Universiteit Utrecht

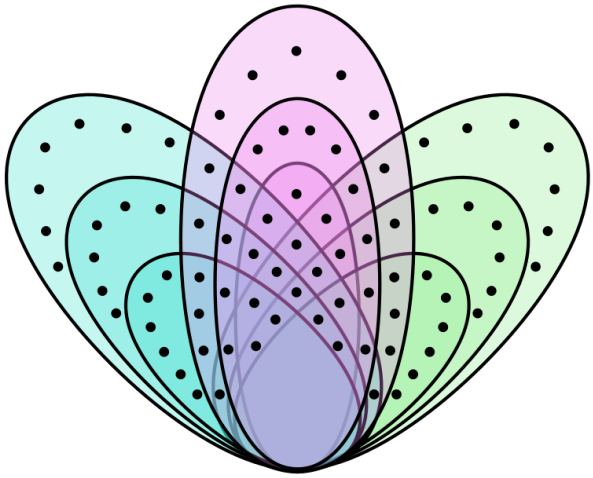
Faculteit Bètawetenschappen

PMondriaan: A Parallel Hypergraph Partitioner

Master Thesis

Sarita de Berg

Mathematical Sciences



Supervisor:

Prof. Dr. R. H. Bisseling
Mathematical institute, Utrecht

Supervisor:

J. Buurlage
CWI, Amsterdam

June 26, 2020

Abstract

Hypergraph partitioning is an important problem with many applications, such as very large scale integration design and sparse matrix distribution for parallel computation. The data provided by these applications has become so large, that serial partitioners have trouble handling these hypergraphs because of memory and/or running time limitations. In this thesis, a new parallel partitioning method is presented for a distributed-memory architecture. We extend the multilevel framework to a parallel multilevel framework that is used to recursively bipartition the hypergraph. The parallel coarsening algorithm uses a sampling scheme to reduce communication. The parallel refinement algorithm moves groups of vertices together, to limit the number of synchronizations. The method was implemented in a parallel hypergraph partitioner, named PMondriaan, that can partition a hypergraph into k parts using p processors. Our experiments show that PMondriaan achieves our goal of memory scalability, while providing state-of-the-art solutions. A reasonable speedup is achieved with respect to the single processor running times of PMondriaan, but no speedup is achieved with respect to the Mondriaan partitioner.

Contents

1 Introduction

Hypergraph partitioning is an important problem in graph theory. A hypergraph $H = (V, E)$ consists of a set of vertices V and a set of hyperedges E. Each hyperedge is a subset of the vertices. The goal of hypergraph partitioning is to partition the vertices of a hypergraph into k parts, such that each part has relatively equal size and the number of hyperedges that contain vertices belonging to different parts is minimized. This has many applications, such as very large-scale integration (VLSI) design [1, 2] and matrix decomposition for parallel computation [3–5]. Hypergraph partitioning is closely related to graph partitioning. For many applications, using hypergraph models instead of graph models offers more flexibility. For example, in matrix decomposition for parallel sparse-matrix vector multiplication (SpMV), a hypergraph-partitioning-based decomposition can accurately reflect the communication volume requirement [3].

Hypergraph partitioning is a well-studied problem. There are several software packages available: PaToH [3], hMeTiS [2, 6] and KaHyPar [7, 8] (general purpose), Mondriaan [4] (sparse matrix partitioning), MLPart [9] (circuit partitioning), UMPa [10] (directed hypergraph model, multi-objective), kPaToH [11] (multiple constraints, fixed vertices) and two parallel hypergraph partitioners, Zoltan [12] and Par$k$way [13]. All of these use the multilevel framework. This partitioning method consists of three phases: the coarsening, the initial partitioning, and the uncoarsening phase. In the coarsening phase, the size of the hypergraph is reduced in a number of steps by grouping vertices. In each step a new hypergraph is constructed, resulting in a number of *layers*. The coarsened hypergraph is partitioned in the initial partitioning phase. Finally, the partitioning is projected back to the previous layers and refined in the uncoarsening phase.

Different applications provide bigger and bigger inputs for hypergraph partitioners. These hypergraphs are so big, that traditional serial hypergraph partitioners are not able to handle them, because of memory or running time limitations [7]. A possible solution to these problems is the use of parallel computations on a distributed-memory system.

The goal of this thesis was to develop a prototype of a parallel hypergraph partitioner for a distributed-memory system, that achieves a reasonable speedup and is memory scalable, while providing state-of-the-art solutions. The successful multilevel framework will be the foundation of our new parallel partitioner: PMondriaan. PMondriaan achieves our goal of memory scalability, while providing state-of-the-art solutions. A linear speedup is achieved with respect to the single processor running times of PMondriaan, but no speedup is achieved with respect to the Mondriaan partitioner.

This thesis is structured as follows. In Chapter 1, we introduce the problem and some basic notation. We also discuss related work. In Chapter 2, we discuss the new parallel partitioning algorithm used in PMondriaan. In Chapter 3, we discuss some implementation

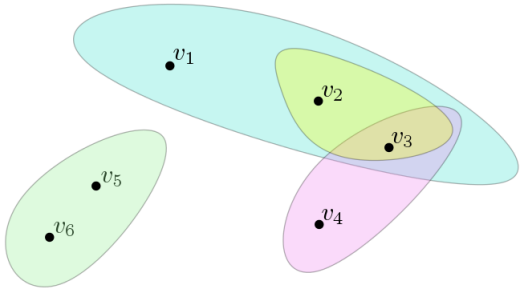


Figure 1.1: A hypergraph with six vertices, v_1, v_2, v_3, v_4, v_5, and v_6, and four hyperedges, $\{v_1, v_2, v_3\}$, $\{v_2, v_3\}$, $\{v_3, v_4\}$, and $\{v_5, v_6\}$.

details. In Chapter 4, we show the results achieved by PMondriaan. Finally, in Chapter 5, we provide our conclusions.

1.1 Preliminaries

1.1.1 Notation

Before we begin our discussion, let us first fix the notation used throughout this thesis. A hypergraph $H = (V, E)$ is a generalization of a graph. It consists of a set of vertices (or nodes) V and a set of hyperedges (or nets) E. Hyperedges connect two or more vertices of a hypergraph. We identify a hyperedge e by the subset $e \subseteq V$ of vertices it connects. The size of a hyperedge $e \in E$ is defined as $|e|$. An example of a hypergraph is shown in Figure 1.1. By $w(v) \geq 0$ we denote the weight of a vertex $v \in V$ and by $c(e) \geq 0$ the cost of a hyperedge $e \in E$. We denote by $I(v)$ the set of hyperedges containing $v \in V$. The degree $d(v)$ of v is defined as $|I(v)|$. In applications, often $w(v)$ is set to $|I(v)|$. We denote the maximum vertex degree in the hypergraph by $\Delta(H)$. We call a vertex v and u adjacent if there exists a net containing both vertices. By $N(v)$ we denote the set of adjacent vertices (or neighbors) of v. By $L(v)$ we denote the label of vertex v, that is the part v is assigned to.

We define the hypergraph partitioning problem as follows: given a hypergraph $H = (V, E)$ and an integer k, partition the vertices V into k nonempty, mutually disjoint parts $V_0, V_1, ..., V_{k-1}$ satisfying $\cup_{i=0}^{k-1} V_i = V$, such that all parts have relatively equal weight and a cut metric is minimized. We call $P = \{V_0, ..., V_{k-1}\}$ a partitioning and $V_0, ..., V_{k-1}$ its parts. For $k = 2$, we also call the problem the hypergraph bipartitioning problem. The weight W_i of part V_i is the sum of the weights of the vertices it contains. We say that a partitioning adheres to the balance constraint if

$$W_i \leq \frac{1}{k} \sum_{j=0}^{k-1} W_j (1 + \epsilon), \quad \text{for } i = 0, 1, ..., k-1, \tag{1.1}$$

where $\epsilon > 0$ is the load imbalance parameter. This ϵ expresses the relative maximum load imbalance that is allowed.

Under this constraint, we minimize a so-called cut metric. Common choices for a cut metric are the hyperedge-cut, the sum of external degrees (SOED) and the $(\lambda - 1)$-cut

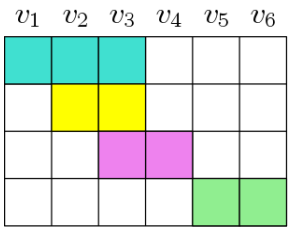


Figure 1.2: The matrix representing the hypergraph from Figure 1.1 using the row-net model. The colored elements represent the nonzeros.

metric [6]. Let λ_i be the number of parts spanned by $e_i \in E$. Then the hyperedge-cut metric minimizes (1.2a), the SOED metric minimizes (1.2b) and the $(\lambda - 1)$-cut minimizes (1.2c):

$$\text{(a)} \sum_{i:\lambda_i \geq 2} c(e_i), \qquad \text{(b)} \sum_{i:\lambda_i \geq 2} \lambda_i c(e_i), \qquad \text{(c)} \sum_{i:\lambda_i \geq 1} (\lambda_i - 1) c(e_i). \tag{1.2}$$

Here, we have generalized the definition of the SOED metric from [6] to account for hyperedge weights. For hypergraph bipartitioning, the hyperedge-cut and the $(\lambda - 1)$-cut metric result in the same minimization problem.

We can also view a hypergraph as a sparse matrix. We use the row-net model to express the hypergraph as a matrix. In this model, a row of the matrix represents a hyperedge (net) and a column a vertex of the hypergraph. For each $v_j \in V$ and $e_i \in E$ we have that a_{ij} is 1 if $v_j \in e_i$ and 0 otherwise. An example of such a matrix can be found in Figure 1.2. We denote by $nz(A)$ the number of nonzero elements in this matrix A. In this representation, the weights of the vertices and the cost of the hyperedges are given by two additional vectors. Note that any binary matrix can be transformed into a hypergraph by reversing this process.

We will use the following notations regarding parallel computing. By p we denote the total number of active processors. Following common notation in graph theory, we denote by s the index of the current processor (source) and by t the index of a remote processor (target). We denote a specific processor by $P(s)$ or $P(t)$. A subscript s is added to a set to indicate the local subset of that set. For example, we denote the local vertices by V_s. Note that this is the set of local vertices during the computation, which is generally not equal to one of the parts $V_0, ..., V_{k-1}$ of the final partitioning. The local hyperedge set, containing only the vertices in V_s, we denote by E_s. By $H_s = (V_s, E_s)$ we denote the local hypergraph.

1.1.2 The BSP model

The bulk-synchronous parallel (BSP) model consists of a computer architecture, a class of algorithms and a cost function [14]. The computer architecture on which the model is based is a distributed-memory system, where each identical processing unit has its own memory. Messages can be sent to and received from other processors through a communication network. BSP algorithms are based on the *single program multiple data* (SPMD) technique. This means that all processors execute the same program on different data.

A BSP program consists of a number of computation, communication and/or mixed *supersteps*. After each superstep all processors synchronize. The BSP cost function is

expressed in floating-point operations (flops) and charges a fixed cost of l for each synchronization. The cost of a computation superstep is therefore

$$T_{comp} = w + l, \tag{1.3}$$

where w is the maximum number of flops performed by any processor in the superstep. The cost of a communication superstep depends on the maximum number of data words h that is received or sent by any processor. We call this an *h-relation*. This amounts to a cost for a communication superstep of

$$T_{comm} = hg + l. \tag{1.4}$$

Here, g is a machine-dependent constant expressing the cost of communicating a single data word. It is the ratio of the communication cost relative to a flop execution.

1.2 Related Work

1.2.1 The multilevel framework

The hypergraph partitioning problem is NP-hard [15], even for bipartitioning. Therefore, in practice mostly heuristics are used. The most successful heuristic is the multilevel partitioning approach [16]. This framework consists of three phases: coarsening, initial partitioning and uncoarsening. The coarsening phase consists of a number of steps in which the size of the hypergraph is reduced in such a way that the structure of the hypergraph is preserved. The hypergraph is often coarsened by contracting similar vertices, for example using a matching algorithm. When the hypergraph has become sufficiently small, the coarsening stops. In the initial partitioning phase, the coarsened hypergraph is partitioned. The chosen partitioning method can be more expensive than the one used to partition the original hypergraph, because the size of the hypergraph is small. Finally, in the uncoarsening phase, the partitioning is mapped back to each previous layer and improved, often using an iterative refinement algorithm. A schematic overview of the multilevel method is shown in Figure 1.3.

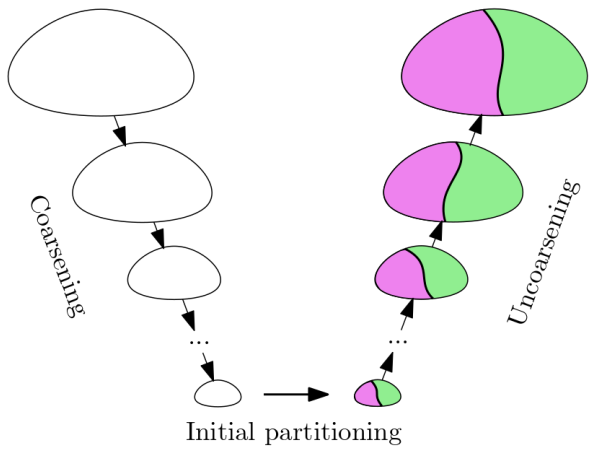


Figure 1.3: Schematic overview of the multilevel method. In the coarsening phase, the size of the hypergraph is reduced such that the structure of the hypergraph is preserved. In the initial partitioning phase, the coarsened hypergraph is partitioned. Finally, in the uncoarsening phase, the partitioning is mapped back to the previous hypergraphs and refined.

There are two main approaches to create k-way partitionings using the multilevel framework: recursive bipartitioning and direct k-way partitioning. We will first explain these approaches. In Section 2.2 we discuss the advantages and disadvantages of both, and provide motivation for our choice to use recursive bipartitioning in our new partitioner.

Recursive bipartitioning

In the recursive bipartitioning method, the set of vertices of the hypergraph is first partitioned into two parts V_0 and V_1. The induced hypergraphs of V_0 and V_1, containing only the vertices in the respective set, are again bipartitioned. We bipartition the resulting hypergraphs recursively until we reach the desired number of parts. Because all parts are disjoint, we now have a partitioning of the entire hypergraph. We use the multilevel approach for each bipartitioning, so in total we apply the multilevel scheme $k-1$ times. Note that, when k is not a power of two, we cannot create a good k-way partitioning by only splitting the graph into parts of equal weight. To create k-way partitionings when k is not a power of 2, we need to split the graph into uneven parts at some levels. For example, if we want three parts, we first bisect the graph into parts of size roughly equal to 1/3 and 2/3 of the total weight. In the next bisection step, we bipartition only the bigger part corresponding to 2/3, to obtain three parts of roughly equal weight.

Direct k-way partitioning

In the direct k-way partitioning method, the hypergraph is split directly into k parts in the initial partitioning phase. This partitioning is later refined in the uncoarsening phase with more involved refinement algorithms that can cope with k parts.

Coarsening methods

The idea of all coarsening methods is to contract (groups of) vertices that are similar, so the structure of the hypergraph is preserved, but the size of the hypergraph is reduced. When we contract two or more vertices $v_1, ..., v_j$ into a new vertex u, we set $w(u) = \sum_{i=1}^{j} w(v_i)$ and $I(u) = \bigcup_{i=1}^{j} I(v_i)$. We can divide most coarsening methods into two categories [17]: matching-based [3, 4, 9, 11, 12] and clustering-based [3, 6, 13, 18] algorithms. In matching-based algorithms, we always contract two vertices into one new vertex. In clustering-based algorithms, a group of more than two vertices is also allowed to form a new vertex. The new vertex gets assigned a weight equal to the sum of the weights of its children, and is included in every hyperedge that contains at least one of its children. Most of these methods use a rating function to determine the similarity of two vertices. Examples of these functions are the inner product [3], which counts the number of nets two vertices have in common, and the heavy-edge rating function [7], which adjusts this value for the size of the nets, emphasizing nets with fever vertices.

Uncoarsening methods

In every step of the uncoarsening phase, the previous partitioning is projected to the next layer as follows. If two vertices v and u have been contracted into a vertex v' that has label $L(v')$, both v and u will get assigned label $L(v')$. For contractions of clusterings, all vertices contracted into v' get assigned label $L(v')$. After this, the partitioning is improved using an

iterative refinement scheme. For some examples of iterative refinement schemes we refer to [19].

Most iterative refinement schemes are based on the FM method [20], named after Fiduccia and Mattheyses. The original FM heuristic improves a bipartitioning in multiple passes. A pass starts by computing all gain values, defined as the change in the objective function when moving a vertex to the other part. The vertex with highest gain, among all moves that do not violate the balance constraint, is selected. This vertex is moved to the other part and is locked, so it cannot be moved back later in the pass. Subsequently the relevant gain values are updated and new vertices are repeatedly selected. This continues until no vertex can be moved anymore. During the algorithm, the best partitioning encountered is stored. Finally, the moves that took place after reaching the best partitioning of the pass are reversed. Because vertices with negative gain can also get moved, this is not necessarily the last partitioning. Using a specialized data structure and smart gain updates, this can be implemented such that the computation time of each pass is linear. In practice, the algorithm usually converges within a few passes, giving a nearly linear algorithm.

1.2.2 Parallel hypergraph partitioners

As discussed before, hypergraph partitioning is a widely studied problem and there are several partitioners available. In this section, we will take a closer look at the two parallel hypergraph partitioners, Zoltan [12] and Par$k$way [13], as these are most relevant to this project. The parallel partitioners were developed simultaneously. The biggest difference in the design of Zoltan [12] and Par$k$way [13] is the used data distribution. Zoltan uses a so-called Cartesian 2D distribution, while Par$k$way uses a 1D distribution, as explained in Section 2.1. Another important distinction is that Zoltan uses recursive bipartitioning, while Par$k$way uses the direct k-way partitioning method. In the next sections we will describe these parallel partitioners in more detail.

Zoltan

The parallel hypergraph partitioner Zoltan was initially released in 2006 [12]. Zoltan uses a multilevel approach using recursive bipartitioning to obtain k-way partitionings. The data is distributed using a Cartesian 2D distribution of the matrix corresponding to the hypergraph.

In the coarsening phase, a parallel inner product matching algorithm is used. The algorithm consists of multiple rounds. In each round, a set of unmatched local vertices called *candidates* is sent to the other processors. Next, the inner products of the local vertices with the received candidates are computed. Finally, the globally best matches are computed and communicated. To get the candidates to the correct processors and to compute the global inner products, both horizontal and vertical communication is necessary. Here, horizontal and vertical refer to the underlying Cartesian 2D distribution. After the specified number of rounds, all vertices have been matched and are contracted. Because the matching requires a lot of communication, this phase is often the most time consuming part of the entire partitioning algorithm.

In the initial partitioning phase, each processor greedily computes a bipartitioning using a randomized algorithm. The globally best partitioning is selected. Because the coarsened hypergraph is relatively small, every processor can hold the entire hypergraph in their local memory.

In the uncoarsening phase, a refinement method based on the FM heuristic is used. First, the gain values of all vertices are computed. Next, in each processor column a processor is selected as *vertex mover*. This vertex mover tries to move each of its vertices to the other part without violating the balance constraint. When a vertex is moved, only the gain value of local neighbors are updated. This way, no communication is necessary. Only updating locally could of course affect the quality of the partitionings, but the authors conclude the partitionings produced are quite good for a not too large number of processors.

Two approaches to parallel recursion are described in the original paper on Zoltan. The first method uses all processors to first recursively produce the partitioning of one part of each bipartitioning. The other method splits the processors into two sets, where the first set works on one part of the bipartitioning and the other set on the other part. Splitting the processors is shown to reduce execution time and improved quality, and therefore this method is used in all experiments.

Experiments with different processor configurations $p_x \times p_y$ are also discussed. The configurations with p_x small seem to work best. From the 1D distributions, $p_x = 1$ works a bit better than $p_y = 1$. Using the $(\lambda - 1)$-cut metric, the quality of the partitionings found is similar to those of PaToH. The running times are much shorter in most cases than those of the next partitioner we discuss in detail: Par$k$way.

Par$k$way

Par$k$way [13] uses a multilevel approach with direct k-way partitioning. The data is distributed among the processors in a 1D fashion. The vertices are distributed among the processors by a block distribution. Randomization can be used to improve load balance, however this could remove a structure from the problem that is actually useful to exploit. At the start, hyperedges are also allocated consecutively. Later, they are allocated using a hash function, which has the advantage that duplicates can be found easily. At the start of each level of the coarsening and uncoarsening, each processor assembles all hyperedges that are incident to one of its vertices. This results in the duplication of data, but reduces the amount of communication needed.

In the coarsening phase, a parallel matching based on the First-Choice or the Heavy Connectivity Clustering algorithm is used [3, 6]. Because each processor has access to all hyperedges incident to its vertices, it can locally compute matches for its vertices. If the desired match is located on another processor, a match request is sent to that processor. These are handled by each processor s by first considering all requests from processors $t < s$. In the next step, the request from processors $t > s$ are handled. This way, a vertex cannot be matched more than once. After this, a load-balancing step is included, to make sure every processor has the same number of vertices at the start of the next coarsening step.

The initial partitioning phase is similar to that of Zoltan: the best partitioning is selected from the partitionings computed by all processors separately. In the uncoarsening phase, a parallel version of greedy k-way refinement [6] is used. Contrary to the serial algorithm, the parallel algorithm moves sets of vertices simultaneously. Each processor determines the best move for every vertex that does not violate the balance constraint, and moves these vertices. To make sure there are not too many conflicting moves, the vertices are moved in two phases. Moves from higher index parts to lower index parts are performed, before the other moves. During this process, the balance constraint is maintained through global communication. This induces more communication, but was chosen because local constraints limit the exploration of the solution space.

An additional refinement method is introduced as part of the partitioning process: parallel multi-phase refinement. After partitioning the hypergraph, the resulting partitioning is refined using another multilevel scheme. Only the coarsening phase differs from the previously described algorithm. In this phase, only matches between vertices in the same part (on the same processor) are allowed. This means no communication is necessary, leading to an efficient algorithm.

The parallel greedy k-way algorithm was shown to give reasonable speedups and high quality partitions on hypergraphs with small maximum hyperedge degrees and/or a high degree of structural locality. The multi-phase algorithm produced even better partitionings, but had a significantly larger running time.

1.2.3 Recent advances

In this section we will discuss some interesting recent advances in hypergraph partitioning.

Using n-level coarsening

A hypergraph partitioner called KaHyPar that uses an extreme multilevel scheme, namely using n levels, with $n = |V|$, was released in 2016 [7]. This partitioner uses the recursive bipartitioning approach. Later, the partitioner was used as a basis for the development of a direct k-way partitioner that uses n levels [8].

We briefly explain the algorithm used in KaHyPar [7]. In each level of the coarsening phase, only a single pair of vertices is contracted, which results in n levels. The selection of the vertices to be contracted is based on the heavy-edge rating function. This is scaled inversely to the product of the vertex weights, to keep the weights in the coarser hypergraph more uniform. The resulting rating function is:

$$r(u,v) := \frac{1}{w(u)w(v)} \sum_{e \in I(u) \cap I(v)} \frac{c(e)}{|e|-1}. \tag{1.5}$$

When the coarsened hypergraph has become small enough, the hypergraph is partitioned using different algorithms and the best partitioning is chosen for uncoarsening. In the uncoarsening phase, a localised FM search is used. Here, only the area around the just uncontracted vertices gets refined in each step.

Using efficient data structures and caching, the resulting KaHyPar tool is not much slower than other tools, such as PaToH and hMetis, and produces better partitionings in most cases. Parallelizing this framework is definitely not straightforward, as in every step of (un)coarsening there is not much work to be divided. Furthermore, parallelizing the optimizations necessary for this approach to be viable, seems to be out of the scope of our project. Therefore, we will not use this n-level approach in our partitioner.

Label propagation

Label propagation is commonly used for cluster detection in graphs, and recently found a new application in partitioning large graphs. The PuLP (Partitioning using Label Propagation) method is a parallel and memory-efficient graph partitioning method [21]. The method was later adapted to work for hypergraph partitionings [22]. At the same time, other parallel methods were introduced for partitioning very large graphs [23, 24]. These methods use a

combination of the multilevel approach and label propagation. In 2017, an updated version of PuLP was released, called XtraPuLP [25].

The advantage of label propagation is that it has linear time complexity, assuming we stop after a small constant number of iterations. They are also memory-efficient, in contrast to most multilevel methods [21].

The algorithm, summarized in Algorithm 1.2.1, works as follows. First, each vertex $v \in V$ is assigned a random label from the set $\{0, ..., k-1\}$. Then, each vertex is visited in a random order and assigned the label that is most common among its neighbors. This is repeated until a set number of iterations has been reached, or the labeling has become stable, meaning all vertices have the label that is most common among their neighbors. This process results in a partitioning, where a set of vertices with the same label corresponds to a part in the partitioning. To get partitionings that optimize one or more objectives, the PuLP method further improves this partitioning. The improvement consists of two steps for each objective. In the first step, the partitioning is improved using a slightly altered label propagation algorithm to optimize the desired metric and to conform to the balance constraints. In the second step, the partitioning is improved even further using another iterative refinement scheme, such as the FM method. By repeating these steps, we can optimize the partitioning for multiple objectives.

Algorithm 1.2.1 Basic label propagation. The Random(i, j) function generates an integer in $\{i, ..., j\}$ uniformly at random.

Input: Graph $G = (V, E)$, number of labels k, maximum number of iterations I.
Output: Labels L for all vertices.

```
for all v ∈ V do
    L(v) ← Random(0, k − 1)
iter := 0
change := True
while iter < I and change do
    change ← False
    for all v ∈ V do
        for i := 0 to k − 1 do
            C[i] ← 0
        for all u ∈ N(v) do
            C[L(u)] ← C[L(u)] + 1
        m ← argmax_{0≤i<k} C[i]                ▷ The majority label
        if m ≠ L(v) then
            L(v) ← m
            change ← True
    iter ← iter + 1
return L
```

The medium-grain method

In the medium-grain method [5], a matrix is not partitioned directly by using the row-net or column-net model, which create 1D distributions. Instead, the matrix A is first translated to a new matrix B, that is then partitioned using the row-net model. By creating B as follows, a 1D partitioning of B corresponds to a 2D partitioning of A. This way, we can

create a 2D partitioning, while keeping the advantages of easy 1D partitioning. To create this new matrix, the original $m \times n$ matrix A is first split into two parts A_r and A_c such that $A = A_r + A_c$. Each nonzero element a_{ij} is assigned to either its row or column and put into A_r or A_c respectively. The choice for the row or column is based on the number of nonzeros in the row and column: the one with the least nonzeros is chosen. The actual partitioning is performed on the matrix:

$$B = \begin{bmatrix} I_n & A_r^T \\ A_c & I_m \end{bmatrix}, \tag{1.6}$$

where I_k is the $k \times k$ identity matrix. In this method, large nets and vertices of high degree are split. It follows that average net sizes and vertex degrees are relatively small. The medium-grain method is an important application of our hypergraph partitioner. Therefore, we will optimize our partitioner for this use case with low vertex degrees.

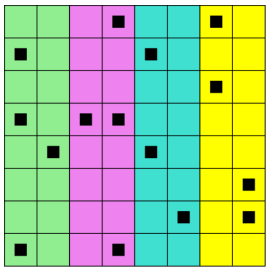
2 Design of the parallel partitioning algorithm

In our parallel hypergraph partitioner we use a multilevel approach. It was previously shown that this framework is able to achieve good results [2, 3, 6–8]. Within the multilevel framework there is still a lot a freedom in making algorithmic choices. For example, we need to choose how to divide the data over the processors. This choice will affect both the running time and the involvement of the algorithms. In Section 2.1, we explain our choice of using a 1D distribution, in which the vertices are divided over the processors. Also, we need to choose one of the two approaches for creating a k-way partitionings using a multilevel algorithm: recursive bipartitioning or direct k-way partitioning. Both of these approaches are viable. In Section 2.2, we will motivate our choice for recursive biparitioning by discussing the advantages and disadvantages of both approaches. In Section 2.3, the new parallel bipartitioning algorithm is introduced. We give a general overview of the algorithm and discuss the coarsening and uncoarsening algorithms used in more detail.

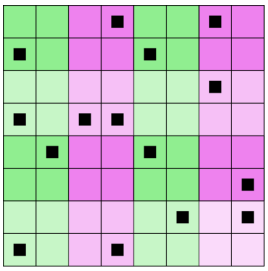
2.1 Data distribution

There are two obvious choices to divide the hypergraph over p processors: a 1D or a 2D distribution. Here, 1D or 2D refers to the division of the associated sparse matrix, as shown in Figure 2.1. In a 1D distribution, we can divide the rows/hyperedges or the columns/vertices over the processors. Par$k$way [13] uses a combination of these approaches by dividing both the vertices and the hyperedges over the processors. Penalty is that the hypergraph is stored twice, however operations such as finding all vertices in a hyperedge are faster. Zoltan [12] uses a 2D approach instead. The main disadvantage of using a 2D approach is that the algorithms become more involved. In this approach, horizontal and vertical communication between processors is necessary, for example when finding a good match for a vertex in the coarsening phase. An advantage of the 2D approach is that it could improve the running time [12].

For our new partitioner, we prioritize simplicity over efficiency in this case, so we have chosen a 1D distribution. This still leaves us with a choice: distributing the vertices or hyperedges over the processors. In [12], slightly better results were obtained by distributing the hyperedges instead of the vertices. Again, this would result in more complex algorithms. As the improvements reported were only small, we have chosen to distribute the vertices over the processors. Each processor also maintains information on the locally non-empty nets: the cost, the global size, and the local vertices it contains.



a 1D block distribution



b Cartesian 2D distribution

Figure 2.1: A 1D and 2D distribution of a hypergraph over four processors. Here, 1D and 2D refers to the distribution of the associated sparse matrix, shown in the figures. The colors and shades represent the processor that owns the element and the squares the nonzero elements.

In Section 1.2.3, we discussed the medium-grain method, an important application of our hypergraph partitioner. This method creates a hypergraph with small net sizes and vertex degrees. Because of this, storing all pins of a vertex on the same processor will not result in too much load imbalance. This also favors the 1D partitioning approach.

2.2 Recursive bipartitioning versus direct k-way partitioning

Our choice to use recursive bipartitioning (RB) and not direct k-way partitioning, is based on three main considerations: solution quality, running time and whether the approach is suitable for parallelization. In this section, we will discuss previous results relating to these considerations, to support our choice for recursive bipartitioning. With regard to the solution quality and running time, we will mostly consider the three most used hypergraph partitioners, PaToH [3], hMeTiS [2, 6] and KaHyPar [7]. All of these include a recursive and a direct version, which makes them suitable for comparison of the two approaches. The direct k-way variants of the partitioners are called kPaToH [11], hMeTiS-Kway [6], and KaHyPar-K [8] respectively.

Solution quality

The oldest of the k-way partitioners is hMeTiS-Kway. In [6], the RB algorithm is compared to the direct k-way algorithm using the ISPD98 circuit partitioning benchmark suite [26]. They found that hMeTiS-Kway produces solutions of comparable quality for the hyperedge-cut metric. However, for the SOED metric, the k-way algorithm performs on average 4.8%, 6.8% and 7.6% better for $k = 8$, 16 and 32. This can be explained by the fact that SOED is a more global metric than the hyperedge-cut metric, and therefore requires a more global view of the problem. In [11], PaToH and kPaToH are compared using the $(\lambda - 1)$-cut metric. In the experiments, 12 matrices from the SuiteSparse collection [27], formerly called the University of Florida Sparse Matrix Collection, are used. Again, the k-way method finds better solutions, 4.82% on average for $k = 32$

More recently, a direct k-way partitioning version of KaHyPar was introduced [8] and compared to all of the previously mentioned partitioners. The authors used the $(\lambda - 1)$-cut metric. A much larger benchmark set is used, that includes includes hypergraphs from the ISPD98 circuit partitioning benchmark suite [26], the SuiteSparse collection [27], and the international SAT Competition 2014 [28]. From preliminary experiments, the authors conclude that kPaToH performs significantly worse than PaToH, so this partitioner is not included in additional experiments. They conclude that hMeTiS-Kway performs worse than hMeTiS, contradicting previous results. A reason for this contradiction could be that the recursive bipartitioners have been improved over the years, while the k-way partioners have not. Another reason could be that the original benchmarks sets were too small to draw any conclusions. A difference is made between hypergraphs with median net size ≥ 3 and < 3. For the hypergraphs with a lot of small net sizes, both the RB variants KaHyPar and hMeTiS outperform KaHyPar-K in solution quality, while for the larger median net sizes it is the other way around. We can conclude that direct k-way partitioning algorithms are better at handling hypergraphs with many large nets.

Another important aspect related to the solution quality is the balance of the computed solutions. A benefit of using direct k-way partitioning, it that it is possible to set a stricter balance constraint. Using recursive bisection, the balance constraint has to be adapted at each step, so that the final partitioning still adheres to the overall balance constraint. This limits the solution space.

Running time

We now discuss the conclusions of the papers with regard to running time. In the comparison between hMeTiS and hMeTiS-Kway [6], the authors found that hMeTiS-Kway is about twice as fast as hMeTiS. In [11], kPaToH is again faster than PaToH, a speedup of 1.88 is achieved for $k = 32$. The recently introduced direct k-way partitioning version of KaHyPar [8], also finds that the new k-way partitioner is a least 2.5 times faster than the RB version. However, we note that, while the k-way versions of hMeTiS and KaHyPar are faster than their RB variants, PaToH still remains the fastest partitioner. This implies that optimization also has a significant effect on running time.

Parallelization

Finally, we consider the parallelization of both approaches. We discuss the parallelization of the three phases of the multilevel framework and the approaches as a whole. The first phase, where the hypergraph is coarsened, is similar for both approaches. The differences are in the initial partitioning and the uncoarsening phase.

In the initial partitioning phase, the hypergraph is partitioned into either 2 or k parts in the different methods. However, this phase is always easy to parallelize. Because the hypergraph is small in this phase, a copy can be made available on each processor and each processor can compute a partitioning separately. After creating these partitionings, the best partitioning is selected for the uncoarsening phase. This works for both the recursive bipartitioning and the direct k-way partitioning method.

The uncoarsening phase is more complex in direct k-way partitioning, because refinement algorithms for k-way partitionings are more complex. Most of the time, at least k priority queues are used [8]. Because priority queues have a very global view, this is difficult to parallelize and might induce a lot of communication.

	Coarsening (%)		Uncoarsening (%)		Avg. run time	
Matrix	PaToH	kPaToH	PaToH	kPaToH	PaToH	kPaToH
`language`	73.5	45.2	19.1	47.4	12.266	9.721
`pre2`	84.5	32.1	9.5	59.8	24.406	15.070
`hood`	86.3	56.8	4.6	27.1	15.693	5.386

Table 2.1: Results presented in [11] of the comparison of percentage breakdown of execution of PaToH and kPaToH for $k = 32$ including the average running times, no unit was given in the paper.

In the direct k-way partitioning method, we only go through each phase once, while in the recursive bipartitioning method, we have to perform $k-1$ bipartitionings, so we go through each phase $k-1$ times. In [11], the time spent by both approaches in the different phases is measured. The relevant results regarding the time spent in the coarsening and uncoarsening phase are shown in Table 2.1. We see that the percentage of time spent in the coarsening phase is much larger for the recursive method than the direct one and the reverse is true for the uncoarsening phase. This can be explained by the fact that refinement methods are much more expensive for k-way partitionings than bipartitionings, as previously discussed. Because the coarsening phase seems better suited for efficient parallelization, this suggests using recursive bipartitioning might be better for parallelization.

Another important aspect of parallelization is how to split the tasks over the processors. For recursive bipartitioning there are two main options. One option is to leave the data in place after a split and first let all processors work on one of the smaller problems and then continue with the other subproblem. Alternatively, we can split the processors into two groups and let the first group work on one subproblem and the second on the other. The disadvantage of this approach is that the data has to be redistributed. On the other hand, it should result in less communication. In [12], both these approaches are compared. The authors found that the second option improved both quality of the solution and running time. This approach can also be used to optimize the distribution of a hypergraph over a machine that has multiple layers, for example a machine with two nodes that both have four cores. In this case, the first split can be optimized for different parameters, for example a larger allowed imbalance, than the later splits. In the direct k-way partitioning method, it is not possible to split the processors, because during the entire computation a global view of the problem is required.

2.2.1 Design choice

The direct k-way partitioning method has a lower runtime than recursive bipartitioning, but this method does not seem to consistently improve solution quality. We have also seen that RB has difficulty with hypergraphs with many large nets. But, since our partitioner will mostly be used in the medium-grain method, the size of large nets has already been reduced. Recursive bipartitioning has an advantage in parallelization: we can split the processors into groups so we induce less communication. This can also be used to optimize the partitioning for different machine architectures. Because we think the parallelization of the method and solution quality are more important than running time, we have chosen to use recursive bipartitioning.

2.3 Overview of the parallel bipartitioning algorithm

As discussed before, our parallel bipartitioning method is based on the multilevel framework. We have extended this framework to a parallel multilevel framework. This is shown schematically in Figure 2.2. At the start of the algorithm, the vertices are distributed over the processors, as described in Section 2.1. We start with a number of parallel coarsening steps. We stop coarsening in parallel, when the parallel algorithm can no longer significantly decrease the size of the hypergraph. When this point is reached, the entire hypergraph is communicated among all processors, so that every processor has a full copy of the coarsened hypergraph. Next, each processor coarsens this hypergraph further sequentially, computes an initial partitioning, and sequentially uncoarsens the partitioning to reverse the local coarsening. Each processor now has a, potentially different, partitioning of the coarsened hypergraph. The best of these partitionings is chosen and communicated to all processors. We then translate this partitioning to the (same) distributed hypergraph. Finally, the distributed hypergraph is uncoarsened in parallel to find a partitioning of the original hypergraph. In the Section 2.4, 2.5, and 2.6, we will discuss the three phases of the parallel multilevel framework in more detail.

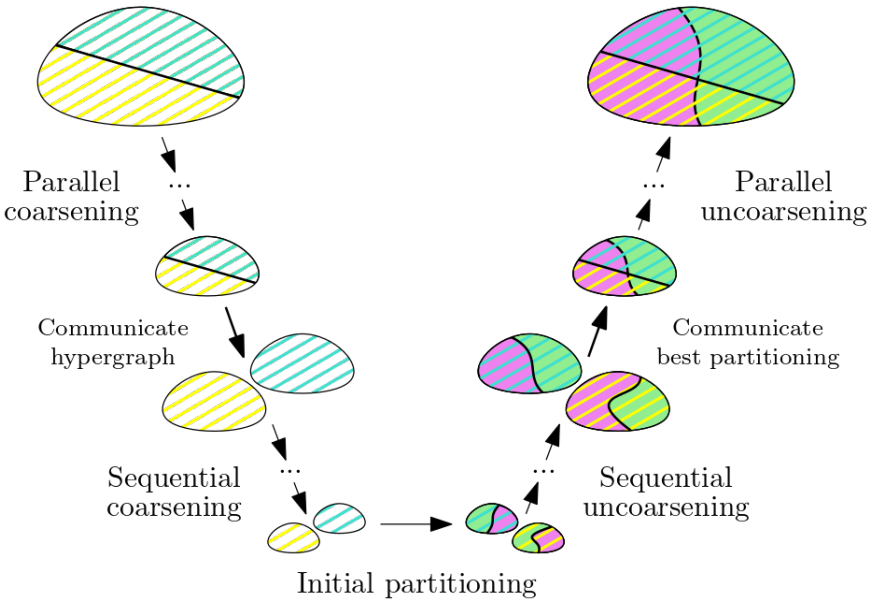


Figure 2.2: An overview of the parallel bipartitioning method for two processors. The yellow shaded areas represent the vertices on processor 0 and the blue shaded areas the vertices on processor 1. The distributed hypergraph is first coarsened in parallel. When it is small enough, each processor receives a local copy of the coarsened hypergraph. This is coarsened further, an initial partitioning is created and the partitioning is uncoarsened up to the first sequential level. The best partitioning found by any processor is communicated to all processors and translated to the corresponding partitioning of the distributed coarsened hypergraph. Finally, the distributed hypergraph is uncoarsened further in parallel.

2.3.1 Redistribution of the vertices

After every bipartitioning, the vertices need to be redistributed. As discussed before, we split the processors into two groups, so they can work on separate problems. To divide the hypergraph evenly over the processors, we introduce a balance parameter η. This η puts a balance constraint on the distribution of the hypergraph during the computations. The maximum weight a processor can be assigned after a bipartitioning is:

$$W_{\max} = \begin{cases} \frac{W_0}{|P_0|}(1+\eta), & \text{if } s \in P_0, \\ \frac{W_1}{|P_1|}(1+\eta), & \text{if } s \in P_1. \end{cases} \tag{2.1}$$

Here, P_0 and P_1 are the sets of processors assigned to part 0 and 1 respectively. The number of processors assigned to the parts depends on the ratio of the previous split. For example, if the previous bipartitioning used a ratio of 4:5, the processors will also be assigned to the parts using this ratio.

We want to redistribute the hypergraph over the processors in such a way that processors in P_0 only contain vertices of part 0 and processors in P_1 only vertices of part 1. To achieve this, we first reassign vertices of part 0, and then of part 1. We will discuss the reassignment of vertices of part 0, the approach for part 1 is similar. By $W_{0,\text{loc}}$ we denote the total weight of all local vertices assigned to part 0. We define the *surplus* of a processor as the weight that can still be assigned to that processor. This is negative if a processor has too much weight assigned to it. For a processor in P_0, the surplus is $W_{\max} - W_{0,\text{loc}}$. A processor in P_1 cannot contain any vertices of part 0, so the surplus is $-W_{0,\text{loc}}$. After communicating all of these surplusses, each processor can reassign its vertices of part 0.

We use a ring algorithm to reassign the vertices. We start at the next processor, and assign as many vertices to this processor as possible. We repeatedly go to the next processor and reassign vertices, until the local surplus is no longer negative. During this process, we keep track of the *rest*: the surplus of other processors that still needs to be reassigned first. When we consider a new processor, we add the surplus of that processor to *rest*. If *rest* is positive after this, meaning that the surplus of all other processors up to this processor has been reassigned, we can assign local vertices of maximally *rest* weight to this processor. The weight we can reassign is also bounded by local weight that still needs to be reassigned, so we assign $\min(\mathit{rest}, -\mathit{surplus}_s)$ weight to this processor. Note that $\mathit{surplus}_s$ is negative while we still have vertices to reassign. Before going to the next processor, we set *rest* to $\min(0, \mathit{rest})$, because the other processors can never assign more than their local surplus, so *rest* should never be positive.

In this process, we are not very strict in our balance constraint. When reassigning vertices to another processor, we always reassign at least one vertex, even if the weight of this vertex is larger than $\min(\mathit{rest}, -\mathit{surplus}_s)$. This way, all vertices in part 0 of processors in P_1 always get reassigned to a processor in P_0. However, this could have the result that $W_{0,\text{loc}} > W_{\max}$ for some processor(s) in P_0. As the maximum weight can only be exceeded by the weight of one vertex, we prefer this over using a more complicated algorithm using more communication. The algorithm to redistribute the vertices of part 0 is shown in Algorithm 2.3.1.

Algorithm 2.3.1 Parallel redistribution of the vertices of part 0 for processor $P(s)$, with $0 \leq s < p$

Input: Distributed hypergraph $H = (V, E)$, P_0, P_1 sets of processors assigned to part 0,1.
Output: Redistributed hypergraph $H = (V, E)$, where only $P(t) \in P_0$ have vertices of part 0.

```
function RedistributeHypergraph(H)
    if P(s) ∈ P_0 then
        surplus_s := W_max − W_0,loc
    else
        surplus_s := −W_0,loc
    for t := 0 to p − 1 do
        Put surplus_s in P(t)
    j := 0                                   ▷ Index of next vertex to be considered
    t := (s + 1) mod p
    rest := 0                                ▷ Surplus of the other processors to be reassigned
    while surplus_s < 0 do
        rest ← rest + surplus_t
        if surplus_t > 0 and rest > 0 then
            max ← min(rest, −surplus_s)
            sent := 0
            while sent < max do
                while L(v_j) ≠ 0 do
                    j ← j + 1
                Send v_j to P(t)
                sent ← sent + w(v_j)
                Remove v_j from H
            surplus_s ← surplus_s + sent
        rest ← min(rest, 0)
        t ← (t + 1) mod p
    Add all received v to H
```

2.4 Coarsening

In this section, we describe the coarsening scheme used in our parallel bipartitioning algorithm. We start by briefly discussing our sequential coarsening method, because our parallel method is based on this. Next, we describe the parallel coarsening scheme in detail.

2.4.1 Sequential coarsening

The coarsening algorithm used in PMondriaan is clustering-based, so groups of vertices are contracted into a new vertex in each coarsening step. The rating function we use is based on the default rating function used in Mondriaan [29]. The inner product is scaled by the net cost, the net size, and the minimum of the degrees of the two vertices. This results in the rating function

$$r(u,v) := \frac{1}{\min(d(u),d(v))} \sum_{e \in I(u) \cap I(v)} \frac{c(e)}{|e|-1}. \tag{2.2}$$

To make sure that not too many vertices are matched to the same vertex, we set a maximum cluster size. We now find matches for the vertices greedily. The vertices are visited in a random order. Initially, each vertex forms a singleton cluster. When an unmatched vertex v is visited, we first compute $r(u,v)$ for all u adjacent to v. The best match that is not already part of a full cluster, is computed. Finally, v is matched to this vertex and added to its cluster. If no such vertex exists, v is not matched. After visiting all vertices, each cluster is contracted into a vertex in the coarsened hypergraph, as described in Section 1.2.1.

2.4.2 Parallel coarsening

The parallel coarsening algorithm used in PMondriaan is also clustering-based. Most coarsening methods consider all neighbors of a vertex for matching. However, in the parallel case this is quite expensive, as many neighbors might reside on other processors. We therefore introduce a method based on *sampling*. In this method only a select group of samples is considered when finding a match for a vertex. An overview of the algorithm can be found in Algorithm 2.4.2.

First, ℓ samples are generated, here ℓ is the chosen sample size. Each processor generates ℓ/p samples, which gives ℓ samples in total. We assume ℓ is divisible by p, otherwise $p\lfloor \ell/p \rfloor$ samples are used. When creating a partitioning into $k > 0$ parts, the lower levels in the recursive bipartitioning use a scaled sample size of $p'\lfloor \ell/p \rfloor$, where p' is the number of processors in the current group.

After generating the samples, each processor sends its local samples to all other processors. These samples are considered for matching by the local vertices. Vertices are only allowed to match with a sample, not another local vertex. Again, we enforce a similar size restriction. The algorithm to compute the necessary inner products is shown in Algorithm 2.4.1. In the algorithm, we have not scaled the inner products for simplicity. In reality, the inner products are scaled using the same rating function as in the sequential coarsening. In the algorithm, all of the inner products are stored, but in PMondriaan, we only keep track of the best inner product found so far for each vertex. This way, we reduce the memory usage of this function by a factor of ℓ.

To enforce a global restriction of $c_{\max}$ on the cluster size, the final clustering is computed in two phases. In the first phase, each processor requests a match for each vertex v matched

Algorithm 2.4.1 Inner product using samples

Input: Hypergraph $H = (V, E)$, set of sample vertices S.
Output: $ip_v(u)$ contains inner products of v and u for all $v \in V, u \in S$.

```
function InnerProduct(H, S)
    Initialize ip_v(u) ← 0 for all v ∈ V, u ∈ S
    for all u ∈ S do
        for all e ∈ I(u) do
            for all v ∈ e do
                ip_v(u) ← ip_v(u) + 1
    return ip
```

to a sample u at the owner of u. A request consists of the index of the processor, the sample index, the vertex index and the scaled inner product value. In the second phase, each processor considers the requests for each of its samples and accepts the best $c_{\max}$ samples, based on the inner product values. We have chosen this two-phase scheme over a scheme where each processor is only allowed to match $c_{\max}/p$ vertices to each sample, because of the local structure often present in a hypergraph. Because of this structure, it is possible that many similar vertices reside on the same processor. These vertices would then not be allowed to all match the same sample, which would result in a bad coarsening.

Contraction phase

After the matching is computed, the clusters are contracted. A new hypergraph $H' = (V', E')$ is created, where V' contains the samples and unmatched vertices and E' all hyperedges containing one of these vertices. Every local sample is contracted with its matches into a new local vertex. For this contraction, the weights of the vertices and the information on its nets are needed. For each sample, every processor computes the sum of the weights of the local vertices matched to the sample. The union of the nets excluding the nets of the sample is also computed. We do not include the nets of the sample in this set, because the owner of the sample already has the information on these nets. This information is sent to the owner of the sample, that can now contract the sample. The owner also stores the ids and owners of the vertices matched to the sample, so it can send the relevant information when uncoarsening.

Sample selection

For our scheme, the selection of the samples is very important. Selecting similar samples will result in bad clusterings. For that reason, we are interested in ways to select these samples such that there is enough variety in the samples. We only consider this variety locally and presume this will give enough variation globally. The algorithm for selecting samples should be very fast, otherwise a parallel global matching would be preferable.

In PMondriaan, two ways of selecting samples are included; one selects the samples uniformly at random, the other uses a label propagation step. As we have seen in Section 1.2.3, label propagation is a very fast algorithm to find clusters in a hypergraph. We use this algorithm to find similar vertices in the hypergraph, and then select vertices from different labels, representing clusters in the hypergraph. This results in the following two steps:

Algorithm 2.4.2 Parallel coarsening for processor $P(s)$, with $0 \leq s < p$

Input: Hypergraph $H = (V, E)$ distributed over p processors, maximum cluster size $c_{\max}$.
Output: Coarsened hypergraph H'.

```
function Coarsen(H)
    S_s := SelectSamples(H_s, ℓ/p)                          ▷ ℓ is the sample size
    for t := 0 to p − 1 do
        Send S_s to P(t)
    ip := InnerProduct(H_s, S)
    for all v ∈ V_s \ S_s do
        match_v := argmax (ip_v(u) : 0 ≤ u < ℓ)
        Add (v, ip_v(match_v)) to r_{match_v}               ▷ Add v to the request list of its match
    for i := 0 to ℓ − 1 do
        t := ⌊i · p/ℓ⌋                                      ▷ The owner of the sample
        Send the best c_max matches (i, v_j, ip_{v_j}(i)) in r_i to P(t)   ▷ Request matches (i, v_j)
    for i := s · ℓ/p to (s + 1) · ℓ/p − 1 do                ▷ All local samples
        for the best c_max received requests (i, v_j) do
            t := owner of v_j                               ▷ The owner of the request
            Add v_j to cluster of i
            Send (i, v_j) to P(t)                           ▷ Accept the request from P(t)
    for i := 0 to ℓ − 1 do
        w_i := 0                                            ▷ Total weight of the cluster
        I_i := ∅                                            ▷ Union of the hyperedges of the cluster
        for all accepted matches (i, v_j) do
            w_i ← w_i + w(v_j)
            I_i ← I_i ∪ I(v_j)
        t := ⌊i · p/ℓ⌋                                      ▷ The owner of the sample
        Send w_i and I_i to P(t)

    V'_s := S_s                                             ▷ Contraction stage
    for all unmatched v ∈ V_s do
        V'_s ← V'_s ∪ {v}
    Create new hypergraph H'_s = (V'_s, E'_s)
    for all received w_i, I_i do
        w(i) ← w(i) + w_i
        I(i) ← I(i) ∪ I_i
    Remove free nets from H'
    Merge free vertices of H'                               ▷ Explained in Section 2.4.3
    return H'
```

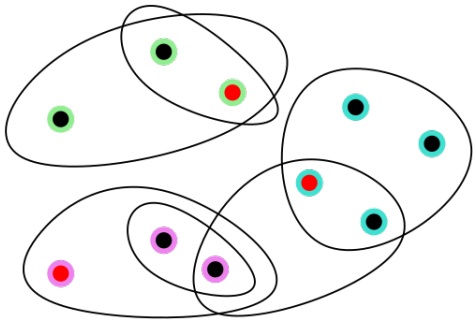


Figure 2.3: The selection of samples using label propagation. To select three samples, we first do a label propagation with three labels. The resulting labeling is shown by the green, purple, and blue circles around the vertices. From each group a sample is selected, depicted by the red vertices.

1. Use label propagation to give each vertex a label in $\{0, 1, ..., \ell/p - 1\}$.
2. For each label, select one vertex with that label uniformly at random to create a list of ℓ/p samples.

This process is shown in Figure 2.3. The results of the two methods are compared in Section 4.1.

Stopping criterion

We keep the sample size constant at all coarsening levels. Because of this, the fraction of vertices that is chosen as a sample will increase. The samples cannot match each other, so at some point only a small fraction of the vertices will get matched. To avoid this, we stop coarsening in parallel when there are less than $q\ell$ vertices left, where q is a constant. We set $q = 3$ in PMondriaan by default. We have found that for this value, the hypergraph is small enough to be communicated, and the parallel coarsening still works well.

Alternatives

Different variations of this coarsening scheme are imaginable. For example, we could also allow vertices to match with other local vertices. Another possible variation is to only allow matches with inner product greater than some minimum value, instead of restricting the cluster size. In future work it would be interesting to explore these variations and determine the best method.

2.4.3 Merging free vertices

At the end of each coarsening level, we remove all hyperedges e for which $|e| \leq 1$. We call these *free* hyperedges, as they cannot contribute to the cut size. After removing these hyperedges, we can easily identify all *free* vertices, vertices only contained in free nets, as vertices with degree zero. These vertices can be assigned to any part, without changing the cut size. Therefore, we can later use them to improve the balance of the partitioning that is found. To make sure they are not assigned to a part before, we remove them from the

hypergraph. The vertices that are removed are stored as a separate set in the contraction, so we can add them to the hypergraph again during the uncoarsening.

Adjusting the maximum weights

During the multilevel bisection, we have to make sure the partitioning adheres to the given maximum weight for both parts. When removing free vertices from the hypergraph, the total weight of the hypergraph decreases. However, the maximum weight of both parts does not change, which gives more freedom when partitioning the vertices. *Claim*: we can always make a partitioning this way that still adheres to the maximum weights, when adding the free vertices.

Proof. We denote by $W_{0,\max}$ and $W_{1,\max}$ the maximum weights of the parts. By W we denote the total weight of the hypergraph before removing free vertices. Suppose we have a set of free vertices F with total weight f. By W_0, W_1 we denote the weights of the parts after the partitioning. We know that

$$W = f + W_0 + W_1 \leq W_{0,\max} + W_{1,\max}, \tag{2.3}$$

otherwise there would be no feasible solution. We also have $W_0 \leq W_{0,\max}$ and $W_1 \leq W_{1,\max}$. We now need to bipartition F into sets with weight f_0 and f_1, such that $W_0 + f_0 \leq W_{0,\max}$ and $W_1 + f_1 \leq W_{1,\max}$. We assume that for any $f_0 \in \{0, ..., f\}$, there exists a bipartitioning of F with weight of part 0 equal to f_0. We take $f_0 = \min(f, W_{0,\max} - W_0)$ and $f_1 = f - f_0$. It follows that $W_0 + f_0 \leq W_{0,\max}$. When $f_0 = f$, clearly $W_1 + f_1 = W_1 + 0 \leq W_{1,\max}$. When $f_0 \neq f$, we get $W_1 + f_1 = W_1 + f - W_{0,\max} + W_0$. Using Equation 2.3 we find $W_1 + f_1 \leq W_{1,\max}$, showing that we do not exceed that maximum weight, which proves our claim. □

The assumption that we can bipartition F as in the proof, does not always hold in practice. In PMondriaan, a different method of bisecting the free vertices is used from the one in the proof. In practice, this method always succeeds in finding an assignment of the free vertices that adheres to the balance constraint. In Section 2.6.3, we discuss our algorithm for assigning the free vertices.

2.5 Initial partitioning

As described before, each processor has its own coarsened hypergraph at the start of the initial partitioning phase. So, the initial partitioning phase can be performed locally. We run a biparitioning algorithm on the hypergraph ten times, each time starting from a different randomly initialized labeling. The best partitioning of the ten found is chosen. Because the hypergraph is small, partitioning the hypergraph multiple times does not have a large influence on the overall running time. To partition the hypergraph, we first bisect the hypergraph using label propagation. We subsequently improve this bipartitioning using the FM heuristic, see Section 1.2.1.

2.6 Uncoarsening

In this section, we describe the uncoarsening scheme used in PMondriaan. We first give an overview of the uncoarsening scheme, which is similar for the sequential and parallel

uncoarsening. Next, we give a more detailed description of the sequential and parallel refinement schemes. Finally, we discuss the algorithms used for the assignment of free vertices.

The uncoarsening of a hypergraph H' to H consists of three stages. In the first stage, we assign all vertices of H to the same part as the vertex of H' they were contracted into. In the parallel case, communication is needed here to assign the contracted vertices to the correct part. In the second stage, we improve the partitioning of H using an iterative refinement algorithm. The iterative refinement algorithms we use, are based on the FM heuristic [20]. In the final stage, we assign all free vertices of H to a part.

2.6.1 Sequential refinement

The sequential refinement scheme uses the FM heuristic [20]. A general explanation of this scheme can be found in Section 1.2.1. Here, we will discuss some of the definitions and data structures in more detail, because they are also needed in our new parallel refinement scheme.

As described earlier, the FM heuristic proceeds in passes. In each pass, each vertex v is moved to the complementary part and locked, in an order based on their gain value $gain_v$. We now explain how to update these gain values efficiently after moving a vertex. To do this, we need a vector C, containing the number of vertices in part 0 and 1 for each net. By $C_0(n)$ we denote the number of vertices in part 0 in net n. By $C_1(n)$, we denote the same for part 1. We call a net *critical*, if moving a vertex in the net can change the net from cut to uncut, or the other way around. A net n is critical if and only if $C_0(n)$ or $C_1(n)$ is 0 or 1. Only critical nets influence the gain value of a vertex. We therefore only need to update the gain values of free vertices that share a net with the moved vertex that is critical before or after the move. Algorithm 2.6.1 shows how to update all gain values after moving a vertex.

Algorithm 2.6.1 Update of the gain values for the move of vertex v.

Input: Vertex v that is moved from part F to part T.

```
function UpdateGains(v)
    for all e ∈ I(v) do
        if C_T(e) = 0 then
            gain_u ← gain_u + c(e) for all free u ∈ e
        else if C_T(e) = 1 then
            gain_u ← gain_u − c(e) for the only u ∈ e with L(u) = T if u is free
        C_F(e) ← C_F(e) − 1
        C_T(e) ← C_T(e) + 1
        if C_F(e) = 0 then
            gain_u ← gain_u − c(e) for all free u ∈ e
        else if C_F(e) = 1 then
            gain_u ← gain_u + c(e) for the only u ∈ e with L(u) = F if u is free
```

To find the only vertex in a net in part 0 or 1 efficiently, we sort the vertices in each net so that all vertices in part 0 precede those in part 1. When moving a vertex, we make sure this order is preserved. This way, we can easily find the desired vertex as the first or last vertex in a net.

2.6.2 Parallel refinement

Just like the sequential algorithm, the parallel refinement algorithm proceeds in passes. In each pass, we try to move all vertices to the other part, in an order based on their gain values. However, to update the gain values after moving a vertex, communication between all processors is needed. Communicating after every vertex move is very expensive, and would result in an almost sequential algorithm with a lot of communication. Therefore, we choose a different way of updating the gain values; instead of synchronizing after every move, we allow the local information to become outdated. This way, we can move a group of vertices together to limit the number of synchronizations.

Using outdated information also has disadvantages. For example, consider a net $\{u, v\}$, where u and v reside on different processors. Suppose $L(u) = 0$ and $L(v) = 1$, then both processors might reassign its local vertex, giving no improvement in the cut size. To ensure this does not happen too often, the number of moves after which we synchronize must be chosen small enough. Moving a group of vertices can also invalidate the balance constraint. To make sure we do adhere to the balance constraint after moving a group of vertices, we let each processor decide redundantly which of the suggested moves can actually take place. We call everything up to the synchronization of C a round. A round consists of the following steps:

1. **Find best moves:** Every processor finds the x best moves based on *outdated* gain values and sends these to the other processors.
2. **Reject unbalanced moves:** Every processor finds the best moves that still adhere to the global balance constraint redundantly. It rejects all other moves.
3. **Reverse rejected moves:** All moves that have been rejected are reversed.
4. **Update counts:** The counts C are updated to the correct values using communication. Also, the correct cut size is computed.
5. **Update gains:** The outdated gains are updated using the new C.

Each pass consists of a number of such rounds. At the end of the pass, we again reverse moves up to the best partitioning found during the pass. Because we only check the cut size after each round, we can only reverse moves in groups; the same groups that were moved in the rounds. To make sure the best partitioning is likely found at the end of a round, we only allow vertices with positive gain to be moved. This diminishes the hill climbing capability of the algorithm, but increases the probability we can reverse to the best solution found during the pass. Because we explore the solution space in a different way than the original FM algorithm, incorporating the hill climbing capability of the algorithm would not have the same effect in our new approach. When multiple processors move vertices with negative gain in a round, most of these moves will not contribute to finding a better partitioning later on. So, even though part of the partitioning might improve because of the negative moves, the overall partitioning will likely still be worse than the best found so far.

We now describe each of these five steps in more detail.

Find best moves

We find the best moves based on the information present at the start of each round. We find these moves in a similar way as in the sequential algorithm. When a vertex is moved,

Owner	Gain	Balance
$P(3)$	7	+3
$P(2)$	5	-2
$P(3)$	4	-4
$P(0)$	4	+1
$P(1)$	3	+5
$P(1)$	2	+2
$P(2)$	0	-1
$P(0)$	0	+1

Table 2.2: Example of rejecting unbalanced moves. Suppose $W_{0,\max} = W_{1,\max} = 20$, $W_{0,\text{prev}} = 17$, and $W_{1,\text{prev}} = 16$. By summing the balance column we find $b = +5$. This gives $W_0 = 17 + 5 > 20$. Until we find a balanced partitioning, we reject the positive moves with least gain, resulting in the rejection of the red moves.

all local information, such as C and the weights of part 0 and 1, is updated. Because other processors are also moving vertices, this information becomes outdated. The more vertices each processor moves, the more outdated this information gets. We therefore only allow processors to move x vertices in each round. We keep track of these moves by storing the vertex index, the gain achieved (which could be based on outdated information) and the difference in the weight balance it gives. The gain and balance values of these moves are sent to all processors, including a tag stating from which processor the move was sent.

Reject unbalanced moves

Each processor now finds the best combination of moves that still adheres to the balance constraint. The other moves are rejected and reversed. First, we add up all balance changes, which are negative when a vertex is moved from part 0 to part 1 and positive when it is moved from 1 to 0. With this total b, we can compute how much weight has to be moved back and from what part. The current weights of the parts are $W_0 = W_{0,\text{prev}} + b$ and $W_1 = W_{1,\text{prev}} - b$. If $W_0 > W_{0,\max}$, we need to move $W_0 - W_{0,\max}$ weight back to part 1. If $W_1 > W_{1,\max}$, we need to move $W_1 - W_{1,\max}$ weight back to part 0. To do this, we sort the received moves based on their gain value. We find the moves that have to be rejected by starting at the worst move, with respect to its gain, and rejecting it if that move is in the right direction. We also do not reject the move, if this would result in too much imbalance in the other direction. Note that such a vertex v would need to span the gap between W_1 and $W_{1,\max}$, so have weight $w(v) > W_{1,\max} - W_1$. Using that $W_{0,\max} + W_{1,\max} = (1+\epsilon)(W_0 + W_1)$ and $W_0 > W_{0,\max}$, we find $w(v) > \epsilon(W_0 + W_1)$, which is highly unlikely to happen. When we encounter such a vertex, we mark its processor as *done*. When a processor is marked as *done*, we are not allowed to move any of its vertices back anymore. We do this until we have reached the weight that had to be reversed. Each processor now knows the moves it has to reverse. An example of this process is shown in Table 2.2.

Reverse rejected moves

All rejected moves are reversed by each processor. The vertices belonging to these moves are moved to the complementary part. These vertices are not considered again in the next round. This way, we still consider each vertex only once in a pass.

	v_0	v_1	v_2	v_3	v_4
$L(v)$ start round	0	0	0	1	1
$L(v)$ end round	1	1	1	1	1

(a) Parts of vertices at the start and end of the round.

	$v \in e$	Start round	Before update	Update	After update
$P(0)$	v_0, v_1	$C(e) = (3, 2)$	$C(e) = (1, 4)$		$C(e) = (0, 5)$
$P(1)$	v_2, v_3	$C(e) = (3, 2)$	$C(e) = (2, 3)$	$C'_0(e) = 3 - 2 - 1 = 0$ $C'_1(e) = \|e\| - C'_0(e) = 5$	$C(e) = (0, 5)$
$P(2)$	v_4	$C(e) = (3, 2)$	$C(e) = (3, 2)$		$C(e) = (0, 5)$

(b) The local values of $C(e)$ at different steps of the round.

Table 2.3: Example of updating $C(e)$ for a net $e = \{v_0, v_1, v_2, v_3, v_4\}$ spread over three processors. The parts of the vertices before and after moving can be found in (a). In (b), the $v \in e$ column shows the local vertices in the net for each processor. The next columns show the local values of $C(e)$ at different times. In the update step, 1 processor computes the correct counts by adding the difference between the local value and the starting value of $C_0(e)$ to the starting value. The new count is sent to the other processors, so they again have the correct local value.

Update counts

To update the vector C, we assign each net to a processor using a block distribution. This way, we can easily find the owner of each net. Each processor is responsible for computing the correct counts C_0 and C_1 of its assigned nets. It first sends its local counts for each net whose counts have changed in this round to the corresponding processor. This processor now has to compute the correct counts for each net. To do this, each processor stores the counts of its assigned nets at the start of each round. Now, it can compute the local change in a net for any processor. Adding up these local changes for a net gives the global change, which we can use to find the new counts. An example of this computation is given in Table 2.3. During this computation, we also keep track of the cut size of the nets assigned to the processor. This way, we can easily compute the total cut size by adding all local cut sizes.

Next, all new values of counts that differ from the counts at the start of the round are sent to all processors. We let each processor store the local counts at the start of each round, so we can easily find the new counts of all local nets C_{new}. For each net e for which no new count was received, $C_{\text{new}}(e)$ is set to the count as it was at the start of the round, otherwise, $C_{\text{new}}(e)$ is set to the received value.

Update gains

The gain values of the vertices need to be updated efficiently. To do this, we compare the local count values to the correct values for each net. We only need to update the gain values for vertices in nets that are critical before or after the update. All other changes in counts of a net do not influence the gain value of its vertices. There are 8 such cases, in which we (possibly) need to adjust the gain of some vertices. We denote by C_{loc} the local (outdated) counts and by C_{new} the updated counts of a net. For each net e there are four cases relating to the counts of part 0, and four similar cases for part 1. The algorithm to update the

outdated gain values is shown in Algorithm 2.6.2.

Algorithm 2.6.2 Update outdated gain values with new C

Input: The local outdated counts C_{loc} and the correct new counts C_{new}.
Output: The gain values are modified in place to reflect the new counts.

```
function UpdateOutdatedGains(C_loc, C_new)
    for all e ∈ E do
        for i ∈ {0, 1} do
            if C_{i,new}(e) = 0 and C_{i,loc}(e) > 0 then                ▷ Case 1
                gain_v ← gain_v − c(e) for all v ∈ e
            else if C_{i,new}(e) = 1 and C_{i,loc}(e) > 1 then           ▷ Case 2
                gain_v ← gain_v + c(e) for the only v ∈ e with L(v) = i
            if C_{i,new}(e) > 0 and C_{i,loc}(e) = 0 then                ▷ Case 3
                gain_v ← gain_v + c(e) for all v ∈ e
            else if C_{i,new}(e) > 1 and C_{i,loc}(e) = 1 then           ▷ Case 4
                gain_v ← gain_v − c(e) for the only v ∈ e with L(v) = i
```

In case 2 and 4, it could be that there is actually no local vertex in the net for which the condition holds. In this case, we do not need to do anything. Because the vertices in a net are sorted, we can check this by looking at the part of the first or last vertex respectively. If it is equal to the part we are looking for, we have found the vertex to update, otherwise, there is no local vertex in that part.

2.6.3 Assigning free vertices

After each uncoarsening step, each of the free vertices is assigned to a part. We first discuss how we assign free vertices in the sequential case, and then explain how we use a similar method in the parallel case.

The algorithm to assign a free vertex to a part is shown in Algorithm 2.6.3. To assign a vertex, we first check whether assigning it to one of the parts would violate the balance constraint. If this is the case, the vertex is assigned to the complementary part. If not, the vertex is assigned to the part that has the smallest ratio between its current and maximum weight. By considering the ratio instead of the largest difference between the current and maximum weight, we make sure the split comes closer to the desired ratio. Finally, if these ratios are equal, the vertex is randomly assigned to a part.

In the case where we have a distributed hypergraph, we use a similar approach. We start by computing the optimal weight to assign to part 0: f_0^*. This is the free weight we assign to part 0 such that if $f_0 = f_0^*$, then $W_{0,\text{new}}/W_{0,\max} \approx W_{1,\text{new}}/W_{1,\max}$. Let f be the global free weight and f_0, f_1 the free weights assigned to part 0 and 1. Then, $W_{0,\text{new}} = W_0 + f_0$ and $W_{1,\text{new}} = W_1 + f - f_0$. Solving $W_{0,\text{new}}/W_{0,\max} = W_{1,\text{new}}/W_{1,\max}$ for f_0 using these equations, we find the following expression for f_0^*:

$$f_0^* = \frac{W_1 W_{0,\max} - W_0 W_{1,\max} + f W_{0,\max}}{W_{0,\max} + W_{1,\max}}. \tag{2.4}$$

We compute f by communicating the local free weights. To assign the free weight, we go through all processors and assign its entire free weight to part zero, until this would assign more weight to part 0 than f_0^*. Suppose this happens for processor $P(t)$, then for all

Algorithm 2.6.3 Assign a free vertex to a part

Input: Free vertex v, weight of the parts W_0, W_1, maximum weight of the parts $W_{0,\max}, W_{1,\max}$.
Output: The part $L(v)$ that v is assigned to.

```
function AssignFreeVertex(v)
    if W_0 + w(v) > W_{0,max} then
        L(v) ← 1
    else if W_1 + w(v) > W_{1,max} then
        L(v) ← 0
    else if W_0 W_{1,max} < W_1 W_{0,max} then
        L(v) ← 0
    else if W_0 W_{1,max} > W_1 W_{0,max} then
        L(v) ← 1
    else
        L(v) ← Random(0,1)
    return L(v)
```

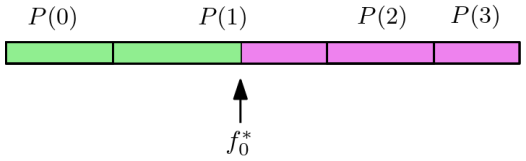


Figure 2.4: Parallel assignment of free vertices using four processors. The green weight is assigned to part 0, the purple to part 1.

$0 \leq i < t$ all free vertices of $P(i)$ are assigned to part 0, and for $t < j < p$ all free vertices of $P(j)$ are assigned to part 1. An example of this procedure is given in Figure 2.4. $P(t)$ can now compute W_0 and W_1 after this assignment. With these adjusted weights, it uses the sequential algorithm to assign its vertices. This can give a slightly different weight to part 0 than f_0^*, so finally $P(t)$ sends the new W_0 and W_1 to all other processors.

3 Implementation

We have implemented the algorithm of Chapter 2 in a new parallel hypergraph partitioner called PMondriaan[1]. It can partition a hypergraph into k parts using p processors. Hypergraph partitioning is the engine of the Mondriaan sparse matrix partitioner [4]. Therefore, a parallel 2D sparse matrix partitioner can be built that uses the PMondriaan hypergraph partitioner, for example by using the medium-grain method, hence the name PMondriaan. Over the years, many options were added to Mondriaan to investigate their benefits. For PMondriaan, we integrated the options that have proven to work best. This way, we build on the concepts used in the Mondriaan partitioner.

PMondriaan was created in the C++ programming language, using the Bulk software library [30]. Bulk is an interface for writing BSP programs in the C++ language. Using Bulk, we were able to write PMondriaan for distributed-memory, shared-memory, and hybrid systems simultaneously. In this section we will discuss some importantant implementation details. First, in Section 3.1, we highlight important aspects for the use of PMondriaan, such as the options included in the partitioner. In Section 3.2, we discuss some important (new) features of Bulk used by PMondriaan. Finally, in Section 3.3, we analyze the memory scalability of PMondriaan. Later, in Section 4.4, we validate this using the results found experimentally.

In this section, we will use a `monospace` font to indicate C++ code and symbols.

3.1 Use of PMondriaan

As discussed before, PMondriaan works for distributed-memory, shared-memory, and hybrid systems. Building the PMondriaan library creates both a thread version, for shared-memory systems, and an MPI version, for distributed-memory and hybrid systems, of the PMondriaan program as `Run_PMondriaan_thread` and `Run_PMondriaan_mpi`. Of course, the MPI version is only created if MPI is available on the system.

PMondriaan is designed to be able to handle very large hypergraphs. Therefore, all data is stored using `long` or `size_t`, which are implemented on most recent systems as 64-bit (un)signed integers.

3.1.1 Input

A number of options and parameters are available to run PMondriaan. These can be set in the `defaults.toml` file, or passed as command line arguments in the program call. Input

[1] Source code available at `https://github.com/SdeBerg/PMondriaan`

Option	Possible values
Metric	`cutnet` `lambda_minus_one`*
Bipartitioning	`random` `multilevel`*
Sampling method	`random`* `label_propagation`
Vertex weight	`one` `degree`*

Table 3.1: Program options included in PMondriaan. The vertex weight option either sets all vertex weights to one, or to their degree. The options with a * are the default options.

given via the command line overrules the settings in the defaults file. The program options can be found in Table 3.1. Two possible metrics are included in PMondriaan: the hyperedge-cut metric and the $(\lambda - 1)$-cut metric. When the hyperedge-cut metric is used, a global communication step is added after each split, where information on the cut nets is sent to all other processors, and these nets are removed from the hypergraph. The random bipartitioning option is only meant for debugging. It randomly assigns vertices to the parts under the given balance constraint. As seen in Table 3.1, there are only two options to set the vertex weights: all vertex weights are set to 1 or to their degree. All net costs are always set to one. However, PMondriaan can handle any integer weights and costs. In a future version, use of custom weights and costs could therefore be implemented easily.

Table 3.2 shows the parameters needed to run PMondriaan and their default values. The sample size and maximum cluster size need to be adjusted to the matrix at hand, because the values of these parameters greatly influence the coarsening phase. Choosing these values too small will result in a slow coarsening process. Choosing the sample size too large will slow down the program and start the sequential coarsening too early. Choosing the cluster size too large will result in a bad coarsening. A sample size of 1-2% of the vertices in combination with a cluster size between 20 and 50 usually works well. In Section 4.1 elaborate on this by performing a number of experiments.

Input parameter	Default	Description
`p`	-	Number of processors used
`k`	-	Number of parts to split into
`file`	-	Path to file of hypergraph in MatrixMarket format
`eps`	0.03	Balance constraint of the partitioning
`eta`	0.03	Balance constraint during computation
`sample_size`	10000	Global sample size
`max_cluster_size`	50	Maximum cluster size
`lp_max_iter`	25	Maximum number of iterations in label propagation
`coarsening_nrvertices`	200	Number of vertices at which we stop coarsening
`coarsening_max_rounds`	128	Maximum number of coarsening rounds
`KLFM_max_passes`	25	Maximum number of passes in FM
`KLFM_max_no_gain_moves`	200	Maximum number of successive no-gain moves in sequential FM
`KLFM_par_send_moves`	20	Number of moves sent before synchronization in parallel FM

Table 3.2: The parameters needed to run PMondriaan. These can be defined in the defaults file or passed as command line arguments to the program. The variables that do not have a default value always need to be adapted for the specific problem.

3.1.2 Output

After partitioning the hypergraph, the resulting partitioning is written to a file and some information on the partitioning is printed. The distributed hypergraph graph is stored using an adapted Matrix Market format, similar to the one used in Mondriaan [4]. The structure of the format is as follows:

```
%%MatrixMarket distributed-matrix coordinate pattern general
m n nnz k
Kstart[0] (this should be 0)
  ...
...
...
Kstart[k] (this should be nnz)
A.i[0] A.j[0]
...
...
...
A.i[nnz-1] A.j[nnz-1]
```

Here, `Kstart` contains the start indices of the parts. For example, the first `Kstart[1]`

nonzeros are assigned to part 0. Contrary to the format used in Mondriaan, we do not include the values of the nonzeros. These values are not used during computation and are therefore not stored when reading the original matrix. Because our format is similar to that used by Mondriaan, the tool `MondriaanStats` can also be used for the analysis of the results produced by our partitioner. The cut size and load balance of the partitioning are also printed at the end of the run of PMondriaan, but can later always be found using `MondriaanStats`.

3.2 Bulk

Bulk [30] is an interface for writing BSP programs in the C++ language. It uses features of C++17 for safe memory use and more code reuse. To write programs for distributed-memory, shared-memory and hybrid systems simultaneously, Bulk can use both a thread and an MPI backend. Because Bulk is a relatively new library, some new features were included in Bulk 2.0.0 for the development of PMondriaan. Changes were also contracted to Bulk upstream. In this section, we focus on the aspects of Bulk that are relevant to PMondriaan and highlight the newly added features. In particular, we discuss the methods of communication between processors.

At the start of the SPMD section, a `bulk::world` object is obtained, that can be used to communicate with the other processors. Within this world, each processor can be identified by its `rank`, similar to our s. After the first bisection in our partitioner, the processors are split into two groups. In the subsequent bisections, only communication within the subgroup of processors is needed. To facilitate this, a new feature was introduced in Bulk: a function to *split* a `world` into a number of new (smaller) `world`s. The `world::split` function takes the group number of the processor as input and returns a new `world` including all processors in the same group. The `rank` of a processor in such a new `world` is based on the order of the processors in the old `world`. Because of this new feature, we were able to write the code for the bipartitioning in a very general way with a `bulk::world` object as input.

Communication methods

There are a number of ways to communicate with other processors through a `bulk::world`. The most used method of communication in PMondriaan is by using a `bulk::queue`. This is used to send a message to a specified processor. After creating a `queue`, a message sent in that `queue` must always contain the same number of arguments. The template parameters of the `queue` specify the types of these arguments. For example, a `queue` is used in PMondriaan to send the best partitioning found after the sequential uncoarsening phase to all processors. In this case, we need to send the index and the label of each vertex to the other processors, so we create a `bulk::queue<long, long>`.

Another way of communicating is using a `bulk::var`, which is a distributed object with an image on each processor. The image is readable and writable from remote processors. A `var` is, for example, used in PMondriaan to identify the processor that has found the best partitioning after the sequential uncoarsening. Each processor sets its local image of a `var` to the cut value it has found, and finds the best cut by reading all remote values.

One more way of communicating is using a `bulk::coarray`. This models distributed data as a 2-dimensional array, where the first dimension is over the processors, and the second over the local array indices. The local size of the array can differ for each processor.

The local values are again readable and writable from other processors. In PMondriaan a `coarray` is, for example, used to compute the global weights of all parts of the partitioning. A `coarray` of local size k is created and each local value is set to the local weight of that part. By reading the remote values, we can find the global weights of the parts.

Bulk also includes a way to perform operations on a `var` or `coarray`. The `bulk::foldl` function performs a left-fold over a distributed variable using a given function. In Bulk 2.0.0, some standard functions were also added, such as `bulk::sum` and `bulk::max`. To accommodate for the use of a `coarray` as described in the previous paragraph, a new function `bulk::foldl_each` was also included. For each local index, this function performs a left-fold over all remote elements at the same index.

3.3 Memory scalability

In this section we will analyse the memory usage of the different components of PMondriaan. We will express the memory use in terms of the local number of vertices $|V_s|$, hyperedges $|E_s|$, and nonzeros $|nz(A_s)|$, where A_s is the associated matrix of the local hypergraph. Storing for each vertex the hyperedges containing the vertex and for each hyperedge the vertices it contains corresponds to compressed row storage (CRS) and compressed column storage (CCS) of A_s. We will also use $|H_s|$ to denote the size of the local hypergraph, so the memory used to store the local hypergraph. We analyse the memory use of a single processor, because our partitioner was developed for a distributed-memory system.

First, we discuss how the local sizes relate to the size of the entire hypergraph. Because we use a block distribution of the vertices, we have $|V_s| \approx |V|/p$ at the start of the program. However, this does not imply that $|nz(A_s)| \approx |nz(A)|/p$. In practice, most matrices have a structure where the nonzeros are evenly distributed over blocks of columns, so this is not a problem. In future work, this could be improved by redistributing vertices such that $|nz(A_s)| \approx |nz(A)|/p$, by using a variation on Algorithm 2.3.1. After the first bisection, in the case that $w(v) = d(v)$, we do have that $|nz(A_s)| \approx |nz(A)|/p$, as the weight is then evenly spread over the processors. The local number of hyperedges also depends on the structure of the hypergraph. Only nets that contain a local vertex are stored. This could be $|E|$ nets, but in practice this is generally $O(|E|/p)$. So, all of these scale inversely to the number of processors used.

3.3.1 Recursive bipartitioning

After each bipartitioning, only the hypergraph itself, including the labels assigned so far, is stored. Then, the vertices are redistributed, but the total number of vertices does not change. So, we do not use any additional memory. During the bipartitioning, we do allocate additional memory. The first bipartitioning is the largest, and therefore uses the most memory. From now on we will consider only this first bipartitioning.

At each level of the multilevel bipartitioning scheme, a (smaller) hypergraph is stored. The number of vertices should always be at least halved after each coarsening. This can be achieved by choosing the right values for the sample size and the maximum cluster size. If we assume the size is halved at each level, we only need

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... < 2,$$

times $|H_s|$ memory to store all these hypergraphs. Again, the first level is the most memory invasive. In the next sections, we discuss the memory use of the parallel coarsening and uncoarsening phase at the top level.

Coarsening

In the coarsening phase, we first select ℓ/p local samples. In the random sample selection method, no memory is allocated except for storing the ℓ/p samples. In the label propagation sample selection method, storing the counts of vertices in all parts for each net requires $O((\ell/p)|E_s|)$ memory. However, this memory is freed again right after selecting the samples. These samples and their nets are subsequently shared among all processors. For each sample v, we need to store $d(v)$ net indices, requiring $O(\Delta(H)\ell)$ memory in total, where $\Delta(H)$ is the maximum degree. Next, the inner products of the local vertices and the samples are computed. As discussed before, we store only the best match found so far for each local vertex during the inner product computation. It follows that this only uses $O(|V_s|)$ memory.

After computing the matches, we have a few communication steps, where processors send requests, accept requests and send information on the formed clusters. Of these, the first or the last can be the largest communication step. In the first communication step, each processor sends a request for each local vertex to the owner of the sample it wants to match with. Because we limit the number of requests to a sample by a single processor by $c_{\max}$, a processor can receive at most $pc_{\max}$ requests per sample, resulting in $O(c_{\max}p\ell/p) = O(c_{\max}\ell)$ requests in total. However, it is not likely that a processor actually receives so many requests, because the total number of requests is limited by the number of vertices. Therefore, in practice this is likely to decrease when increasing p. In the final communication step, the information on the nets and weight of the accepted matches is grouped by each processor for every sample. This information is then sent to the owner of the sample. It follows that a processor can only receive $O(c_{\max}\Delta(H)\ell/p)$ data. This again scales inversely to p.

Finally, we create the contracted hypergraph. As explained before, the size of this hypergraph is at least a factor 2 smaller than the original hypergraph.

Uncoarsening

In the parallel uncoarsening phase, we find the best moves based on the gain values of the vertices. To update these gains efficiently during the computation, we need to keep track of counts C of vertices in part 0 and 1, for all local nets. We also need to store the counts of the local vertices C_{loc}, to later efficiently update the gains of the vertices after receiving the correct C values. Storing both of these vectors, requires $O(|E_s|)$ memory. To implement the FM heuristic efficiently, we use a gain bucket data structure. The size of this bucket data structure is determined by the maximum possible gain of any vertex $gain_{\max}$. We have that

$$gain_{\max} = \max_{v \in V_s} \left(\sum_{e \in I(v)} c(e) \right). \tag{3.1}$$

The total number of buckets needed is then $2gain_{\max}+1$. Sadly, this does not scale with the number of processors. However, the number of vertices in the buckets does decrease when increasing p, so more of these buckets will be empty. For future work, it could be interesting to consider different types of storage when using a large number of processors.

When updating the vector C at the end of a round, every processor is responsible for $|E|/p$ hyperedges. For each of these hyperedges, the cost and processor indices of all processors that contain a vertex in the hyperedge are stored by the corresponding processor. The values of C at the start of the round are also stored for each of these nets. Because the number of different processors in a net is expected to be small, only $O(|E|/p)$ memory is needed to store this information.

In the update of the vector C, the largest two communication steps of the refinement algorithm take place. In the first communication step, the local values of C for each net with changed counts are sent to the processor responsible for this net. In theory, the local value of C could be changed for each net in a single round, so a processor could receive $(|E|/p)|E_t|$ updates from every processor $P(t)$. Because only a few vertices are moved in each round, this is not a realistic estimate. We can bound the number of updates sent by a processor by $x\Delta(H)$, where x is the number vertices moved by each processor in a round. The maximum number of updates received by a processor is thus bounded by $px\Delta(H)$, which is still not a tight bound. After receiving the updates, each processor computes the updated values of C for each of its corresponding nets and sends the updated (changed) values to all processors that contain vertices in the net. In this second communication step, the updated values are only sent to the relevant processors. It follows that any processor can receive at most 1 update for each local net, so $|E_s|$ updates in total in this step, therefore scaling with p.

4 | Experimental results

We have tested PMondriaan to check the quality of the partitionings produced, the running time, and the memory usage using a number of hypergraphs corresponding to sparse matrices from the SuiteSparse collection [27]. We also included 1 test matrix that was used for testing Mondriaan [4], called `tbdlinux`[1]. Table 4.1 presents the matrices used, ordered based on the number of nonzeros. In the following section, we will refer to a hypergraph as the matrix corresponding to it.

All tests used the MPI backend of Bulk compiled with GCC 7.3.0 and OpenMPI 3.1.4. The test were run on the Dutch national supercomputer Cartesius, located at surfSARA in Amsterdam. Unless otherwise indicated, the experiments were done on thin nodes, that have 24 cores, 64GB memory, and either an E5-2690 v3 CPU with a clock speed of 2.6 GHz, or an E5-2695 v2 CPU with a clock speed of 2.4 GHz. Two experiments used fat nodes, that have 32 cores, 256GB memory, an E5-4650 CPU and a clock speed of 2.7 GHz.

The main results are summarized in Table 4.2. In the next sections, we will discuss these results regarding parameter settings, solution quality, and running time. Finally, we discuss some results regarding the memory usage of the partitioner. In our experiments, we compare our partitioner to the Mondriaan partitioner in the `onedimcol` mode. In this mode, Mondriaan creates a 1D partitioning of the columns of a matrix, corresponding to a partitioning of the vertices of the hypergraph (in the row-net model).

[1] Available at `https://www.staff.science.uu.nl/~bisse101/Mondriaan/`

Matrix	Rows	Columns	Nonzeros
`tbdlinux`	112757	20167	2157675
`Si34H36`	97569	97569	5156379
`marine1`	400320	400320	6226538
`atmosmodd`	1270432	1270432	8814880
`delaunay_n21`	2097152	2097152	12582816
`hugetrace-00000`	4588484	4588484	13758266
`StocF-1465`	1465137	1465137	21005389
`asia_osm`	11950757	11950757	25423206
`CurlCurl_4`	2380515	2380515	26515867
`Emilia_923`	923136	923136	40373538
`HV15R`	2017169	2017169	283073458

Table 4.1: Properties of the test matrices, ordered based on the number on nonzeros.

Matrix	ℓ	%	$c_{\max}$	PMondriaan Cut	PMondriaan Time	Mondriaan Cut	Mondriaan Time
tbdlinux	1500	7.4	20	30478	33.3	23845	14.7
Si34H36	10000	10.2	20	**18199**	39.9	18194	6.1
marine1	5000	1.2	50	2086	26.9	1981	4.7
atmosmodd	25000	2.0	50	**17213**	86.6	17174	9.7
delaunay_n21	50000	2.4	50	2945	138.3	2734	11.5
hugetrace-00000	180000	3.9	150	1509	295.6	1340	18.4
StocF-1465	10000	0.7	50	**4822**	180.0	5668	22.4
asia_osm*	700000	5.9	150	**16**	315.8	149	22.6
CurlCurl_4	40000	1.7	50	**25489**	187.8	26257	32.6
Emilia_923	8000	0.9	50	26802	80.4	25395	23.3
HV15R*	20000	1.0	50	62415	332.6	60293	178.2

Table 4.2: Hyperedge-cut and running time for partitioning test matrices using PMondriaan and Mondriaan for $k = 2$ and $\epsilon = 0.03$. For PMondriaan, we use 32 processors, a global sample size of ℓ, and a maximum cluster size of $c_{\max}$. The % column shows the percentage of vertices that is used as a sample. The other parameters are set to the default values shown in Table 3.2. We use random sampling and set vertex weights to their degree. For PMondriaan, the cut is the minimum cut found in 10 runs and T the average time over these runs. The * indicate the tests that were run on fat nodes of Cartesius. Mondriaan was run in the `onedimcol` mode, for fair comparison to PMondriaan. The cut values for which PMondriaan performs better or at most 1 percent worse than Mondriaan are marked in boldface.

4.1 Parameter tuning

In this section we discuss how different sample sizes ℓ, maximum cluster sizes $c_{\max}$, and sampling methods work together. Table 4.3 shows the hyperedge-cut and running time to bisect `StocF-1465` for various values of ℓ and $c_{\max}$ using different sampling methods. It also includes the size reduction of the hypergraph in the first coarsening level (right column). First, we discuss the difference in performance of the two sampling methods. We expect the label propagation sampling method to work better for smaller sample sizes than random sampling, because the diversity of the samples is more important in this case. However, we see that even for $c_{\max} = 20$, the label propagation method produces a partitioning that has an only slightly better cut size than the random method. We conclude that choosing samples randomly provides enough diversity to create good clusters. Also note that using label propagation greatly influences the running time, slowing the program down by at least a factor of 1.5. Therefore, we only use the random sampling method in further experiments.

Next, we consider the influence of the choice of the sample and cluster size on the coarsening process. Looking at the coarsening ratios in Table 4.3, we see that choosing a small sample size (5000) and large cluster size (100) or a large sample size (15000) and small cluster size (20) only reduces the size of the hypergraph by 20 percent in the first coarsening. This results in a longer running time. The fastest coarsening, and lowest running time, is achieved with a large sample size of 16000 and a medium cluster size of 50. The best solution is found for a sample size of 10000 and a cluster size of 50. In this case, the hypergraph is not reduced by 50 percent in the first coarsening, but in the following coarsening levels,

ℓ	$c_{\max}$	Sampling	Cut	T (in s)	$\lvert V'\rvert/\lvert V\rvert$
16000	50	R	4864	**173**	0.53
		LP	4829	250	0.07
15000	20	R	4931	206	0.78
		LP	4916	336	0.79
10000	50	R	**4729**	180	0.68
		LP	4963	263	0.68
5000	100	R	4851	188	0.79
		LP	6197	280	0.79

Table 4.3: Hyperedge-cut and running time T for 1 run of PMondriaan to bisect the hypergraph corresponding to the `StocF-1465` matrix using different parameters. We alternate between the random (R) and the label propagation (LP) sampling method for different sample sizes, ℓ, and maximum cluster sizes, $c_{\max}$. The $|V'|/|V|$ column shows the ratio between the number of vertices in the original hypergraph and the hypergraph created in the first coarsening level.

the reduction was around 50% at each level. We prioritize solution quality over a small improvement in running time, so we use a sample size of 10000 and a cluster size of 50 for the `StocF-1465` matrix in the experiments.

To provide a more general insight into the choice of these parameters, we show the sample size and the cluster size used in our experiments for all matrices in Table 4.2. The choice for these parameters was based on preliminary experiments. In the table, the sample size is also provided as a percentage of the number of vertices. Note that for most matrices around 1-2% of the vertices was used as a sample. A maximum cluster size of 50 worked well in most cases, only for the two smallest matrices, a cluster size of 20 was used. There are four matrices that use a much larger percentage of vertices as samples: `hugetrace-00000`, `asia_osm`, `tbdlinux`, and `Si34H36`. The first two matrices are both much sparser than the other matrices. Because vertices will only be matched to a sample if they share at least 1 hyperedge, a larger sample size is needed in this case to achieve a good coarsening. The last two matrices, `tbdlinux` and `Si34H36`, have a relatively small number of columns, corresponding to vertices in the hypergraph. Choosing a smaller sample size resulted in partitionings with a much larger cut size for these matrices. This indicates that our parallel partitioning algorithm is less suited to partitioning small matrices.

4.2 Quality of partitionings

In this section, we discuss the results of Table 4.2 regarding the quality of the partitionings produced. We compare the cut size of the best partitioning produced by PMondriaan in 10 runs to the cut size of the partitioning produced by Mondriaan in the `onedimcol` mode. In about half of the experiments, a better or comparable partitioning was produced by PMondriaan, indicated using boldface in the table. The worst partitionings produced by PMondriaan were for `tbdlinux`, 28% increase in cut size, and `hugetrace-00000`, 13% increase in cut size.

To study the effect of using different numbers of processors on the partitioning quality, we ran PMondriaan using $p = 2, 4, 8, 16, 32, 64$ processors to partition `StocF-1465`, `Emilia_923`,

and `tbdlinux` into 2 and 8 parts. The results are shown in Table 4.4. We did 10 runs of PMondriaan for each combination of p and k, and report the best and average cut size found. We see that for $k = 2$, the best and average cut found are consistent for all numbers of processors. The slight variations can be accounted for by the randomness in the algorithm. For $k = 8$, there is more variation in the quality of the solutions. In this case, the best and average cut size actually improves slightly when using more than 8 processors. Surprisingly, while the best cut size for $k = 2$ in 4.4(a) is 18% smaller than the cut size of Mondriaan, for $k = 8$, the best cut size is 23% larger than the one of Mondriaan.

When partitioning `tbdlinux` using 64 processors, an unbalanced partitioning was found in 6 out of 10 runs (up to 3.03% imbalance). The imbalance is likely the result of the large number of rows the matrix has with respect to the number of columns. In the coarsening, this can create vertex cluster of high weight, making it difficult to create a balanced partitioning. This problem could be solved by introducing a weight limit on the clusters that are created. The balanced partitionings that were found for $k = 2$, had a much smaller cut size than those produced using less processors.

We conclude that the number of processors used has only a minimal effect on the quality of the partitionings that are produced.

4.3 Running time

In this section, we discuss the running times achieved by PMondriaan. Table 4.2 shows that, when using 32 processors, the running times of PMondriaan are much larger than those of Mondriaan. For several matrices, the running time is even a factor 10 larger. The smallest relative difference in running time is achieved for the matrix with most nonzeros, `HV15R`.

Table 4.4 shows the running times for a varying number of processors. We first discuss the results for partitioning the matrices for $k = 2$, and then the results for $k = 8$. In 4.4(a), we see a linear decrease in running time up to $p = 32$. In 4.4(b), we only see a linear decrease up to $p = 8$, and in 4.4(c) we do not see a linear decrease at all. This again shows that our parallel partitioner is less suited for partitioning small hypergraphs, as the speedup achieved for these matrices is small.

To study why our partitioner is slower than the PMondriaan partitioner, we timed the different phases of the partitioning process when bipartitioning `Emilia_923`. The results are shown in Table 4.5. Here, we immediately see that the most time is spent in the initial partitioning phase, which should be the shortest phase of the partitioning process. To assert whether this phase is slow because of the label propagation (LP) or the FM step in this phase, we compare the average running times of the LP and FM part of the initial partitioning phase. The running times of these parts of the initial partitioning phase, for a maximum number of iterations in the label propagation algorithm of 25 and 0, are given in Table 4.6. Note that the total time increases when using 0 rounds of label propagation, because the FM algorithm converges much slower in this case. Neither of these two phases therefore seems to be solely responsible for the large running time.

Table 4.7 shows the time spent in the initial partitioning phase for all matrices. We see that these times vary greatly, while the coarsened hypergraphs all have less than 200 vertices at this point. The density d, defined as $d = nz(A)/(mn)$ for an $m \times n$ matrix A, is also included in the table. We note that for the more sparse matrices, relatively less time is spent in the initial partitioning phase. This indicates that this phase is slow because of the large number of hyperedges in the coarsened hypergraph. Most hypergraph partitioners,

k	2			8		
p	Best cut	Avg cut	T (in s)	Best cut	Avg cut	T (in s)
2	4762	4980	2027	-	-	-
4	4784	4934	796	-	-	-
8	4725	4922	411	69689	82451	2403
16	4761	4824	258	60485	72144	985
32	4822	5016	180	60140	70475	454
64	4824	4939	160	-	-	-
Mondriaan	5668		22	48805		53

(a) Results for `StocF-1465`. For $p = 2, 4$ and $k = 8$ the runs exceeded the time limit of 1 hour and 15 minutes. For $p = 64$ and $k = 8$, the program failed because of a segmentation fault.

k	2			8		
p	Best cut	Avg cut	T (in s)	Best cut	Avg cut	T (in s)
2	27222	28783	437	-	-	-
4	27474	30877	224	122631	153555	4269
8	27270	30028	103	160914	172267	1720
16	27048	35583	87	150945	165235	627
32	26802	29971	80	125637	145517	343
64	27048	30869	78	125751	134279	226
Mondriaan	25395		23	94134		65

(b) Results for `Emilia_923`. For $p = 2$ and $k = 8$ the runs exceeded the time limit of 1 hour and 15 minutes.

k	2			8		
p	Best cut	Avg cut	T (in s)	Best cut	Avg cut	T (in s)
2	30220	31415	44	105321	107136	110
4	29654	31124	37	110629	113395	77
8	28090	30634	33	116401	120242	64
16	30457	31051	32	117929	120484	66
32	30478	31085	33	117887	119028	66
64	25139	*25689	39	113717	*114939	80
Mondriaan	23845		15	93912		24

(c) Results for `tbdlinux`. For $p = 64$, both for $k = 2$ and $k = 8$, 6 of the 10 runs produced an unbalanced partitioning. The average is taken over these runs.

Table 4.4: Best cut, average cut, and average running time T for partitioning matrices using PMondriaan over 10 runs for the $(\lambda - 1)$-cut metric. The parameters were set as in Table 4.2. Results for one run of Mondriaan using the `onedimcol` mode and $\epsilon = 0.03$ are included for reference.

Multilevel phase	Time (in s)
Parallel coarsening	1746
Communicating hypergraph	1065
Sequential coarsening	8439
Initial partitioning	**37404**
Sequential uncoarsening	9489
Communicating best L	8883
Parallel uncoarsening	10050

Table 4.5: Running time of the phases of the parallel multilevel framework for bisecting `Emilia_923` using 32 processors. The parameters were set as in Table 4.2. The phase with the longest running time is marked in boldface.

such as PaToH [3], collapse duplicate hyperedges after each coarsening step, to reduce the number of hyperedges. When collapsing two duplicate hyperedges e_1 and e_2, one of the hyperedges is removed and the other is assigned cost $c(e_1) + c(e_2)$. This was not included in PMondriaan yet. Implementing this in PMondriaan will reduce the number of hyperedges in the coarsened hypergraphs. This will presumably greatly decrease the running time in the initial partitioning phase, and also speed up the coarsening and uncoarsening phase.

We briefly discuss the running times of the other phases of the parallel multilevel method. The parallel coarsening using sampling achieves the desired speedup of the coarsening phase. Using the PMondriaan partitioner with only 1 processor, we find that coarsening the hypergraph using 32 processors gives a speedup factor of 161 for the first coarsening level. This superlinear speedup is achieved because of the use of the sampling scheme in the parallel coarsening algorithm, that is not included in the sequential coarsening algorithm yet. The parallel uncoarsening also achieves a significant speedup with respect to the sequential uncoarsening, as the highest levels of the uncoarsening phase are always more expensive. Here, a speedup of a factor 10 was achieved for the longest sequential uncoarsening using $p = 1$ relative to the longest parallel uncoarsening using $p = 32$. Of the two communication steps, communicating the best partitioning found after the sequential coarsening phase has the longest running time. This is surprising, as here only the labels of the vertices are sent to all processor rather than the information on all their nets. In this phase, each processor also checks for every received label if it belongs to a local vertex. To find the local index of a vertex an `std::unordered_map` is used. The lookup of a value in this map could be less efficient when the value is not present, which will be the case for most of the vertices, resulting in a longer running time of this phase. In future work, it could be investigated if searching the label queue to find the label for each local vertex, or requesting the labels of the local vertices, will improve the efficiency.

Finally, we discuss the results of Table 4.4 regarding the running times of the 8-way partitionings. We note that, when using a small number of processors, the running times of the 8-way partitioning are over 10 times longer than those of the 2-way partitioning. In the 8-way partitioning, the second bisection should have a running time of less than half of the time of the first bisection, giving total running time of at most twice the bipartitioning time. To demonstrate the origin of this problem, Table 4.8 shows the running times of the bisection and redistribution part of the recursive k-way algorithm for partitioning `Emilia_923` into 8 parts. We see that the time in every bisection is indeed almost halved compared to the bisection at the previous level. The bottleneck is in the redistribution step of the recursive

LP rounds	Time LP	Time FM
25	2098	819
0	245	4508

Table 4.6: Average running time in seconds of label propagation (LP) and FM refinement in the initial partitioning phase for partitioning `Emilia_923`, using different numbers of iterations in the label propagation step. Except for `lp_max_iter`, the same parameters as in Table 4.2 were used. In the initial partitioning phase, we bipartition the hypergraph 10 times using a label propagation and FM refinement step. In the label propagation phase, a random initial partitioning is first created.

Matrix	T (in s)	$d \cdot 10^5$
`tbdlinux`	8157	94.89
`Si34H36`	10758	54.17
`marine1`	13723	3.89
`atmosmodd`	28246	0.55
`delaunay_n21`	7985	0.29
`hugetrace-00000`	8212	0.07
`StocF-1465`	54566	0.98
`asia_osm`	140	0.02
`CurlCurl_4`	82560	0.47
`Emilia_923`	37405	4.74
`HV15R`	103782	6.96

Table 4.7: Running time T of initial partitioning phase when bipartitioning the test matrices using the same parameters as Table 4.2. The coarsened hypergraphs all have less than 200 vertices in the initial partitioning phase. The d column shows the density of the matrices, defined as $d = nz(A)/(mn)$ for an $m \times n$ matrix A.

Level	T bisect	T redistribute
1	79150	62040
2	45184	58510
3	26107	-

Table 4.8: Time T in seconds of the levels in the recursive bipartitioning into $k = 8$ parts using $p = 32$ processors for `Emilia_923`. The time in the bisection and redistribution phases of processor 0 are given. The parameters were set as in Table 4.2. After the final bipartitioning, the vertices are not redistributed.

bisection. In this step, the hypergraph is redistributed over the processors, which requires communication. When using a small number of processors, the h-relation corresponding to the communication step is very large. For example, when $p = 2$, suppose that half of the local vertices in is part 0 and half in part 1, then this processor sends information on $1/4$ of all vertices in the hypergraph to the other processor. For larger numbers of processors this becomes less apparent, but still forms a bottleneck. In future work, we could explore different ways of redistributing the vertices, or, for example, not redistribute at all.

4.4 Memory usage

To test the memory scalability of our partitioner, we used the `job-statistics` command of Cartesius to measure the memory use per processor during a bipartitioning. Table 4.9 states the memory usage of PMondriaan, using 64 processors, and Mondriaan for the bisection of `HV15R`. The difference between the average and maximum memory use of any processor for PMondriaan is only 5%. From this, we can conclude that the data is distributed well among the processors. The memory use per processor of PMondriaan is a factor 5 lower than that of Mondriaan.

In Figure 4.1, the memory use per processor to bisect `Emilia_923` is plotted over a range of processors used. For $p = 2$, the memory per processor of PMondriaan is greater than that of Mondriaan. This implies that the either the memory used to store the hypergraph itself is larger for PMondriaan, or that the (un)coarsening algorithms require more memory. However, we see that when using more than 4 processors, PMondriaan uses less memory than Mondriaan. The memory use always decreases when increasing p. The improvement in memory use gets smaller when using more processors. This can be explained by the fact that more redundant information is stored at this point. For example, nets can be split over more processors, requiring multiple processors to store the indices and costs of these nets. It could also imply that the algorithms used are less memory efficient when using a large number of processors. We do note that reduction in memory use is larger for `HV15R`, which is the bigger matrix, implying that the memory use is better scalable for larger matrices.

Note that, for a shared-memory system the available memory per processor scales as $1/p$. This means that, for example for a Cartesius fat node, the algorithm does not achieve the desired scalablibity when using $p = 32$ processors on a single node. However, we can use multiple nodes with less processes running on each node to exploit the memory-scalability that is achieved for distributed-memory systems.

In future work, the memory use of the algorithms could be researched in more elaborate experiments to improve the scalability of the algorithms. Implementing the collapsing of

	Memory (in GB)
PMondriaan avg	3.8
PMondriaan max	4.0
Mondriaan	19.8

Table 4.9: Memory use per processor during partitioning of `HV15R` using the PMondriaan and Mondriaan partitioners. PMondriaan was run on $p = 64$ processors, using the same parameters as Table 4.2. For PMondriaan, the average and maximum memory use over the processors is given.

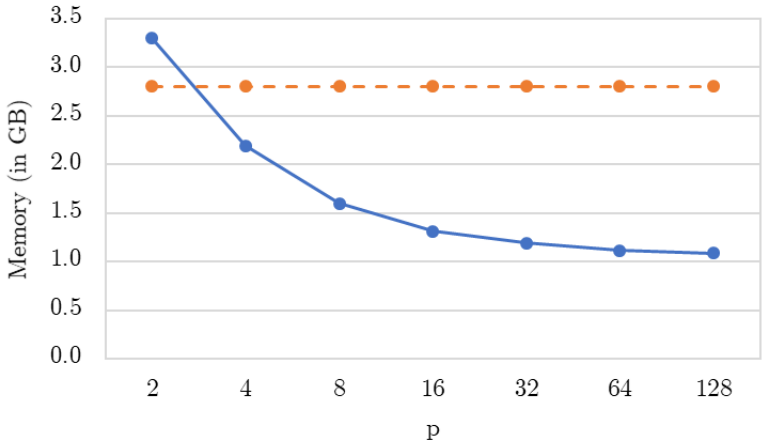


Figure 4.1: Memory usage per processor when bipartitioning `Emilia_923`. The blue line indicates the memory usage of PMondriaan per processor using p processors. The orange dotted line shows the memory use of Mondriaan.

duplicate hyperedges will also further decrease the memory use.

5 | Conclusion

In this thesis, we presented a novel parallel hypergraph partitioning algorithm. The algorithm uses a 1D column distribution of the sparse matrix associated with the hypergraph. To create k-way partitionings, it uses the recursive bisection method. We introduced a parallel extension of the successful multilevel framework, and use this as a foundation for our parallel bisection algorithm. This framework consists of a parallel coarsening, a sequential coarsening, an initial partitioning, a sequential uncoarsening, and a parallel uncoarsening phase. Both the parallel and sequential coarsening methods are clustering-based. The parallel coarsening algorithm is based on a sampling scheme, where only a select group of samples is considered for matching, to reduce the communication. In the uncoarsening phase, the partitioning is refined using the FM heuristic. For the parallel uncoarsening phase, a parallel version of the heuristic is introduced, where we limit the number of synchronizations by moving groups of vertices together.

The algorithms were implemented in a parallel hypergraph partitioner, named PMondriaan, that can partition a hypergraph into k parts using p processors. PMondriaan was created in the C++ programming language, using the Bulk software library [30]. Using Bulk, we were able to write PMondriaan for distributed-memory, shared-memory, and hybrid systems simultaneously.

We provided experimental results of PMondriaan to check the quality of the partitionings produced, the running time, and the memory usage by comparing these to the Mondriaan partitioner [4]. The bipartitionings produced by PMondriaan were of comparable or better quality to those of Mondriaan for 5 out of 11 matrices. For only two matrices the cut size produced by PMondriaan was more than 10 percent higher than those of Mondriaan. The quality of the 8-way partitionings created for three matrices was always at least 10% worse than those of Mondriaan. The quality of the partitionings remained stable when increasing the number of processors used. The running time decreased linearly with the number of processors used. In the coarsening phase, a superlinear speedup was achieved, due to the sampling scheme. However, the running times of PMondriaan were still about 5 times as large as those of PMondriaan, when using 32 processors. The bottleneck here is the initial partitioning phase. For 8-way partitioning, the communication in the redistribution step after each bipartitioning significantly increases the running time. The partitioner is memory scalable. It uses only 3.8GB per processor on average to partition a matrix with 283 million nonzeros, while Mondriaan uses 19.9GB.

In conclusion, PMondriaan achieves our goal to develop a parallel hypergraph partitioner for a distributed-memory system that is memory scalable, while providing state-of-the-art solutions. A reasonable speedup is achieved with respect to the single processor running times of PMondriaan, but no speedup is achieved with respect to the Mondriaan partitioner.

5.1 Future work

Much future work remains to be done. Foremost, the running times need to be improved. To achieve this, two important aspects need to be addressed. First, the running time of the initial partitioning phase should be decreased. The number of hyperedges in the coarsened hypergraph can be decreased by collapsing duplicate hyperedges. This would not only speed up the initial partitioning phase, but also the coarsening and uncoarsening phase. It will also reduce the memory usage of the partitioner.

To reduce the cost of communicating the best partitioning after the sequential uncoarsening phase, a different scheme, where the label of a local vertex is requested by a processor, could be considered.

When creating partitionings into $k > 2$ parts, different ways of redistributing the vertices could be considered to decrease the running time of this process. For example, we could implement a method where vertices are not redistributed at all, and all processors work together on both sub-problems.

The sampling scheme significantly speeds up the coarsening phase, while still giving high quality partitionings. This scheme could also be used in a sequential partitioning scheme. Also, different variations of this scheme could be considered, where, for example, local vertices can also form clusters with other local vertices or only matches of a certain quality are accepted.

Bibliography

[1] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: a survey," *Integration*, vol. 19, no. 1-2, pp. 1–81, 1995.

[2] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in VLSI domain," *IEEE Trans. VLSI Syst.*, vol. 7, no. 1, pp. 69–79, 1999.

[3] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 7, pp. 673–693, 1999.

[4] B. Vastenhouw and R. H. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM Review*, vol. 47, no. 1, pp. 67–95, 2005.

[5] 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, Phoenix, AZ, USA, May 19-23, 2014*, pp. 529–539, IEEE Computer Society, 2014.

[6] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," *VLSI Design*, vol. 2000, no. 3, pp. 285–300, 2000.

[7] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. 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* (M. T. Goodrich and M. Mitzenmacher, eds.), pp. 53–67, SIAM, 2016.

[8] Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag, "Engineering a direct k-way hypergraph partitioning algorithm," in *Proceedings of the Nineteenth Workshop on Algorithm Engineering andf Experiments, ALENEX 2017, Barcelona, Spain, January 17-18, 2017.* (S. P. Fekete and V. Ramachandran, eds.), pp. 28–42, SIAM, 2017.

[9] C. J. Alpert, J. Huang, and A. B. Kahng, "Multilevel circuit partitioning," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 17, no. 8, pp. 655–667, 1998.

[10] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar, "UMPa: A multi-objective, multi-level partitioner for communication minimization," in *Graph Partitioning and*

Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings (D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, eds.), vol. 588 of *Contemporary Mathematics*, pp. 53–66, American Mathematical Society, 2012.

[11] C. Aykanat, B. B. Cambazoglu, and B. Uçar, "Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices," *J. Parallel Distrib. Comput.*, vol. 68, no. 5, pp. 609–625, 2008.

[12] R. Bisseling, K. Devine, U. Catalyurek, E. Boman, and R. Heaphy, "Parallel hypergraph partitioning for scientific computing," in *Proceedings 20th International Parallel and Distributed Processing Symposium*, (Los Alamitos, CA, USA), p. 102, IEEE Computer Society, apr 2006.

[13] A. Trifunović and W. J. Knottenbelt, "Parallel multilevel algorithms for hypergraph partitioning," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 563 – 581, 2008.

[14] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[15] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. New York, NY, USA: John Wiley & Sons, Inc., 1990.

[16] T. Bui and C. Jones, "A heuristic for reducing fill-in in sparse matrix factorization," in *Proceedings Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pp. 445–452, SIAM, Philadelphia, PA, 1993.

[17] T. Heuer and S. Schlag, "Improving coarsening schemes for hypergraph partitioning by exploiting community structure," in *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK* (C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, and R. Raman, eds.), vol. 75 of *LIPIcs*, pp. 21:1–21:19, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[18] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "A hypergraph partitioning: Application in VLSI domain," in *Proceedings of the 34st Conference on Design Automation, Anaheim, California, USA, Anaheim Convention Center, June 9-13, 1997.* (E. J. Yoffa, G. D. Micheli, and J. M. Rabaey, eds.), pp. 526–529, ACM Press, 1997.

[19] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Engineering - Selected Results and Surveys* (L. Kliemann and P. Sanders, eds.), vol. 9220 of *Lecture Notes in Computer Science*, pp. 117–158, 2016.

[20] 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, Las Vegas, Nevada, USA, June 14-16, 1982* (J. S. Crabbe, C. E. Radke, and H. Ofek, eds.), pp. 175–181, ACM/IEEE, 1982.

[21] G. M. Slota, K. Madduri, and S. Rajamanickam, "Pulp: Scalable multi-objective multi-constraint partitioning for small-world networks," in *2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014* (J. J.

Lin, J. Pei, X. Hu, W. Chang, R. Nambiar, C. C. Aggarwal, N. Cercone, V. G. Honavar, J. Huan, B. Mobasher, and S. Pyne, eds.), pp. 481–490, IEEE Computer Society, 2014.

[22] J.-W. Buurlage, "Self-improving sparse matrix partitioning and bulk-synchronous pseudo-streaming," Master's thesis, Utrecht University, 2016.

[23] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014* (I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, eds.), pp. 568–579, IEEE Computer Society, 2014.

[24] H. Meyerhenke, P. Sanders, and C. Schulz, "Partitioning (hierarchically clustered) complex networks via size-constrained graph clustering," *J. Heuristics*, vol. 22, no. 5, pp. 759–782, 2016.

[25] G. M. Slota, S. Rajamanickam, K. D. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*, pp. 646–655, IEEE Computer Society, 2017.

[26] C. J. Alpert, "The ISPD98 circuit benchmark suite," in *Proceedings of the 1998 International Symposium on Physical Design, ISPD 1998, Monterey, CA, USA, April 6-8, 1998* (M. Sarrafzadeh, ed.), pp. 80–85, ACM, 1998.

[27] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

[28] A. Belov, D. Diepold, M. Heule, and M. Järvisalo, "SAT competition 2014." `http://www.satcompetition.org/2014/`. Accessed: 03-10-2019.

[29] T. van Leeuwen, "Expanding Mondriaan's palette," Master's thesis, Utrecht University, 2005.

[30] J. Buurlage, T. Bannink, and R. H. Bisseling, "Bulk: A modern C++ interface for bulk-synchronous parallel programs," in *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings* (M. Aldinucci, L. Padovani, and M. Torquati, eds.), vol. 11014 of *Lecture Notes in Computer Science*, pp. 519–532, Springer, 2018.