are not enough

	Depth d	$P = 16$		$P = 64$	
$c = 1$	0	0.0	(0.00%)	0.0	(0.00%)
	1	5.0	(0.36%)	6.0	(0.47%)
	2	12.8	(0.46%)	18.9	(0.73%)
	3	48.0	(0.86%)	42.3	(0.82%)
	4			94.7	(0.92%)
	5			250.9	(1.22%)
$c > 1$	0	72.0	(24.66%)	64.0	(23.44%)
	1	69.0	(11.82%)	58.6	(10.73%)
	2	136.9	(11.72%)	116.4	(10.66%)
	3	275.0	(11.77%)	228.3	(10.45%)
	4			415.2	(9.51%)
	5			910.6	(10.42%)

Table 3: Number of ZVBPs found at different recursion depths d, counted over all matrices and averaged over 10 runs, using $\epsilon = 0.03$. Also percentages are given, indicating what percentage of the total number of bipartitionings lead to a ZVBP. These percentages are calculated by dividing the presented counts by the total number of matrices and also dividing by 2^d as that is the number of bipartitionings performed per matrix at the given depth. The upper part of the table corresponds to results over all matrices consisting of a single connected component, the lower part of the table corresponds to all matrices with more than one connected component. For $P = 16$, the numbers of matrices for $c = 1$ and $c > 1$ are respectively 701 and 292, while for $P = 64$ these are 642 and 273 (note that for $P = 64$ and $\epsilon = 0.03$, Mondriaan tends to fail for more matrices).

matrix such that the first two columns are assigned to A_0 and the other columns are assigned to A_1. Clearly, this partitioning has $\epsilon' = 0$ and $V = 2$ and hence is optimal (i.e., it is a partitioning Mondriaan might generate). When recursively bipartitioning the generated partitions again, we now observe that partition A_1 can be partitioned with zero volume, assigning each row to a different partition.

The question that arises here is whether such situations often occur in practice, and secondly whenever ZVS finds these created zero volume partitionings, whether this will ultimately improve the solution quality.

In Table 3 we see the number of ZVBPs found at different depths d during partitioning with $P = 16$ and $P = 64$. We see that for matrices with one connected component, the number of ZVBPs found increases with d. Also as a percentage, the number of ZVBPs found increases. However, the percentages themselves are small, only exceeding 1% at depth 5 of $P = 64$. Strictly speaking, this answers one of our questions: ZVBPs can be found in deeper recursions, even if they are not present at first. However, the number of these ZVBPs is not very large.

When the number of components is larger than one, the situation changes. In almost all cases, the number of ZVBPs found at a certain depth exceeds 10% of the total number of bipartitionings performed, even exceeding 20% at depth 0. Note that depth 0 is related to Section 5.3.1.

We now have a look at the effect the found ZVBPs have on the communication volume. The results can be found in Table 4. Note that in general, higher values of P seem to benefit more from the ZVS algorithm. The reported 1.0002 being larger than 1 is caused by different sequences of RNG being used, as the ZVS algorithm may also incur calls to the RNG. Matrices that admit a ZVBP at depth 0 (type 3) clearly benefit more from the ZVS algorithm than other matrices (type 1 and 2), their volume reductions averaged over all P being respectively 7% and 0.2%. In total, the volume reduction averaged over all matrices and values of P comes down to 0.7%.

nonzeros left in a single row, hence both rows are cut, leading to at least 2 communications. Suppose now on the other hand that the first and second column are both not assigned to one partition, then both columns are cut and hence we have at least 2 communications. A partition with volume 2 would be to assign the first two columns to A_0 and the others to A_1.

P	Type 1	Type 2	Type 3	Total
2	1.0000	1.0002	0.8947	0.9920
3	0.9999	0.9975	0.9540	0.9959
4	0.9991	0.9997	0.9386	0.9946
16	0.9981	0.9984	0.9121	0.9916
64	0.9961	0.9924	0.9384	0.9911

Table 4: Volume ratios, averaged over (type 1) all matrices consisting of a single component, (type 2) all matrices consisting of more than one component, but which do not admit a ZVBP in the first recursion for $P = 2$ and (type 3) all matrices that do admit such a ZVBP for $P = 2$. Also total averages are given over all matrices. All results are averaged over 10 runs. The numbers of matrices per type are approximately 700, 220 and 70, respectively. Whenever both the average old and new volume are zero, we define their ratio $\frac{0}{0} = 1$ as no improvement is made.

5.4 Conclusion

In this section, we have discussed a zero volume search (ZVS) algorithm, consisting of an algorithm to search for connected components and two algorithms for solving the arising variant of the subset sum problem. For matrices admitting no zero volume bipartitioning (ZVBP), the overhead of applying the ZVS algorithm increases running times by a few percent, ranging from 1.4% for $P = 2$ to 2.7% for $P = 16$. Matrices that do admit one or multiple ZVBPs have a large benefit from the algorithm, with speed-ups ranging from a factor 2 to 30 for both $P = 2$ and $P = 3$.

It is found to be theoretically and practically possible that Mondriaan generates ZVBPs in matrices that originally do not consist of multiple components. However, using the ZVS algorithm to find these ZVBPs leads to an average volume reduction not higher than 0.2%, at the cost of increasing running times by a few percent. In this regard, the volume improvement due to the ZVS algorithm does not weigh up against the increased running time it induces.

The advice for a feature release of Mondriaan would be to include three options in the software: either perform ZVS at each recursion, never perform ZVS, or perform ZVS in each recursion until it arrives at a recursion for which it cannot find a ZVBP. In other words, the last option will search for ZVBPs as long as all previous ZVS calls were successful. Shipping Mondriaan with this last option as default, Mondriaan will be able to find easy partitionings efficiently without increasing running times too much.

6 Free nonzero search

In [22], an algorithm is introduced which computes an optimal solution to the bipartitioning problem (Problem 1.4 with $P = 2$). In the proposed algorithm, a solution to the problem is characterized by the distribution of the rows and columns of the matrix, rather than assignments of individual nonzeros to partitions. A row or column can be in either of three states: assigned to processor 0, assigned to processor 1 or 'cut', indicating it is assigned to both processors. Each cut row or column leads to exactly one communication volume, hence the total communication volume equals the number of cut rows and columns.

Whenever a nonzero's row and column are both cut, it does not matter for the total communication volume whether we assign the nonzero to processor 0 or 1, as in both cases it will lead to the same amount of communication. In the paper, such nonzeros are referred to as *free nonzeros*. As the assignment of those free nonzeros is not important for the total communication volume, distributions resulting from the in [22] proposed algorithm explicitly distinguish between free and assigned nonzeros. Hence the resulting distribution is not an explicit distribution, but rather a template for generating an explicit distribution: one still has to fill in the blanks. An advantage of this is that when distributing the free nonzeros, one may do so in a way that optimizes some secondary objective. Indeed, in the latest release of MondriaanOpt, a simple algorithm is included which can generate an explicit distribution that assigns free nonzeros in a way that minimizes load imbalance.

It is exactly this secondary objective that we want to include in the Mondriaan package. Suppose we bipartition a matrix with maximum load imbalance ϵ, and we obtain some bipartitioning with volume V and load imbalance $\epsilon' \leq \epsilon$. In other words, among all bipartitionings with load imbalance at most ϵ, we have found one with volume V. From this solution we want to proceed with the two quantities interchanged: among all bipartitionings with volume at most V, we want to find another bipartitioning with load imbalance lower than ϵ'.

There are multiple ways one may approach this. Currently in Mondriaan, after each bipartitioning a single run of the KLFM algorithm is performed to obtain 'easy improvements' on the obtained bipartitioning. While this method is focused on minimizing the primary objective, the total communication volume, one may adapt it to also include the load imbalance as a secondary objective. Our approach will be based on the concept of free nonzeros. As a free nonzero may be assigned to either partition without changing the total communication volume, we may search for these free nonzeros and move them to another partition if this benefits the load imbalance.

Note that when using recursive bipartitioning, the size of the search space in a particular iteration is dependent on the achieved load balance in the previous iteration. Indeed, if we found a bipartitioning with nearly maximum imbalance, in the next iteration we have little freedom in the amount of imbalance we can afford. On the other hand, if we found a bipartitioning with nearly perfect balance, we have more freedom in the amount of imbalance we can afford in the next iteration. Our expectation is that such a bigger search space may gain us access to better bipartitionings in terms of total communication volume and hence may benefit the final total communication volume.

We will present two methods, and analyze and compare their performance within the Mondriaan package. Note that while most of this thesis is primarily focused on accelerating Mondriaan, this particular feature does not serve this purpose but it is interesting enough to consider.

6.1 Local method

We present two algorithms that both implement a *'free nonzero search'*, both searching for free nonzeros and moving them between partitions if it benefits the load balance. The first method we will discuss is the 'local' method, as opposed to the 'global' method that will be discussed in the next section.

The local method is quite straightforward. We consider a set of nonzeros bipartitioned into two partitions A_0, A_1, assuming without loss of generality that $nnz(A_0) \leq nnz(A_1)$. Note that whenever $nnz(A_0) = nnz(A_1)$, we have perfect load balance and hence there is nothing to improve, hence we may assume $nnz(A_0) < nnz(A_1)$. Our algorithm now exists of two steps. Firstly, we iterate through all nonzeros in

A_0 and compute the variables r_i and c_j, which indicate respectively which rows and which columns contain nonzeros in partition A_0:

$$r_i = \begin{cases} 1 & \text{if } \exists j \in [0,n) \text{ s.t. } (i,j) \in A_0 \\ 0 & \text{otherwise} \end{cases} \quad \forall i \in [0,m],$$

$$c_j = \begin{cases} 1 & \text{if } \exists i \in [0,m) \text{ s.t. } (i,j) \in A_0 \\ 0 & \text{otherwise} \end{cases} \quad \forall j \in [0,n].$$

Then, we iterate through all nonzeros in A_1 and move those nonzeros $(i,j) \in A_1$ to partition A_0 for which $r_i = 1$ and $c_j = 1$. Indeed, as (i,j) is assigned to A_1 and $r_i = c_j = 1$, both row i and column j are cut and hence nonzero (i,j) is free. We keep track of the weights of A_0 and A_1 while iterating, and stop whenever we reach perfect load balance. The resulting algorithm can be found in Algorithm 10.

Algorithm 10 FNZLocal()

```
1:  Improve load balance of a partitioning by moving free nonzeros.
2:  Input: Partitions A_0, A_1 that represent respectively np_0 and np_1 processors.
3:  Output: Modified partitions A_0 and A_1
4:  procedure FNZLocal
5:      if nnz(A_0)/np_0 > nnz(A_1)/np_1 then
6:          swap(A_0, A_1)
7:      if (nnz(A_0) + 1)/np_0 ≥ (nnz(A_1) − 1)/np_1 then
8:          return
9:      r_i ← 0 ∀i ∈ [0, m)
10:     c_j ← 0 ∀j ∈ [0, n)
11:     r_i ← 1, c_j ← 1 ∀(i, j) ∈ A_0
12:     for (i, j) ∈ A_1 do
13:         if r_i = 1 ∧ c_j = 1 then
14:             Move (i, j) to A_0
15:             if (nnz(A_0) + 1)/np_0 ≥ (nnz(A_1) − 1)/np_1 then
16:                 return
```

This algorithm is performed after every bipartitioning, between the two partitions that have been newly created. From the algorithm, we may directly deduce that the time complexity is $O(m)+O(n)+O(nnz(A_0))+O(nnz(A_1))$, and the space complexity is $O(m)+O(n)$ for the r_i and c_j variables.

Note that the local method (and the global method in the next section too) uses a slightly weakened definition of free nonzeros. In fact, the communication volume can change due to moving free nonzeros, however it can only decrease and not increase. For an example, see Figure 12.

6.2 Global method

The local method takes two partitions and moves free nonzeros from one partition to the other until either load balance is perfect or there are no free nonzeros left to move. This is a good strategy when performing a bipartitioning, but when we are doing a k-way partitioning, we might want to consider all partitions when shifting free nonzeros, not just the two newly created partitions. This is what we aim to do with the global method.

Free nonzeros have been introduced in a bipartitioning setting, so we first have to generalize the definition. Consider a k-way partitioned matrix. A nonzero (i,j) is called *free for partition l* whenever partition l contains at least one nonzero in both row i and column j. Note that a nonzero assigned to partition p is in this definition always free for partition p. Whenever a nonzero is free for at least two partitions (including the partition it is currently assigned to), we have the possibility to move the nonzero to another partition

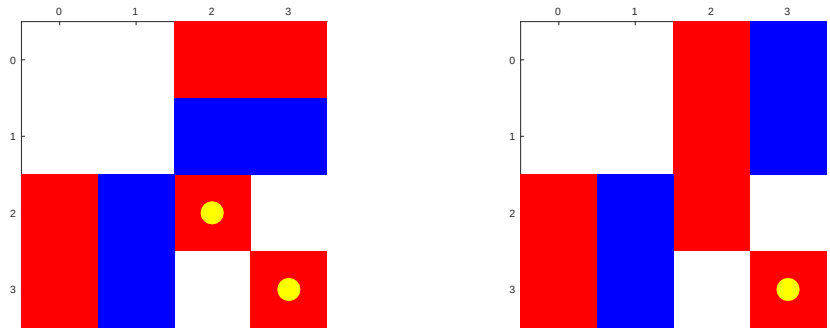


Figure 12: Examples of free nonzeros (marked yellow) in a bipartitioning over a red and a blue processor. In the left figure, the two diagonal red nonzeros (2,2) and (3,3) are free to both processors, hence one of these can be moved to the blue processor, leading to better load balance without affecting the communication volume. These nonzeros would also be considered free in MondriaanOpt. (However, the solutions above are not optimal, so MondriaanOpt will never generate these partitionings.) In the right figure, nonzero $(2, 2)$ *is not free but* $(3, 3)$ *still is free according to Algorithm 10. Moving it to the blue processor again leads to better load balance, but now does decrement the communication volume with one, as the last column is not cut any more. Note that this can not happen with free nonzeros in MondriaanOpt, as this type of free nonzeros can not appear in optimal solutions.*

without increasing the volume, hence we call it a free nonzero in that case. Note that for $k = 2$ this definition is consistent with the definition given earlier. Finally we define the *free degree* of a nonzero to be the number of partitions the nonzero is free for (this includes the partition it is currently assigned to).

In line with Mondriaan's origins, suppose each partition l is given a colour. Now we may define a colour l to be a *free colour* for nonzero (i, j) precisely when this nonzero is free for partition l. In this context, moving a nonzero to a partition it is free for is equivalent with assigning a free colour to the nonzero.

The global algorithm will now be roughly as follows: first determine the free degree of all nonzeros of matrix A, and gather all nonzeros with free degree at least 2. Sort these in ascending order of free degree, and go through the sorted list, moving a nonzero to a partition it is free for whenever there is a partition with less nonzeros than the partition it is currently in. The code can be found in Algorithm 11.

To determine the free degrees of all nonzeros, two matrices r and c are used in Algorithm 11. Note that these matrices have the same nonzero structure as the *communication matrix* [26][6]. These matrices are sparse, but for the intents of this thesis we will implement them as full matrices.

We experimented with three different sorting rules. The first rule is the rule we already mentioned, sorting ascendingly on the free degree of the nonzero. This way, the nonzeros with the least flexibility are considered first, and the nonzeros with the most flexibility are considered last. This choice can be motivated as follows. As during the algorithm nonzeros are moved, also free degrees are bound to change. Hence when you would consider the nonzeros with the least flexibility last, it is likely to happen that the few parts they could move to have already been filled with nonzeros that had more flexibility, likely preventing the last nonzeros from being moved. When sorting ascendingly, the least flexible nonzeros may be moved to smaller partitions first and the more flexible nonzeros may be assigned to other small partitions that otherwise would not have been considered. Hence sorting ascendingly by free degree will be the rule of choice.

However, it is likely that multiple nonzeros have equal free degrees. Hence, we may put effort in including a secondary sorting rule, in case the free degrees are equal. Note that we may wish to switch nonzeros in large partitions first, causing the imbalance to decrease directly, and then consider nonzeros in smaller partitions. Hence a secondary rule may be to sort descendingly on the number of nonzeros in a partition. A third sorting function we consider involves the same two sorting rules as the second sorting function exists of, but with roles switched: the primary sorting rule is on the number of nonzeros, and secondarily we sort on the

Algorithm 11 FNZGlobal()

Improve load balance of a partitioning by moving free nonzeros.
Input: Partitions $A_0, \ldots, A_{P-1}$ that represent respectively $np_0, \ldots, np_{P-1}$ processors.
Output: Modified partitions $A_0, \ldots, A_{P-1}$
procedure FNZGlobal
    $r_{i,p} \leftarrow 0 \quad \forall i \in [0, m-1], p \in [0, P-1]$ ▷ $r_{i,p}$ = Number of nonzeros in row i assigned to partition p
    $c_{j,p} \leftarrow 0 \quad \forall j \in [0, n-1], p \in [0, P-1]$ ▷ $c_{j,p}$ = Number of nonzeros in column j assigned to partition p
    for $p \in [0, P-1]$ **do**
        for $(i, j) \in A_p$ **do**
            $r_{i,p} \leftarrow r_{i,p} + 1$;
            $c_{j,p} \leftarrow c_{j,p} + 1$;

    $F \leftarrow \{\}$
    for $(i, j) \in A$ **do**
        $f \leftarrow |\{q \in [0, P-1] : r_{i,q} > 0 \wedge c_{j,q} > 0\}|$ ▷ f = Free degree of (i, j)
        if $f = 1$ **then**
            continue
        $F.add(((i, j), f))$

    Sort F ascendingly by free degree f

    for $s \in [0, |F| - 1]$ **do**
        $((i, j), \sim) \leftarrow F.get(s)$ ▷ Retrieve sth nonzero in sorted F
        $p \leftarrow$ the partition (i, j) belongs to
        $q \leftarrow \arg\min_{q'} \{nnz(A_{q'})/np_{q'} \mid r_{i,q'} > 0 \wedge c_{j,q'} > 0\}$
        if q does not exist or $p = q$ **then**
            continue
        if $(nnz(A_q) + 1)/np_q > (nnz(A_p) - 1)/np_p$ **then**
            continue
        Move (i, j) from part A_p to A_q
        $r_{i,q} \leftarrow r_{i,q} + 1$
        $c_{j,q} \leftarrow c_{j,q} + 1$
        $r_{i,p} \leftarrow r_{i,p} - 1$
        $c_{j,p} \leftarrow c_{j,p} - 1$

free degree.

Unfortunately, the two more complicated sorting rules did not improve upon the simpler first sorting rule. Therefore we will henceforth only be using the first sorting rule.

To obtain time and space complexities, denote the number of free nonzeros found in Algorithm 11 by $|F|$. Then the space complexity can be derived to be $O((m+n)P)+O(|F|) = O((m+n)P)+O(nnz)$, while the time complexity is given by

$$O(nnz) + O(nnz \cdot P) + O(|F| \log |F|) + O(|F| \cdot P) = O(nnz \cdot P) + O(nnz \log(nnz)).$$

Note that this complexity is considerably higher than for the local method. However, as the global method works on all partitions rather than just two, we do not need to perform the global method after each bipartitioning. Rather, we perform the global method only once a recursion depth has finished. This way, when computing P partitions, we only have $\lceil \log P \rceil$ depths, hence the global method is only performed $\lceil \log P \rceil$ times. When we sum the time complexities of the local method over a certain depth $d \leq \lceil \log P \rceil$, we obtain a total time complexity of $2^d O(m) + 2^d O(n) + O(nnz(A))$. Comparing this to the time complexity of the global method, we see that the local method still has better complexity behaviour. Notice that the space complexity of the local method is also better than for the global method. This can be solved by implementing r and c as sparse matrices, but the time complexity will then be typically worse.

The complexities of the global method are so high that the method may not be considered a feasible candidate in production code. However, we will consider both algorithms to be able to compare their practical results.

6.3 Results

We perform tests with Algorithms 10 and 11, on machine 2 for timing and 3 for quality. The matrices considered are taken from the standard test set.

First we take a look at the number of free nonzeros found and moved in the tests, see Figure 13. Clearly, the global method is able to find large amounts of free nonzeros, and this amount increases with the depth at which the method is applied. This can be explained by the global nature of the algorithm; with more partitions it is more likely that a nonzero is free to at least two partitions. The number of moved nonzeros also increases with the depth, while the global method generally moves more free nonzeros than the local method. This last observation is easily explained by the fact that the global method can be seen as a generalization of the local method, as it considers all partitions instead of just two, which causes the local method to only explore a subset of the free nonzeros that the global method can explore and move.

Figure 13 also shows how much improvement is made on the imbalance for both methods, computed by taking the difference of the imbalance before and after the free nonzero method. Overall the global method performs best among the two methods, as may be expected. For values $P = 4$ and $P = 16$ the qualitative characteristics are similar to the graphs in Figure 13.

Next we consider the total communication volume results. While Algorithms 10 and 11 do not increase communication volumes, in Figure 12 we explained that the local and global methods could in theory decrease communication volumes, by moving nonzeros in a row or column such that a certain partition does not contain nonzeros in that row or column any more.

However, our main goal is to determine whether the enlarged search spaces lead to lower communication volumes. To establish a fair comparison, we will distinguish between *direct* communication gain, as caused by the algorithms themselves as in Figure 12, and *indirect* communication gain, as caused by the enlarged search spaces.

We measure the direct communication gain by computing the communication volume before and after executing the free nonzero algorithm, and taking the difference. Summing these differences over all recursive bipartitionings, we arrive at a total amount of direct communication gain for a particular run of a matrix. We average these figures per matrix and divide these averages by the mean total communication volume of Mondriaan without free nonzero algorithm for that matrix, which gives us the direct volume gain.

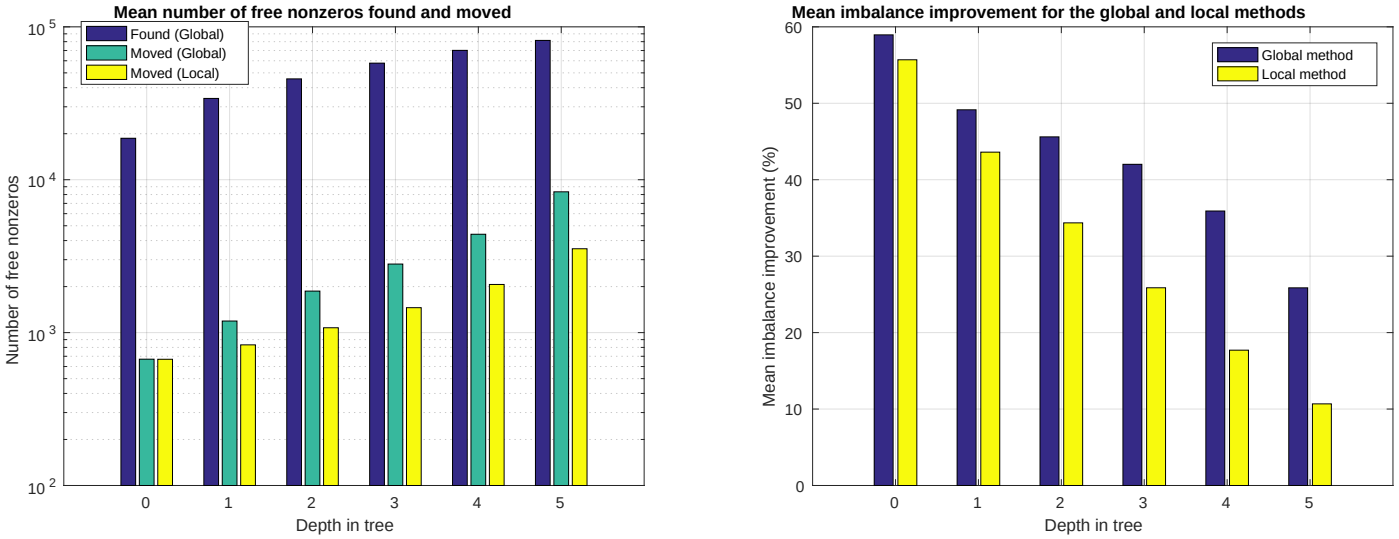


Figure 13: (Left) Average number of free nonzeros found and moved, using $P = 64$ and $\epsilon = 0.03$. Note that while for the global method we can keep track of both the number of found and moved nonzeros, for the local method it is only possible to keep track of the number of moved nonzeros. For the global method, the results come from the single run of the algorithm at the given depth, while for the local method the number of moved free nonzeros are summed over all iterations at the given depth, to obtain a fair comparison. (Right) Imbalance improvements achieved by the free nonzero algorithms, also for $P = 64$ and $\epsilon = 0.03$, averaged over all bipartitionings at a given depth of all matrices. In both panels, the number of runs is 10. Numbers of matrices averaged over can be found in Table 5.

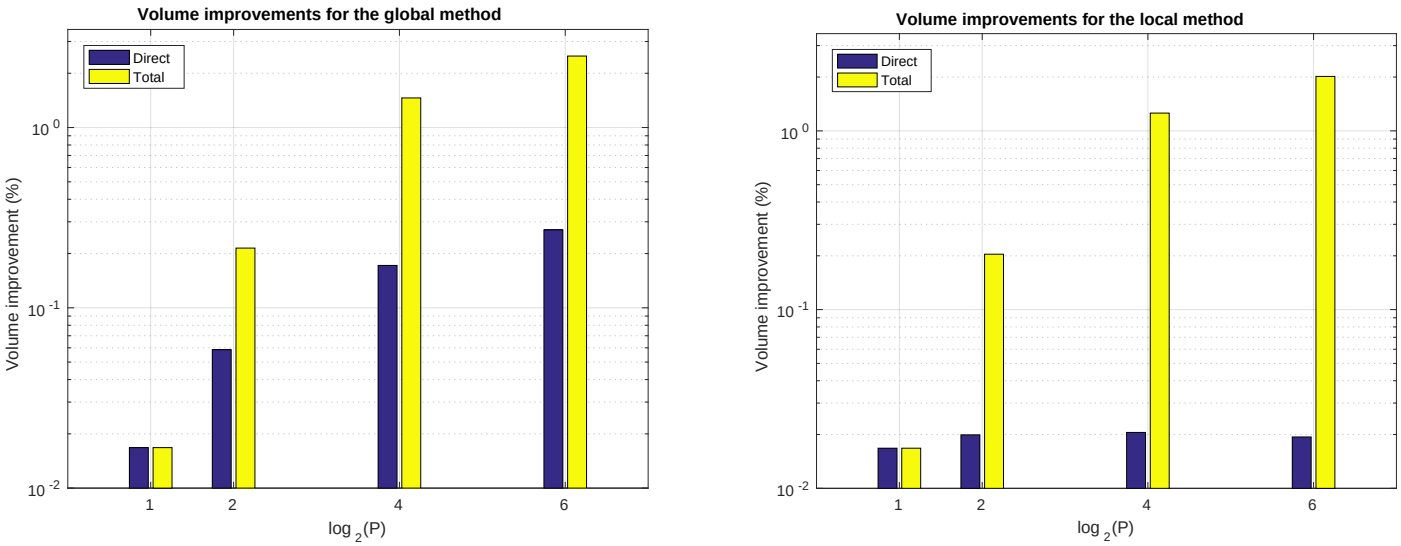


Figure 14: Average communication volume improvements, for the global method (left figure) and local method (right figure). For each value of P two bars are presented, the right of which is the total volume gain, and the left of which represents the direct gain. Means for the total gains are given in Table 5. The results are averaged over 10 runs, the numbers of matrices averaged over can be found in Table 5.

Additionally, we determine the total communication gain by performing runs of Mondriaan with a free nonzero algorithm and without a free nonzero algorithm, averaging the resulting communication volumes, giving us respectively V_{FNZ} and V_{clean}, after which V_{FNZ}/V_{clean} provides us with the total volume performance ratios, and one minus this ratio provides the gain. The indirect volume gain, caused by enlarged search spaces, can then be derived by taking the difference of the total communication gain with the direct communication gain. Caution must be taken with this however, as this identity is not entirely true, as it compares a value determined after the algorithm is finished (total gain) with a value (direct gain) that may influence communication volumes produced in deeper recursions.

The volume results are shown in Figure 14. It is clear that the direct gains are not negligible compared to the total gains, in particular for the global method. Moreover, for all considered values of P, we observe a total reduction in communication volume of at least 0.017%, and at most 2.02% and 2.49% for the local and global method, respectively. A remarkable result is that in a qualitative sense, both methods appear to have surprisingly similar total gains. For $P = 64$, the global method works up to 23% better relative to the local method, but this only leads to an difference of 0.47 percent point volume improvement. Given that the total volume gains are close to each other, another remarkable observation is that the direct volume improvements of the global method are significantly larger than for the local method. However, we currently do not have an explanation for this.

We now take a look at the timing results as presented in Table 5. We present timings of both machine 2 and 3, because we only have timings of $P = 2$ and $P = 16$ on machine 2. Although timings on machine 3 are less reliable, we include them to draw attention to the timing ratios for the global method, which in all cases prove to be higher than for the local method. This can be explained by the fact that the global method has a higher complexity in both space and time. Indeed, the maximum memory consumption of the runs with $P = 64$ with the global method was 2.350 gigabyte, while for the local method this was 1.896 gigabyte.

We can also see from Table 5 that the timings of machine 2 are in line with machine 3: the global method incurs more overhead than the local method. Another observation is that the local method introduces a reasonable trade-off: we have a mean reduction of communication volumes of up to 2%, at the expense of around 1% extra computation time. Additionally, as a bonus, we obtain results that have better load balance

	P	Matrices	Imbalance	Volume	Time (2)	Time (3)
Local	2	955	0.7415 ± 0.3483	0.9998 ± 0.0012	1.0028 ± 0.0231	1.0643 ± 0.1359
	4	955	0.7495 ± 0.3311	0.9980 ± 0.0279	-	1.0318 ± 0.1175
	16	954	0.8035 ± 0.2629	0.9874 ± 0.0267	1.0072 ± 0.0215	1.0441 ± 0.0580
	64	871	0.8960 ± 0.1891	0.9798 ± 0.0302	-	1.0510 ± 0.0438
Global	2	955	0.7414 ± 0.3483	0.9998 ± 0.0012	1.0088 ± 0.0222	1.0758 ± 0.1309
	4	955	0.7218 ± 0.3378	0.9979 ± 0.0296	-	1.0402 ± 0.0937
	16	954	0.7019 ± 0.3095	0.9854 ± 0.0293	1.0214 ± 0.0245	1.0675 ± 0.0736
	64	870	0.7448 ± 0.2987	0.9751 ± 0.0370	-	1.0871 ± 0.0513

Table 5: Mean imbalance, volume and timing performance ratios for the local and global free nonzero algorithms. The timing ratios are presented separately for machines 2 and 3. Whereas the imbalance improvements given in Figure 13 are taken from the individual free nonzero method calls, the numbers presented here are determined in the same way as the volume and timings: based on the end results only. Also the numbers of matrices over which the averages are computed are presented.

than Mondriaan produces itself, as Mondriaan without free nonzero algorithm tends to produce partitionings with achieved load imbalance ϵ' not much lower than ϵ.

Lastly, we take a look at the numbers of matrices in Table 5. All averages in this section are determined over matrices on which Mondriaan was able to determine a partitioning successfully in all 10 performed runs. Note that indeed, Mondriaan may not always succeed in generating a partitioning if it is unable to obey the imbalance criterion. The number of matrices in Table 5 is the number of matrices that Mondriaan never fails upon for both Mondriaan without free nonzero algorithm and Mondriaan with the concerning algorithm. Of interest are also the individual numbers: for $P = 64$, Mondriaan without free nonzero algorithm succeeds for 871 of the 955 matrices, with the local algorithm this number is 875 and for the global algorithm it increases to 887. For 14 of the 955 matrices (see Table 10), Problem 1.4 with $P = 64$ and $\epsilon = 0.03$ is infeasible according to Lemma 1.5. There is no particular reason for the other matrices for which Mondriaan fails to succeed, other than the fact that it happens whenever a recursive bipartitioning generates a partition that can not be feasibly partitioned. As this is immediately related to the imbalance constraint, it is not surprising that the free nonzero algorithms lead to more successful runs of Mondriaan.

6.4 Conclusion

In this section, we developed two algorithms for finding free nonzeros and using those to improve load imbalance, with the goal to improve communication volumes in the end due to larger available search spaces. We have seen that for high P, the local and global method both produce better communication volumes than Mondriaan without free nonzero algorithm. We have seen that this is caused by both direct gains, due to the algorithms improving the communication volume themselves, and by indirect gains, caused by the availability of larger search spaces.

The global method works only marginally better than the local method, at the cost of worse space and time complexity. The global method may have its applications as an imbalance improver, for example after having calculated a complete partitioning, but for high P it may be infeasible to use it.

The local method can be applied in every recursive bipartitioning without leading to too much extra computing time, leading to a volume gain of a few percent. As the communication volume improvement is larger than the time penalty, the advice would be to include the local method into a new version of Mondriaan. Along with an improved communication volume, we also get an improved work balance and less chance of failures for free.

7 Time improvements in PGA

In the multilevel method of Mondriaan, the hypergraph representing the input matrix is first coarsened to a coarse hypergraph, after which this coarse hypergraph is bipartitioned and then uncoarsened to its original size. In Mondriaan, the coarsening phase often accounts for most of the computation time, hence it is worth to investigate whether this phase can be improved.

As described in Section 1.3, the coarsening phase consists of multiple matching phases. In each matching phase, a matching of vertices is computed, where the aim is to match vertices that share many nets. In Mondriaan, the matching algorithm is divided into two parts, being the matching itself and the neighbour-finding function. The neighbour-finding function finds the heaviest unmatched neighbour of a given vertex, while the matching algorithm uses this neighbour-finding function to compute the matching.

When looking at solution quality, the best performing matching algorithm in Mondriaan is the Path Growing Algorithm (PGA) [11][10][2], and the best performing neighbour-finding function is the Inproduct function[2], that exactly computes the number of nets two vertices have in common. In this section, we will try to improve the PGA algorithm in two ways: one small modification in the code, and a new method to match multiple vertices at once instead of just two.

7.1 Improved marking of matched vertices

Algorithm 12&13: Current and new PGA algorithm used in Mondriaan. `PGA` generates a maximal matching M on a graph with vertex set V. Edge weights are computed on demand.

Algorithm 12: Current PGA implementation.

```
1: procedure PGA(Vertex set V)
2:     M ← {}
3:     Matched[v] ← False ∀v ∈ V
4:     for v ∈ V do
5:         if Matched[v] then
6:             continue
7:         P ← {v}
8:         Matched[v] ← True
9:
10:        while v has unmatched neighbours do
11:            w ← FindHeaviestUnmatchedNeighbour(v)
12:            P.extend(w)
13:            Matched[w] ← True
14:
15:            v ← w
16:        Matched[v] ← False ∀v ∈ P
17:        if |P| ≤ 1 then
18:
19:
20:            continue
21:        M' ← FindOptimalPathMatching(P)
22:        for (v, w) matched in M' do
23:            Matched[v] ← True
24:            Matched[w] ← True
25:            MoveVtxInNetAdjncy(v)
26:            MoveVtxInNetAdjncy(w)
27:        M ← M ∪ M'
28:
29:
30:
31:    Extend M to a maximal matching
32:    return M
```

Algorithm 13: New PGA implementation.

```
1: procedure PGA(Vertex set V)
2:     M ← {}
3:     Matched[v] ← False ∀v ∈ V
4:     for v ∈ V do
5:         if Matched[v] then
6:             continue
7:         P ← {v}
8:         Matched[v] ← True
9:         MoveVtxInNetAdjncy(v)
10:        while v has unmatched neighbours do
11:            w ← FindHeaviestUnmatchedNeighbour(v)
12:            P.extend(w)
13:            Matched[w] ← True
14:            MoveVtxInNetAdjncy(w)
15:            v ← w
16:        Matched[v] ← False ∀v ∈ P
17:        if |P| ≤ 1 then
18:            MoveVtxBackInNetAdjncy(v)
19:            continue
20:        M' ← FindOptimalPathMatching(P)
21:        for (v, w) matched in M' do
22:            Matched[v] ← True
23:            Matched[w] ← True
24:
25:
26:        M ← M ∪ M'
27:        for v ∈ P do
28:            if Matched[v] = False then
29:                MoveVtxBackInNetAdjncy(v)
30:    Extend M to a maximal matching
31:    return M
```

The Path Growing Algorithm (PGA) implementation in Mondriaan is given in Algorithm 12. Using the Inproduct neighbour-finding function, the graph (V, E) on which the algorithm acts can be derived from the

hypergraph (V, N). The vertex set of the graph equals the vertex set of the hypergraph, and we draw an edge between every pair of vertices, with an edge weight equal to the number of nets that the vertices share in the hypergraph. Note that this quantity can be seen as an inproduct between the vertices. Edges with weight 0 (i.e., the vertices have no nets in common) may be left out from the graph. While this derived graph is theoretically present, this graph is never explicitly calculated in the algorithm. The edge weights are computed on demand by the `FindHeaviestUnmatchedNeighbour` procedure, and as the vertex sets of the graph and the hypergraph are identical, the concept of the derived graph is completely absent in the code.

Refer back to Algorithm 12. In words, the algorithm iterates over all unmatched vertices. In each iteration, it grows a path of heaviest unmatched vertices from the found unmatched vertex, until an unmatched vertex is included that has no unmatched neighbours any more. A maximum matching on the path is computed, which is then applied to the overall matching. Drake and Hougardy proved [11][10] that this PGA algorithm[4], is a $\frac{1}{2}$-approximation algorithm to the graph matching problem.

When computing the heaviest unmatched neighbour of a vertex v, we compute the inner product of v with each of its unmatched neighbours v'. These inner products are calculated by the `FindHeaviest-UnmatchedNeighbour` procedure presented in Algorithm 14. We will inspect Algorithm 14 before comparing Algorithm 12 with Algorithm 13. Note that in the first set of tight loops in `FindHeaviestUnmatchedNeighbour`, it is not profitable to perform extra conditionals to check whether a vertex is unmatched: any speed improvement obtained by discarding matched vertices is undone by the overhead of the extra conditionals. However, as matched vertices are never eligible to be used in the path, as imposed by the second loop, skipping matched vertices in the first set of loops is desirable for performance. In other words, we want to progressively shrink the search space of vertices, to only include those eligible for matching.

A trick often used in Mondriaan is to temporarily delete (deactivate) a vertex from all nets it is in once it is matched. Applying this trick here, the first set of loops in `FindHeaviestUnmatchedNeighbour` will only consider unmatched vertices, as the nets n only contain unmatched vertices. The temporary removal of vertices is done by `MoveVtxInNetAdjncy`, which owes its name to the fact that vertices are not deleted but moved to the end of each net, after which decrementing the virtual net size by 1 effectively deactivates the vertex.

As is clear from Algorithm 12, the `MoveVtxInNetAdjncy` trick is already applied in the PGA algorithm. However, we can do better, as presented in Algorithm 13. In Algorithm 12, vertices are only deactivated once a path has been completely grown. Whenever the algorithm finds a long path, any vertices already used in the path are continuously reconsidered during the construction of the path, while it is already known that they cannot be used in the path again. Hence in Algorithm 13, we deactivate a vertex immediately after it has been added to the path. This way, already used vertices are never reconsidered.

The drawback of Algorithm 13 is that we have to re-activate vertices with `MoveVtxBackInNetAdjncy` if the path is not long enough or if vertices are not used in the optimal path matching M'. However, the number of vertices that need to be re-activated is assumed to be only small. By this assumption, also the number of calls to `MoveVtxInNetAdjncy` should not increase too much: the number of times it is called in Algorithm 13 should be equal to the number of times it is called in Algorithm 12 plus the number of times `MoveVtxBackInNetAdjncy` is called.

7.2 Agglomerative coarsening

In the coarsening phase, Mondriaan may take a long time to iteratively coarsen the hypergraph, halving the hypergraph size in each iteration. If we can increase this reduction factor, maybe we can manage to obtain a partitioning of hopefully equal quality using less coarsening iterations, which may in turn improve running times.

In the PGA algorithm, the only place where we use the fact that a matching exists of two vertices is the `FindOptimalPathMatching` algorithm. Indeed in Algorithm 13, generating the path is independent of the number of vertices we want to match, as the actual matches are generated in the `FindOptimalPathMatching`

[4]In their terminology the PGA' algorithm.

Algorithm 14 FindHeaviestUnmatchedNeighbour()

Stripped down version of the neighbour finding function in Mondriaan. In Mondriaan, many more features are available such as inner product scaling, net scaling and use of free vertices, but these details are not of interest here.
Input: Vertex $v \in V$, matched array $Matched$ from the PGA algorithm.
Output: Unmatched vertex $w_{\max} \neq v$ having maximum inner product with v
procedure FindHeaviestUnmatchedNeighbour
    $P_{v'} \leftarrow 0 \ \forall v' \in V$ ▷ Inner products
    $W \leftarrow \{\}$ ▷ Neighbours
    for each net n that v is in **do** ▷ Determine all inner products
        for each vertex v' in n **do**
            if $P_{v'} = 0$ **then**
                $W \leftarrow W \cup \{v'\}$
            $P_{v'} \leftarrow P_{v'} + 1$

    $max \leftarrow -1$, $w_{\max} \leftarrow null$ ▷ Determine maximum inner product
    for each vertex v' in W **do**
        if $v' \neq v \wedge Matched[v'] = False \wedge P_{v'} > max$ **then**
            $max \leftarrow P_{v'}$
            $w_{\max} \leftarrow v'$
    return $w_{\max}$

algorithm. In other words, the amount of work needed to find the path is independent of the maximum number of vertices in a match n_v. As this work needs to be done irrespective of n_v, we may gain significant speed improvements by generating matches with $n_v > 2$. As the current `FindOptimalPathMatching` algorithm only works for $n_v = 2$, we develop a new algorithm that works for $n_v \geq 2$.

We define a *multiple matching* to be a matching with $n_v > 2$, and name the coarsening procedure using multiple matchings *agglomerative coarsening*, after the terminology used in [35]. We develop a dynamic programming solution to the problem of finding a maximum multiple matching on some weighted path $P = (v_0, v_1, \ldots, v_{|P|-1})$. The weight of the edge between vertices v_t and v_{t+1} is given by w_t, for all $t \in [0, T-1]$, where $T = |P| - 1$. The algorithm we consider will be focused around the edges instead of the vertices, hence we will say an edge is matched whenever both its vertices are matched into the same set of matched vertices. Furthermore, we will be working with a maximum number of edges in a match $n_e = n_v - 1$.

We define the variable $W_{t,k}$ ($t \in [0, T-1]$, $k \in [0, n_e]$) to be the maximum weight of a multiple matching using edges in the subpath $[0, t]$ only, where the last sequence of matched edges (including edge t) is of length at most k. If edge t is not matched in subpath $[0, t]$, we say the last sequence has length zero.

For $t, k > 0$, we can make the following two choices for $W_{t,k}$:

1. We may include the last edge t in the multiple matching. Additionally, we may include at most $k-1$ of the edges before edge t. Note that the maximum weight of a multiple matching with at most $k-1$ matched edges in the last sequence ending with edge $t-1$ is given by $W_{t-1,k-1}$, hence the maximum possible weight when we include edge t is $W_{t-1,k-1} + w_t$.

2. We may also not include the last edge t. In this case, we may use at most n_e edges before edge t, as there are no limitations on the number of matched edges in the second-to-last sequence. Hence the maximum possible weight for this choice is W_{t-1,n_e}.

Note that the second option above also holds for $k = 0$, so we may define $W_{t,0} \equiv W_{t-1,n_e}$ for $t > 0$.

Clearly, $W_{0,0} = 0$ as selecting no edges yields no weight, and $W_{0,k} = w_0$ for $k \in [1, n_e]$ as $k \geq 1$ implies we may select the first edge.

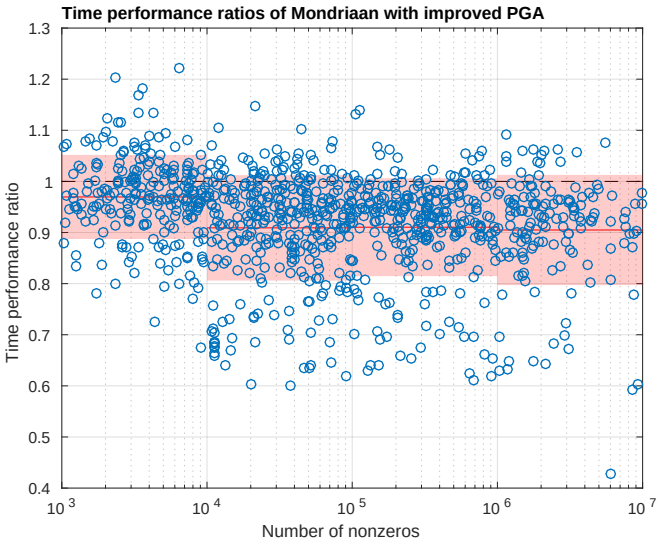


Figure 15: Time performance ratios of the improved `PGA` algorithm implementation in Mondriaan, using $P = 2$ and $\epsilon = 0.03$.

$\log_{10}(nnz)$	Ratio
$3-4$	0.9696 ± 0.0823
$4-5$	0.9091 ± 0.1031
$5-6$	0.9104 ± 0.0959
$6-7$	0.9049 ± 0.1079
$3-7$	0.9231 ± 0.1006

Table 6: Time performance ratio means belonging to Figure 15, taken over 5 runs.

Summarizing, we can compute $W_{t,k}$ as follows:

$$W_{t,k} = \begin{cases} 0 & \text{if } t = k = 0, \\ w_0 & \text{if } t = 0 \text{ and } k > 0, \\ W_{t-1,n_e} & \text{if } t > 0 \text{ and } k = 0, \\ W_{t-1,k-1} + w_t & \text{if } t, k > 0 \text{ and } W_{t-1,k-1} + w_t > W_{t-1,n_e}, \\ W_{t-1,n_e} & \text{if } t, k > 0 \text{ and } W_{t-1,k-1} + w_t \leq W_{t-1,n_e}, \end{cases} \qquad \forall t \in [0, T-1], k \in [0, n_e].$$

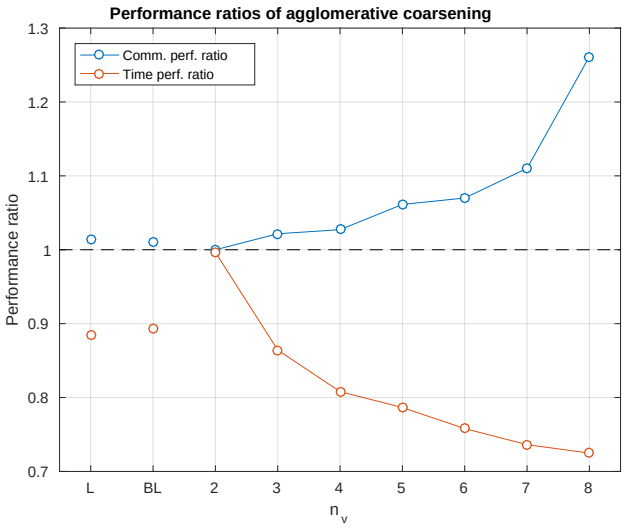
Clearly, computing all values of $W_{t,k}$ takes $O(n_e T)$ time. In an implementation, we iterate through all values $t \in [0, T-1]$, in each iteration computing all values $W_{t,k}$, $k \in [0, n_e]$. We will only keep track of all weights in the current (t) and previous $(t-1)$ iteration, however we will still have to keep track of our branching choices in an array of size $O(n_e T)$ in order to be able to backtrack the solution belonging to the maximum weight.

7.3 Results

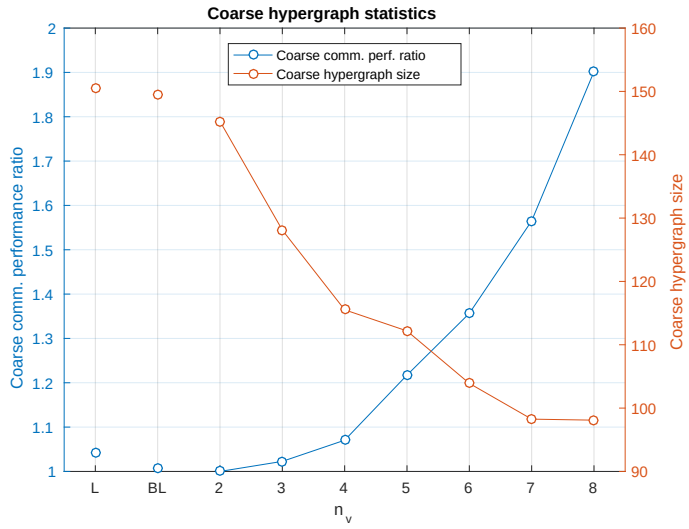
We perform tests with the standard test set. Results for Section 7.3.1 are computed on machine 2, while results for Section 7.3.2 are computed on machine 4. For Section 7.3.2, the control computation (with 'clean' Mondriaan) is taken to be Mondriaan with the improved `PGA` algorithm of Section 7.3.1. In other words, Section 7.3.2 builds forward on Section 7.3.1.

7.3.1 Improved marking of matched vertices

In Figure 15 and Table 6, the results of the improved vertex marking of Section 7.1 can be found. For small matrices the performance gain is smaller than for larger matrices: matrices with $nnz \geq 10^4$ have a mean performance gain of 9%, while the overall gain is 7.7%. The small gain for small matrices does not come unexpected, as for small matrices Mondriaan spends relatively less time in the coarsening phase. Additional tests were performed with another version of Algorithm 13, in which the last loop in lines 27-29 does not



(a) Communication and time performance ratios when applying the new multiple matching algorithm versus the original matching algorithm.



(b) Communication performance ratio of the coarsest hypergraph, and the mean sizes of these coarsest hypergraphs. The maximum number of vertices in the coarsest hypergraph was set to 200*.*

Figure 16: Results of the agglomerative coarsening method, using $P = 2$ and $\epsilon = 0.03$, averaged over 5 runs. On the horizontal axis the maximum number of vertices in a match n_v is plotted, L indicates the log *rule for n_v, BL indicates the bounded* log *rule for n_v. Notice that for $n_v = 2$, the new method is compared to the old method, hence the end result Mondriaan produces is exactly the same. The numerical data is presented in Table 7.*

iterate over all $v \in P$ but only over all v that are not used in the path matching. However, additional logic was required in `FindOptimalPathMatching` to provide a list of unused vertices, which caused the alternative method to be slightly slower than the method reported here.

7.3.2 Agglomerative coarsening

In Figure 16 and Table 7 the results of the agglomerative coarsening method of Section 7.2 are presented. We can see from Figure 16a that, as expected, computation times decrease with n_v, while communication volumes increase with n_v. Observe that computation times are significantly lower already for $n_v = 3$, while the communication volume only increases slightly. While the same also holds for $n_v = 4$, for $n_v \geq 5$ the communication volume starts to increase more, while the time gain starts to stagnate.

There are two explanations for the fact that the communication volume increases with n_v. Firstly, generating matches with $n_v > 2$ can be considered as a rough replacement of multiple runs of matchings with $n_v = 2$. Additionally, as we mentioned before, with $n_v = 2$ the algorithm used is a $\frac{1}{2}$-approximation, while for $n_v > 2$ we lose this property. Secondly, when matching more vertices at once, less coarsenings are required to arrive at a sufficiently small coarse matrix. This also implies that less uncoarsenings are required, and hence less iterative refinements can be performed. As iterative refinements are important for fine tuning the partitioning, this last observation may contribute to the higher communication volumes.

In Figure 16b we see that the communication volume performance ratio of the coarse hypergraph (that is generated by the initial partitioning) increases faster than the total communication volume does in Figure 16a. From this we can conclude that the refinement in the uncoarsening phase seems to undo a significant part of the increased communication volume in the coarse hypergraph. Hence the lack of opportunity to refine the partitioning in the uncoarsening phase will not be the primary cause of increasing total communication

n_v	TPR	CPR	CCPR	Coarse size	V_{total}/V_{coarse}
L	0.8850 ± 0.1461	1.0145 ± 0.2581	1.0433 ± 0.5927	150.56 ± 86.66	0.6230 ± 0.2462
BL	0.8925 ± 0.1222	1.0104 ± 0.2820	1.0075 ± 0.2987	149.52 ± 72.30	0.6271 ± 0.2409
2	0.9965 ± 0.0469	1.0000 ± 0.0000	1.0000 ± 0.0000	145.26 ± 31.93	0.6250 ± 0.2399
3	0.8643 ± 0.1155	1.0217 ± 0.6639	1.0227 ± 0.4438	128.11 ± 81.15	0.6226 ± 0.2416
4	0.8080 ± 0.1304	1.0272 ± 0.4058	1.0704 ± 0.3436	115.52 ± 82.43	0.6074 ± 0.2421
5	0.7865 ± 0.1546	1.0613 ± 0.5186	1.2173 ± 1.1156	112.18 ± 94.91	0.5807 ± 0.2486
6	0.7580 ± 0.1647	1.0700 ± 0.3080	1.3565 ± 0.9776	103.90 ± 100.00	0.5617 ± 0.2597
7	0.7362 ± 0.1696	1.1102 ± 0.4197	1.5651 ± 1.6562	98.26 ± 99.24	0.5311 ± 0.2666
8	0.7248 ± 0.1848	1.2615 ± 2.8860	1.9017 ± 3.4532	98.10 ± 118.71	0.5162 ± 0.2745

Table 7: Results belonging to Figure 16: mean Time Performance Ratio (TPR), mean Communication Performance Ratio (CPR), mean Coarse Communication Performance Ratio (CCPR), mean size of the coarsest hypergraph and mean volume ratio between final partitioning and coarse partitioning. Computations are done with $P = 2$ and $\epsilon = 0.03$, and averaged over 5 runs. The TPR, CPR and CCPR are ratios computed between the runs with the respective n_v settings and the runs with a clean Mondriaan installation (both including the improvements of Section 7.1). The last two columns are computed directly from the runs with the respective n_v settings without normalization from a control run.

volumes with increasing n_v. This leads to the conclusion that the increase in total communication volume is primarily caused by the coarsening phase producing a coarse hypergraph of less quality for higher n_v.

Next, notice that Mondriaan attempts to coarsen the hypergraph until the number of vertices drops below 200. Whenever the number of vertices is just above 200, coarsening with e.g. $n_v = 8$ may reduce the hypergraph size to well below 100, or even to 26 vertices in ideal circumstances. Indeed, in Figure 16b we see that the mean number of vertices in the coarse hypergraph decreases significantly with increasing n_v. This may be a cause for the reduced quality of the coarse hypergraph, as a too coarse hypergraph is likely to not sufficiently represent the structure of the original hypergraph any more. To circumvent this problem, we introduce a log rule for the number of vertices, to make n_v dependent on the number of vertices $|V|$ the hypergraph currently exists of:

$$n_v = \max\left\{2, \left\lfloor \log\left\lfloor \frac{|V|}{max.nr.vertices} \right\rfloor \right\rfloor\right\}.$$

Notice that the maximum number of vertices in above formula is 200 by default; with this setting the maximum number of vertices in a match equals 2 for $|V| < 1600$. This way, the latest coarsenings will not be too coarse. For $1600 \leq |V| < 3200$, the maximum number of matched vertices n_v equals 3, for $3200 \leq |V| < 6400$, $n_v = 4$, et cetera. Note that n_v becomes large for large hypergraphs, i.e. it exceeds 8 whenever $|V| \geq 102400$.

Results of this log rule are presented in the figures with $n_v = L$. Note that indeed, the mean size of the coarse hypergraph is not smaller than that of $n_v = 2$ (to the contrary, it is slightly larger). In all other regards the results are very similar to the case $n_v = 3$, compared to which the log rule leads to slightly less time improvement and a slightly less increased communication volume.

Note that while the log rule produces reasonable results, they do not significantly differ from $n_v = 3$. As mentioned, in the log rule the value of n_v may become large for large matrices, which may possibly lead to poor results in the first few coarsenings. As for $n_v = 3$ and $n_v = 4$, Figure 16a showed good results, the last alternative we tested is a bounded log rule, where n_v is defined by

$$n_v = \min\left\{4, \max\left\{2, \left\lfloor \log\left\lfloor \frac{|V|}{max.nr.vertices} \right\rfloor \right\rfloor\right\}\right\}.$$

In words, the value of n_v is the same as in the log rule, except it may not exceed 4. Results of this rule are presented in the figures with $n_v = BL$. We see the same difference between $n_v = BL$ and $n_v = L$ as we saw between $n_v = L$ and $n_v = 3$: compared to the normal log rule, the bounded log rule leads to slightly less time improvement and a slightly less increased communication volume. Note however, that while the coarse communication volume was higher for $n_v = L$ than for $n_v = 3$, the coarse communication volume for $n_v = BL$ is close to that of $n_v = 2$.

7.4 Conclusion

We have discussed two ways to improve the PGA algorithm used in Mondriaan. The first was a modification in the marking of matched vertices, which clearly proved itself to be worthwhile, with an overall speed improvement of 7.7%. The second modification was to generate generalized matches with more than two vertices per match, to obtain a coarse hypergraph in fewer coarsening iterations. As expected, this modification decreases running times at the cost of increased communication volumes. When n_v is taken equal to a fixed number $n_v \geq 3$, communication volumes increase with 2% to 26%, traded off with speed improvements of 13% to 27%. When using the log or bounded log rule, communication volumes increase with respectively 1.5% and 1.0%, while speed improvements of approximately 11% are achieved.

As the first modification improves the mean computation times, we advise that this modification should be included in a next release of Mondriaan. Agglomerative coarsening may be included into Mondriaan to facilitate a speed improvement at the expense of a slight increase in communication volume. In particular, the bounded log rule provides a fair trade-off between speed improvement and solution quality. However, as the aim of Mondriaan is to find solutions of the best quality, the advice would be to not activate agglomerative coarsening by default.

$\log_{10}(nnz)$	Imbalance	Volume (Mean)	Volume (Geomean)	Time
$3-4$	0.7676 ± 0.6472	1.0044 ± 0.1392	1.0154	0.6648 ± 0.1859
$4-5$	0.7126 ± 0.4211	1.0006 ± 0.1222	0.9958	0.7012 ± 0.1440
$5-6$	0.7862 ± 0.4257	1.0003 ± 0.0963	0.9894	0.7865 ± 0.1281
$6-7$	0.7576 ± 0.4280	1.0126 ± 0.1040	1.0081	0.8299 ± 0.1284
$3-7$	0.7528 ± 0.4864	1.0032 ± 0.1173	1.0005	0.7357 ± 0.1605

Table 8: Mean imbalance, volume and time performance ratios for the Mondriaan 4.2 candidate versus Mondriaan 4.1, for $P = 2$ and $\epsilon = 0.03$. The time averages are computed over 5 runs, the imbalance and volume averages are computed over 10 runs. The mean imbalance improved from 0.0194 to 0.0137.

8 Conclusion

In this thesis, we have considered five improvements on Mondriaan version 4.1. The improvements to the sorting algorithm, gain bucket structure and PGA algorithm as well as the new zero volume search algorithm were aimed at improving the run time of Mondriaan, while the new free nonzero algorithm was aimed at improving the quality of partitionings.

The performance of the sorting algorithm was enhanced significantly, but as part of the Mondriaan algorithm speed improvements were only small. The changes to both the gain bucket structure and the PGA algorithm improved the run time significantly. The zero volume search finds partitionings with zero volume much faster than Mondriaan 4.1 does, but when also taking into account matrices for which no zero volume partitioning is possible, the additional code executed leads to an average run time close to that of Mondriaan 4.1. For high P, the local free nonzero algorithm proved to be able to improve both imbalance and total communication volume, while also decreasing the number of failed partitionings.

Agglomerative coarsening did prove do reduce run times, but it also increased communication volumes. This leads us to the conclusion that agglomerative coarsening is a worthwhile option to include into Mondriaan, but not as a default. The global free nonzero method proved to reduce volumes slightly better than the local method, but at the cost of a worse run time both in complexity and in practice, hence the global method is not considered a good default option.

While writing the thesis, also some other improvements were shortly considered:

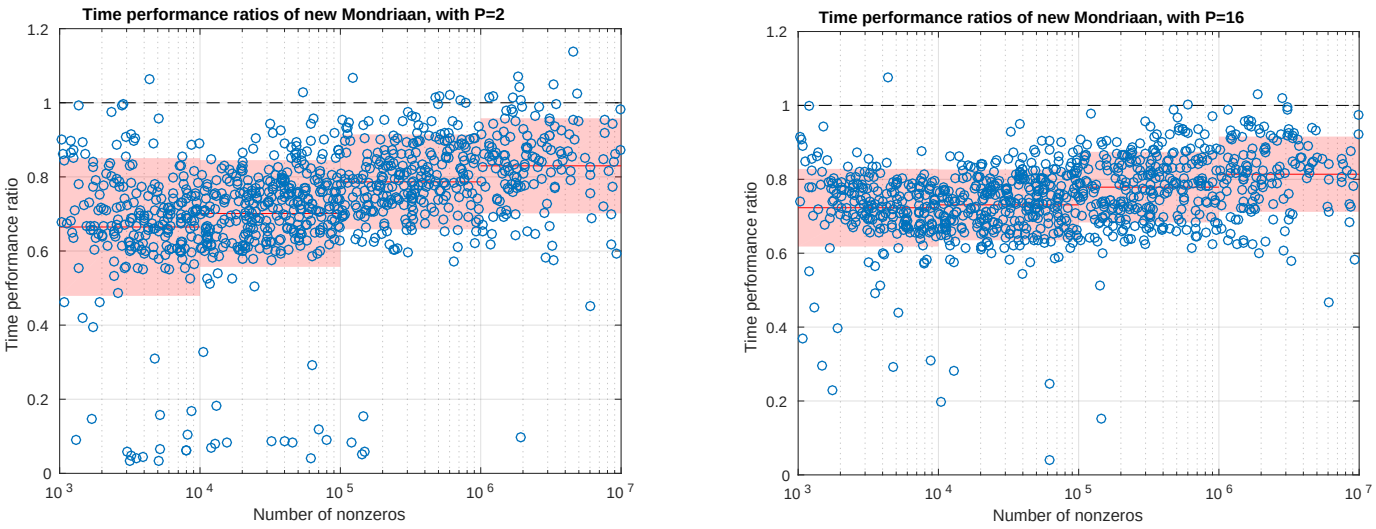
- The `MoveVtxInNetAdjncy` trick from Section 7.1 was also applied to the `UpdateGains` method in the `HKLFM` method. However, while in Section 7.1 we primarily moved logic, in the `UpdateGains` method additional logic was needed, which proved to outweigh any speed improvements gained. A single run showed an increase in run time of 5%, hence this improvement was discarded.

- An attempt was made to implement the Net Split method proposed in [30] as an alternative initial partitioning method. A few tests showed that in general this method did not perform better than the default method in Mondriaan, hence also this method was discarded.

8.1 Putting everything together

Putting all proposed improvements together, we obtain a candidate version for Mondriaan 4.2. Using our standard test set again, we perform a few runs on machine 2, of which the results can be found in Figure 17 and Tables 8 and 9. For $P = 2$, we see that overall run time improvements range from 17% for large matrices to 33% for small matrices, with an overall mean of 26%. Also the imbalance improves greatly with 25% on average.

Note that for $P = 2$, none of the modifications in the previous sections improved the communication volumes significantly, hence we expect the mean ratios to be close to 1. In Table 8, we see an effect we have not yet had to deal with: taking the standard mean over ratios tends to return a result that may be higher than expected, as one doubling and one halving lead to a mean of $\frac{2+0.5}{2} = 1.25 > 1$. The reason that we have not had to deal with this before, is that the random processes used between versions were mostly



(a) $P = 2$ *(b)* $P = 16$

Figure 17: Time performance ratios of all proposed improvements together, for $\epsilon = 0.03$ *and* P *as given, averaged over 5 runs. Averages are given in Tables 8 and 9.*

identical in sections involving communication volume results. In contrast, combining all modifications leads to different random processes (or, RNG) being performed, hence outliers start to appear here. Indeed, note that the mean presented here is given by

$$\frac{1}{955}\sum_{i=1}^{955}\frac{V_i^{new}}{V_i^{old}} = 1.0032 \pm 0.1173,$$

where V_i^{old} and V_i^{new} are the average volumes of matrix i in Mondriaan 4.1 and 4.2, respectively, averaged over 10 runs. On the other hand, we can also compute

$$\frac{1}{955}\sum_{i=1}^{955}\frac{V_i^{old}}{V_i^{new}} = 1.0277 \pm 0.7261,$$

indicating that both going from 4.1 to 4.2 and going back from 4.2 to 4.1 would lead to a volume increase, which is obviously not the case.

Moreover, as in this case we expect the volume ratios to be around 1, we can interpret the given standard deviations as measures for uncertainties. Given that these are quite large, and the fact that we have no reason to expect that volumes have increased due to our proposed changes, we conclude that the volumes being higher than 1 is caused by outliers. To compare the results with another mean, also the geometric mean of all nonzero ratios[5] are given in Table 8.

The results for $P = 16$ are given in Figure 17b and Table 9. Here, run times show an improvement of almost 19% for large matrices, up to nearly 28% for small matrices, with an overall mean of just under 25%. The imbalance improvement is less than for $P = 2$, but a mean improvement of 20% is still present. The communication volume also shows a slight reduction.

[5]Note that whenever one number is 0, the geometric mean over the set also becomes 0 regardless of the other numbers present in the set.

$\log_{10}(nnz)$	Imbalance	Volume (Mean)	Volume (Geomean)	Time
$3-4$	0.7861 ± 0.2841	0.9620 ± 0.0877	0.9643	0.7231 ± 0.1051
$4-5$	0.7946 ± 0.2784	0.9844 ± 0.0763	0.9858	0.7308 ± 0.0959
$5-6$	0.8144 ± 0.2633	0.9934 ± 0.0570	0.9910	0.7788 ± 0.0956
$6-7$	0.7987 ± 0.2886	1.0002 ± 0.0357	0.9996	0.8138 ± 0.1019
$3-7$	0.7987 ± 0.2771	0.9840 ± 0.0709	0.9842	0.7549 ± 0.1043

Table 9: Mean imbalance, volume and time performance ratios for the Mondriaan 4.2 candidate versus Mondriaan 4.1, for $P = 16$ and $\epsilon = 0.03$. All averages are computed over 5 runs. The mean imbalance improved from 0.0294 to 0.0236.

In conclusion, we can say that the proposed candidate version for Mondriaan 4.2 introduces a significant run time improvement, along with a significant reduction in load balance and a slight reduction in total communication volume.

8.2 Suggested future work

A lot of study has already been done to improve Mondriaan, but there are still opportunities for further improvement.

Initial partitioning In [30], alternative initial partitioning methods were proposed. As mentioned before, while writing this thesis an attempt was made to implement the Net Split method mentioned there, however it did not produce better results than the current initial partitioning method. While I am quite confident the implementation matches the description included in [30], it might be possible that some part of the method was misunderstood. As this thesis was mainly focused on improving run times, the Net Split method was discarded, but further research to this method may be taken up again. Even if the Net Split does not improve communication volumes of matrix partitionings in general, it may do so for specific kinds of matrices, as was found in [30].

Also applying the Karmarkar-Karp method, as mentioned in [30], might provide an improvement to Mondriaan, as we will motivate shortly. Another initial partitioning that may improve solution quality would be applying MondriaanOpt as an initial partitioner. Some conversion work would have to be done, to convert the hypergraph into a structure MondriaanOpt can work with. Once this is done an interesting question would be whether using an optimal partitioning as initial partitioning will provide improved total communication volumes, and what the run time penalty would be of doing this.

Too large imbalance For some matrices, Mondriaan tends to generate a partitioning that does not obey the imbalance criterion, as shown in Appendix A.4. We expect that Mondriaan 4.2 will have better behaviour due to the free nonzero algorithm, but we have not tested Mondriaan 4.2 with the values of P mentioned there. If the high achieved imbalances are caused by the initial partitionings, this behaviour can maybe be solved by using Karmarkar-Karp for the initial partitioning as mentioned above. Otherwise, the cause must be sought somewhere else, for example in the way Mondriaan determines the weights for the partitions.

Too large volume Corollary A.3 provides us with a general upper bound on the optimal communication volume. While in most cases Mondriaan produces far better volumes, for low P it sometimes does not and returns a volume higher than the upper bound, see Appendix A.3. Hence, it might be a good idea to check whether generated partitionings do not exceed the upper bound, and if they do, generate a partitioning using another algorithm, perhaps the one provided in Corollary A.3.

Beyond free nonzeros In Section 6, free nonzeros were used to improve the load balance of bipartitionings. It was shown that improving the load balance may also improve communication volumes, hence further

research on this subject may be fruitful. We already suggested in Section 6 that while free nonzeros provide one way to improve the load balance, one may also come up with other methods to do this. For example the gain bucket structure available in Mondriaan may be used, computing the gains of all vertices in the largest partition, and moving vertices with nonnegative gain to the other partition as long as it improves the load balance. This would yield a method working on the hypergraph rather than on the resulting matrix, possibly leading to different results, which together with the free nonzero algorithm may reduce the load imbalance even further.

A Appendix

A.1 Infeasible problems for $P = 64$ and $\epsilon = 0.03$

For $P \in \{2, 3, 4, 16\}$, there are no matrices from the standard test set for which Problem 1.4 with $\epsilon = 0.03$ is infeasible. For $P = 64$, there are. They can be found in Table 10.

ID	Group	Name
2	HB	494_bus
6	HB	arc130
16	HB	bcspwr04
66	HB	bcsstm11
76	HB	bcsstm26
700	LPnetlib	lp_vtp_base
884	Pothen	sphere3
1112	Sandia	oscil_dcop_01
1198	Rajat	rajat14
1476	Pajek	GD00_c
1488	Pajek	GD96_a
1676	Meszaros	gams30am
2014	JGD_Homology	ch5-5-b2
2110	JGD_Homology	n4c5-b2

Table 10: All 14 *matrices from the standard test set of* 955 *matrices that are infeasible for* $P = 64$ *and* $\epsilon = 0.03$.

A.2 Upper bound on communication volume

In the MondriaanOpt package, a general upper bound on the communication volume was used, but not proven. Here we provide a proof.

Adopt the imbalance constraint as used in MondriaanOpt,

$$W(A_0, \ldots, A_{P-1}) \leq (1 + \epsilon) \left\lceil \frac{nnz(A)}{P} \right\rceil, \tag{13}$$

and note the differences with Equations (7) and (9).

Lemma A.1. *Using the imbalance constraint in Equation* (13) *and* $\epsilon = 0$*, an* $m \times n$ *matrix* A *can be partitioned into* P *partitions with volume at most* $(\min(m, n) + 1)(P - 1)$.

Proof. Assume $m \leq n$, otherwise transpose the matrix. Now $\min(m, n) + 1 = m + 1$, hence we seek a partitioning with volume at most $(m + 1)(P - 1)$. Denote the number of nonzeros in column j with nz_j. Consider the algorithm presented in Algorithm 15.

In the for-loop, we assign entire columns to partitions, hence at the end we have a partitioning that uses at most $m(P - 1)$ communication volume. However, the imbalance constraint may be violated. If it is not, we are done, having created a feasible partitioning with volume at most $m(P - 1)$. If the constraint is violated, we continue with the next step in the algorithm.

The target weight w_t is computed, this is the weight that all partitions at most may have to obtain zero imbalance. Visualize the first for-loop as P vertical bars, stacking weights on top of each other as defined in the loop (see Figure 18). Every next weight is then put on top of the bar $p \in [0, P - 1]$ that is smallest (i.e., has least weight). Note that no column weights will be added to a partition p any more once

Algorithm 15 Algorithm for constructing a feasible partitioning

$J_p = \{\}\ \forall p \in [0, P-1]$ ▷ J_p = column indices j assigned to p
$w_p = 0\ \forall p \in [0, P-1]$ ▷ w_p = total weight of all columns in J_p
for $j \in [0, n-1]$ **do**
    $p \leftarrow \arg\min_{q \in [0,P-1]} w_q$
    $J_p \leftarrow J_p \cup \{j\}$ ▷ Assign column j to partition with currently least weight
    $w_p \leftarrow w_p + nz_j$

$w_t \leftarrow \left\lceil \frac{nnz(A)}{P} \right\rceil$ ▷ Target weight

while true **do**
    $p_{\min} \leftarrow \arg\min_{q \in [0,P-1]} w_q$
    $w_{\min} \leftarrow w_{p_{\min}}$
    $p_{\max} \leftarrow \arg\max_{q \in [0,P-1]} w_q$
    $w_{\max} \leftarrow w_{p_{\max}}$
    if $w_{\max} \leq w_t$ **then**
        break
    $\Delta \leftarrow \min\{w_{\max} - w_t, w_t - w_{\min}\}$
    Remove Δ weight from $p_{\max}$ and add it to $p_{\min}$ (and update $w_{\min}$, $w_{\max}$)

$w_p \geq w_t$. Indeed, if some column with $nz_j > 0$ would be assigned to a partition with $w_p \geq w_t$, this means $\min_{q \in [0,P-1]} w_q = w_p \geq w_t$, hence the total assigned weight after adding nz_j will become

$$nz_j + \sum_{q=0}^{P-1} w_q \geq nz_j + P \cdot w_t = nz_j + P \cdot \left\lceil \frac{nnz(A)}{P} \right\rceil > nnz(A).$$

This means we would have assigned more weight than there are nonzeros, which is obviously impossible. Hence we can conclude that no weight will be assigned to partition p once $w_p \geq w_t$. This implies than in our stack of column weights, only the top one (i.e. the one appended last) can exceed w_t. Al weights added before the last weight will be entirely below w_t.

Now in the while-loop, we determine the partition with maximum and minimum weight. Then we do either of the following:

- We remove $w_{\max} - w_t$ weight from the partition with maximum weight and add it to the partition with minimum weight. This way, $p_{\max}$ now has weight equal to w_t, hence it can be marked as finished.
- We remove $w_t - w_{\min}$ weight from the partition with maximum weight and add it to the partition with minimum weight. This way, $p_{\min}$ now has weight equal to w_t, hence it can be marked as finished.

When 'removing weight', we split the top column weight in the maximum partition (i.e., the weight appended last), and move part of this column to the other partition. This way, we add one communication to improve the load balance. By construction, the weight of $p_{\max}$ never drops below w_t and the weight of $p_{\min}$ never rises above w_t. Note that by our previous argumentation, only the last column weight appended to $p_{\max}$ exceeded w_t, hence only this lastly appended column weight is cut by the algorithm.

In every iteration, at least one partition can be marked as finished (and possibly two, if $w_{\max} - w_t = w_t - w_{\min}$), and one communication is added. As soon as all partitions are finished ($w_p \leq w_t$ for all p), the algorithm terminates. This is the case after at most $P-1$ iterations, as whenever $P-1$ partitions have weight equal to w_t, also the last partition has weight at most w_t. Hence in total, at most $P-1$ communications are added to the $m(P-1)$ communications we had before, leading to a total of $(m+1)(P-1)$ communication volume. □

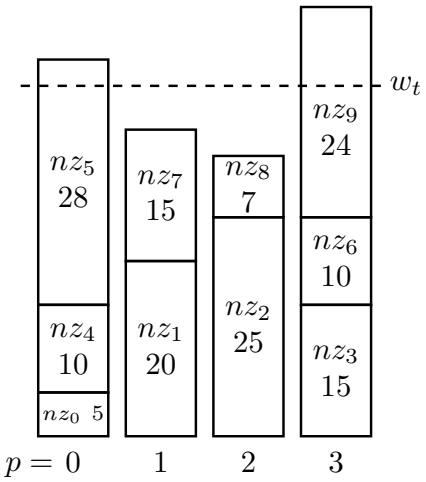


Figure 18: Illustration of the state of Algorithm 15 at line 8, in a situation with $P = 4$. Here $w_0 = 43$, $w_1 = 35$, $w_2 = 32$, $w_3 = 49$ and $w_t = 40$. The variables nz_j are given, along with their values. In the first iteration of the while-loop, w_3 will lose 8 weight to w_2, leaving w_3 at 41 and setting w_2 to 40. Then w_0 loses 3 weight to w_1, making w_0 equal to 40 and leaving w_1 at 38. Lastly, w_3 loses 1 weight to w_1, leaving w_3 at 40 and increasing w_1 to 39. Now $w_p \leq w_t$ for all $p \in [0, P-1]$.

Corollary A.2 (Default upper bound used in MondriaanOpt)**.** *Using the imbalance constraint in Equation (13), an $m \times n$ matrix A can be bipartitioned with imbalance at most ϵ and volume at most $\min(m, n) + 1$.*

Corollary A.3. *Suppose for a given $m \times n$ matrix A, number of processors P and imbalance parameter $\epsilon \geq 0$, Problem 1.4 is feasible. Then the optimal communication volume is at most $(\min(m, n) + 1)(P - 1)$.*

Proof. Lemma A.1 constitutes a solution feasible to modified imbalance constraint $W(A_0, \ldots, A_{P-1}) \leq \left\lceil \frac{nnz(A)}{P} \right\rceil$ in Equation (13), with volume at most $(\min(m, n) + 1)(P - 1)$. As Problem 1.4 is assumed to be feasible, we have $\left\lceil \frac{nnz(A)}{P} \right\rceil \leq \left\lfloor (1 + \epsilon) \frac{nnz(A)}{P} \right\rfloor$ by Equation (8). Combining these inequalities, we obtain

$$W(A_0, \ldots, A_{P-1}) \leq \left\lfloor (1 + \epsilon) \frac{nnz(A)}{P} \right\rfloor,$$

which implies the imbalance constraint in Equation (7) also is satisfied by the solution generated in Lemma A.1. Hence, this solution is also feasible for Problem 1.4, giving us a feasible solution to this problem with volume at most $(\min(m, n) + 1)(P - 1)$. □

A.3 Volumes exceeding the upper bound

For $P = 2$, the matrices for which Mondriaan 4.1 attained a total communication volume exceeding the upper bound given in Corollary A.3 are given in Table 11. For $P = 3$, the only occurrence was the matrix `JGD_Kocay/Trec12`, coming from a combinatorial problem, with volumes 1105 and 1117 while the upper bound equals 1104. For higher P, no solutions exceeded the upper bound.

Note that many matrices with high volumes seem to come from random matrices in the `Gset` group. However, for many other `Gset` matrices Mondriaan is successful, hence whether or not Mondriaan is bad at partitioning random matrices is not immediately clear.

Note that these are not the only cases, but rather the cases we came across while performing calculations, hence the matrices mentioned here should not be considered as a complete list.

ID	Group	Name	Upper bound	Achieved volumes
470	Gset	G10	801	$1034(4\times), 1035(6\times)$
480	Gset	G2	801	$1032(5\times), 1033(5\times)$
484	Gset	G23	2001	$2388, 2390, 2391, 2393, 2394, 2396, 2397(2\times), 2398, 2399$
486	Gset	G25	2001	$2386, 2387(2\times), 2388(2\times), 2389, 2390, 2394(2\times), 2400$
488	Gset	G27	2001	$2385, 2387, 2388, 2390(2\times), 2391, 2392, 2393(3\times)$
490	Gset	G29	2001	$2377, 2378, 2379, 2381(3\times), 2383, 2384, 2389, 2390$
492	Gset	G30	2001	$2386, 2387(2\times), 2388(2\times), 2389, 2390, 2394(2\times), 2400$
502	Gset	G4	801	$1030(8\times), 1031(2\times)$
506	Gset	G43	1001	$1188, 1190(2\times), 1193(2\times), 1195, 1196, 1197, 1198, 1201$
508	Gset	G45	1001	$1201, 1203(3\times), 1208(3\times), 1210, 1213, 1214$
510	Gset	G47	1001	$1196(3\times), 1197, 1198, 1199(2\times), 1202(3\times)$
524	Gset	G6	801	$1037(5\times), 1038(2\times), 1039(3\times)$
534	Gset	G8	801	$1035(3\times), 1036(7\times)$
1506	Pajek	Journals	125	$154(10\times)$
1532	Pajek	WorldCities	101	102
2196	JGD_SPG	EX2	561	$612(2\times), 618$
2198	JGD_SPG	EX4	2601	$2663, 2677, 2707$
2200	JGD_SPG	EX6	6546	$6673, 7137, 7173, 7192, 7220, 7254, 7290, 7292$

Table 11: All 18 *matrices from the standard test set of* 955 *matrices for which Mondriaan 4.1 finds solutions with volumes exceeding the upper bound, for* $P = 2$ *and* $\epsilon = 0.03$*, based on 10 runs. All* `Gset` *matrices come from undirected random graphs, the* `Pajek` *matrices also come from graphs, while the* `JGD_SPG` *matrices come from combinatorial problems.*

A.4 Imbalances exceeding ϵ

For $P = 2$ and $P = 4$, no solutions exceed the imposed imbalance $\epsilon = 0.03$. For $P = 16$ and $P = 64$, a few solutions do, as presented in Table 12. For $P = 3$ and $\epsilon = 0.03$ the behaviour is worse, as in 10 runs, 542 matrices of the 955 matrices in the standard test set fail the imbalance criterion in Mondriaan 4.1. Of these, 81 matrices always end up with an imbalance exceeding 0.03. For $P = 3$, the average imbalance over all 2440 solutions with too high imbalances is 0.128.

Note that these are not the only cases, but rather the cases we came across while performing calculations, hence the matrices mentioned here should not be considered as a complete list.

P	ID	Group	Name	Achieved imbalances $\epsilon' > 0.03$
16	24	HB	bcsstk02	$0.06, 0.69(9\times)$
16	1392	Andrianov	net75	$0.44, 0.58$
16	1471	Pajek	EVA	$0.30, 0.31$
16	2338	Rommes	ww_36_pmec_36	$0.072(10\times)$
64	2	HB	494_bus	0.61
64	24	HB	bcsstk02	$0.75(10\times)$
64	658	LPnetlib	lp_pilot_we	$0.54, 0.59, 0.69$
64	1471	Pajek	EVA	$1.74(2\times)$
64	1510	Barabasi	NotreDame_www	0.19
64	2021	JGD_Homology	ch6-6-b5	$0.07(10\times)$
64	2110	JGD_Homology	n4c5-b2	$0.64, 0.69$

Table 12: All 9 *matrices for which Mondriaan 4.1 finds solutions with imbalances exceeding* $\epsilon = 0.03$*, for* $P = 16$ *and* $P = 64$*, based on 10 runs. Note that 2 matrices with odd ids are present, as the results are taken from the control computations for the ZVS algorithm. The* `HB/bcsstk02` *matrix is special in this list: it is the only matrix with no zero entries in the entire standard test set.*

References

[1] Quicksort implementation in the GNU C Library (glibc). `https://sourceware.org/git/?p=glibc.git;f=stdlib/qsort.c;hb=HEAD`, accessed October 2016.

[2] B. O. Fagginger Auer and Rob H. Bisseling. Efficient matching for column intersection graphs. *ACM Journal of Experimental Algorithmics*, 19(1):1.3:1–22, 2014.

[3] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Softw. Pract. Exper.*, 23(11):1249–1265, November 1993.

[4] Rob H. Bisseling. Mondriaan package for sparse matrix partitioning. `http://www.staff.science.uu.nl/~bisse101/Mondriaan/`, September 2016.

[5] Rob H. Bisseling, Bas O. Fagginger Auer, A. N. Yzelman, Tristan van Leeuwen, and Ümit V. Çatalyürek. Two-dimensional approaches to sparse matrix partitioning. In *Combinatorial Scientific Computing*, chapter 12, pages 321–349. Chapman and Hall/CRC, 2012.

[6] Rob H. Bisseling and Wouter Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *ETNA. Electronic Transactions on Numerical Analysis [electronic only]*, 21:47–65, 2005.

[7] Ümit V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM J. Sci. Comput.*, 32(2):656–683, February 2010.

[8] T. Davis and Y. Hu. The SuiteSparse matrix collection (formerly known as the University of Florida sparse matrix collection). `http://www.cise.ufl.edu/research/sparse/matrices/`, accessed September 2016.

[9] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Çatalyürek. Parallel hypergraph partitioning for scientific computing. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 124–133, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Doratha E. Drake and Stefan Hougardy. Linear time local improvements for weighted matchings in graphs. In *Experimental and Efficient Algorithms: Second International Workshop, WEA 2003, Ascona, Switzerland, May 26–28, 2003*, pages 107–119. Springer Berlin Heidelberg, 2003.

[11] Doratha E. Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85(4):211 – 213, 2003.

[12] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.

[13] O. Fortmeier, H.M. Bücker, B.O. Fagginger Auer, and R.H. Bisseling. A new metric enabling an exact hypergraph model for the communication volume in distributed-memory parallel applications. *Parallel Computing*, 39(8):319 – 335, 2013.

[14] Bruce Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *Proceedings of the 5th International Symposium on Solving Irregularly Structured Problems in Parallel*, IRREGULAR '98, pages 218–225, London, UK, 1998. Springer-Verlag.

[15] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–15, 1962.

[16] Narendra Karmarkar and Richard M Karp. The differencing method of set partitioning. Technical report, Technical Report UCB/CSD 82/113, Computer Science Division, University of California, Berkeley, 1982.

[17] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 526–529, New York, NY, USA, 1997. ACM.

[18] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *International Conference on Parallel Processing*, pages 113–122, 1995.

[19] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *10th Intl. Parallel Processing Symposium*, pages 314–319, 1996.

[20] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, Feb 1970.

[21] D. M. Pelt and R. H. Bisseling. A medium-grain method for fast 2d bipartitioning of sparse matrices. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 529–539, May 2014.

[22] Daniël M. Pelt and Rob H. Bisseling. An exact algorithm for sparse matrix bipartitioning. *Journal of Parallel and Distributed Computing*, 85:79 – 90, 2015. IPDPS 2014 Selected Papers on Numerical and Combinatorial Algorithms.

[23] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way hypergraph partitioning via n-level recursive bisection. In *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*, pages 53–67, 2016.

[24] N. Selvakkumaran and G. Karypis. Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. In *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No.03CH37486)*, pages 726–733, Nov 2003.

[25] A. Trifunovic and W.J. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563–581, May 2008.

[26] Bora Uçar and Cevdet Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004.

[27] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[28] Tristan van Leeuwen. Expanding Mondriaan's palette. Master's thesis, Utrecht University, 2006.

[29] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47(1):67–95, January 2005.

[30] Nick Verheul. Partitioning of domains embedded in a regular grid. Master's thesis, Utrecht University, 2013.

[31] A. N. Yzelman and Rob H. Bisseling. Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing*, 31(4):3128–3154, 2009.

[32] A. N. Yzelman and Rob H. Bisseling. Two-dimensional cache-oblivious sparse matrix–vector multiplication. *Parallel Computing*, 37(12):806 – 819, 2011.

[33] Umit V. Çatalyürek and Cevdet Aykanat. A fine-grain hypergraph model for 2d decomposition of sparse matrices. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, IPDPS '01, pages 118–123, Washington, DC, USA, 2001. IEEE Computer Society.

[34] Umit V. Çatalyürek and Cevdet Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 28–34, New York, NY, USA, 2001. ACM.

[35] Ü. V. Çatalyürek. *Hypergraph Models for Sparse Matrix Partitioning and Reordering.* PhD thesis, Computer Engineering and Information Science, Bilkent University, November 1999.

[36] Ü. V. Çatalyürek and C. Aykanat. A hypergraph model for mapping repeated sparse matrix-vector product computations onto multicomputers. *Proceedings of International Conference on High Performance Computing, HiPC '95, Goa, India*, December 1995.