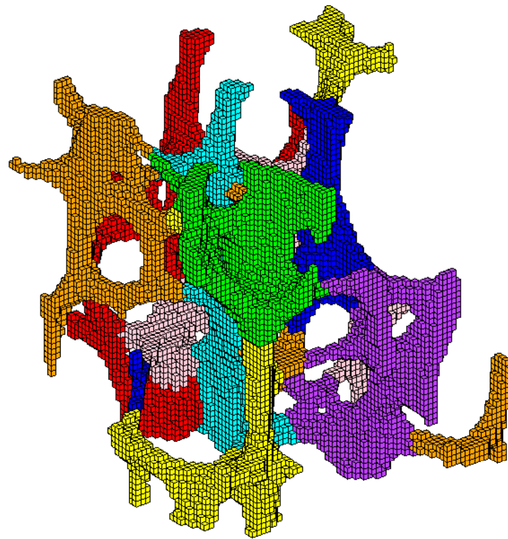
Partitioning of Domains Embedded in a Regular Grid

Author:
Nick Verheul

Supervisor:
Prof. dr. Rob H. Bisseling
Second Reader:
Dr. Paul A. Zegeling



M.Sc. Thesis

February 28, 2013

Universiteit Utrecht

Abstract

In this thesis we present a partitioning algorithm aimed at partitioning domains embedded in a regular grid. Inspired by the multilevel philosophy of the Mondriaan algorithm, we manage to improve Mondriaan's run time up to a factor of approximately 2 for certain test cases, while also constructing a better quality partitioning. While we significantly optimize both the coarsening and the initial partitioning phase of said multilevel method, we do not see comparable improvements in the uncoarsening phase. This means that the overhead of the general partitioner Mondriaan for a grid-embedded domain is limited.

Acknowledgments

I would like to give my thanks to the following people, who all allowed me to be able to write this thesis: Rob Bisseling, for his guidance on and surrounding this thesis, Peter Arbenz, for providing suggestions and the practical application which inspired the work in this thesis, and Thijs Alkemade, for the useful talks we had. Finally, I would like to thank my parents Marco and Mieke for their ongoing support and understanding.

Contents

1 Introduction **3**
1.1 Communication Costs . . . 5
1.2 Load Balance . . . 6
1.3 RCB . . . 7
1.4 Mondriaan . . . 9
1.5 Goals . . . 11

2 Methods **13**
2.1 Coarsening . . . 13
2.1.1 Definition of cluster structure . . . 13
2.1.2 Merge operation . . . 14
2.1.3 Merge cycle . . . 17
2.2 Initial partitioning . . . 18
2.2.1 Karmarkar–Karp . . . 19
2.2.2 Expanding neighbours . . . 19
2.2.3 Net split . . . 21
2.3 KLFM and uncoarsening . . . 21
2.3.1 Kernighan–Lin . . . 21
2.3.2 Fiduccia-Mattheyses . . . 22
2.3.3 Uncoarsening . . . 24

3 Results **26**
3.1 Coarsening . . . 26
3.1.1 Threshold for terminating the merge process . . . 26
3.2 Initial Partitioning . . . 29
3.3 Uncoarsening and the algorithm as a whole . . . 32

4 Conclusion **36**
4.1 Future Work . . . 37

1 Introduction

It is often the case in computational science that computational domains are generated by imaging devices such as CT scanners. These imaging devices scan three-dimensional objects layer-wise and stack the two-dimensional images (layers) to create a three-dimensional representation of the structure. The resulting images can be viewed as domains consisting of *voxels*, i.e. three-dimensional pixels. Since all voxels have the same shape and size, this domain can be considered to be *embedded in a regular grid*.

Recent advances in imaging capabilities have provided new ideas and technologies in the field of *osteology*, the study of bones. In particular, these technological advancements significantly aid the analysis and diagnosis of various bone diseases, e.g. *osteoporosis*. High resolution imaging devices construct extremely detailed three-dimensional models of bone structures. The voxel mapping of the bone structure can be interpreted as a three-dimensional bitmap, each voxel storing whether bone is present at its location. An example of such a high resolution image is shown in Figure 1.1. An important factor for assessing bone strength is bone density, which can be easily estimated from a voxel representation of the bone structure. However, it has been shown that assessing fracture risk based on bone density alone can prove to be very inaccurate [12]. Other factors that contribute significantly to bone strength, such as *bone geometry* and *bone microarchitecture*, are not incorporated in the bone density estimate.

By combining the voxel domains and finite element modelling one can simulate the behaviour of bone structures when subjected to external stress; these simulations can lead to accurate estimations of the bone strength and fracture risk by using *linear elasticity theory*. In order to accurately model the bone microarchitecture, the resolution of the voxel models will be extremely high, leading to millions or even billions of voxels in the domain, e.g. a $1024 \times 1024 \times 1024$ domain. Usually, the voxel model is translated to a finite element model with a direct voxel-to-element transformation, resulting in a finite element model with billions of degrees of freedom.

The discretization of the linear elasticity equation leads to a very large, sparse and symmetric linear system [1]

$$Au = f. \tag{1.1}$$

A good candidate for solving this linear system is the *preconditioned conjugate gradient (PCG)* algorithm. PCG works similar to the standard *conjugate gradient (CG)* algorithm, as proposed by Hestenes and Stiefel in [9], but uses a preconditioning matrix B to improve the spectral properties of the operating matrix and thus the convergence rate of CG, for a more detailed exposition see e.g. [2, 8]. The specific PCG implementation is not appropriate for the goals of this thesis, more details regarding this implementation and the bone strength analysis can be found in [3].

The key operation in CG is a matrix-vector product with matrix A that occurs at every step in the algorithm; because the matrix A is both very large and sparse, performing parallel matrix-vector multiplication is an obvious solution to keep the simulations scalable. Of great importance in parallel matrix-vector multiplication is the partitioning algorithm used to distribute the matrix and vector elements to the available processors. The quality of such a partitioning

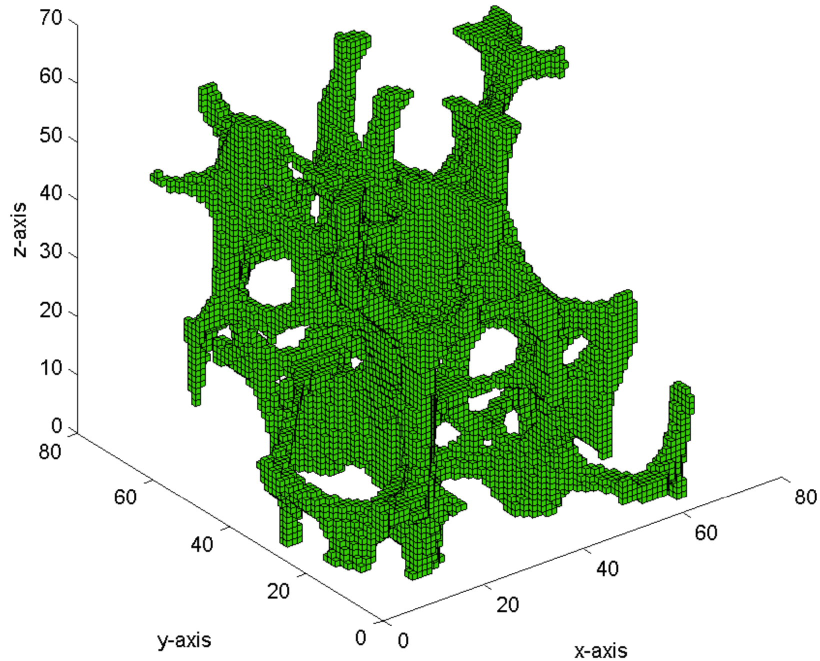


Figure 1.1: Voxel representation of a bone structure. This domain will later be defined as cube064.

is determined by both the *communication* that is necessary between the processors and the *load balance* of the distribution; a good partitioning requires little communication and each processor possesses approximately the same number of elements (or voxels). The partitioning heuristics that are usually applied to these large sparse linear systems are *(hyper)graph partitioning methods*. In such a hypergraph formulation each sparse matrix defines a graph and each voxel defines a *vertex* in the graph. Each row in the sparse matrix represents a vertex and the entries in that row define the adjacency list of this vertex, i.e. the nonzeroes in the row define the adjacent vertices. By translating the partitioning problem to a hypergraph partitioning problem one simplifies the partitioning process; rows in the sparse matrix that are *cut*, i.e. divided over several processors (or equivalently: adjacent vertices that are distributed to different processors), indicate a need for communication between processors. This is instinctively clear when we note that the dot-product for each row is calculated in the matrix-vector multiplication, so a row that is divided over several processors implies necessary communication.

However, hypergraph partitioning methods usually have no additional knowledge of the structure of the domain. Our domain is given to be embedded in a regular grid, which indicates that there likely is room for optimization when applying hypergraph methods to this problem. In this thesis we will be defining a variant of one such hypergraph partitioning method, the Mondriaan algorithm; however, this version will be specifically designed for the fact that our domain is embedded in a regular grid.

1.1 Communication Costs

Let us first define how the communication costs of a partition are defined on the grid. Each *non empty voxel* (a *filled voxel*) on the grid represents an element that needs to be distributed to one of the processors. Each filled voxel also defines a *net*, a collection of filled voxels in the domain. More specifically, a net consists of the filled voxel generating the net and all filled voxels that are adjacent to that voxel. Adjacency between filled voxels is determined by the conditions of the embedding grid; for the purposes of this paper, grid points will be considered to be adjacent if and only if they are direct neighbours on the grid. More precisely, two grid points (a_x, a_y, a_z) and (b_x, b_y, b_z) are adjacent if and only if $a_i = b_i$, $a_j = b_j$ and $a_k = b_k \pm 1$ for $i, j, k \in \{x, y, z\}$ and i, j, k all different. The adjacency of grid points is illustrated in Figure 1.2.

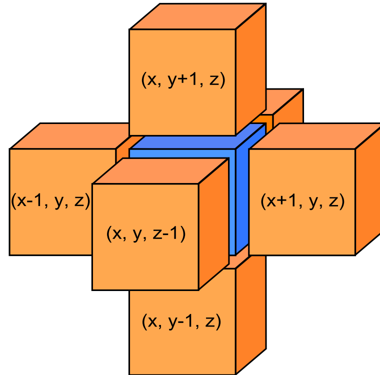


Figure 1.2: A grid point (blue) at coordinates (x, y, z) and its six adjacent grid points (orange).

When all filled voxels on a net are distributed to the same processor, that net is called *intact*; a net is called *cut* otherwise. A net n_i generated by a filled voxel v_i can instinctively be interpreted as all outgoing communication of v_i. Communication becomes necessary on n_i when one or more of its filled voxels are distributed to different processors, i.e. once the net is cut. An important subtlety is introduced in that filled voxels never have more than one outgoing communication to any given processor, i.e. when n_i contains more than one filled voxel that is distributed to processor p_j, with $p_j \neq p_i$ (p_i being the processor v_i is partitioned to), only one outgoing communication is necessary to these filled voxels. This subtlety is illustrated in Figure 1.3, where part *a*) describes all outgoing communication of voxel v_i on processor p_i on a net containing voxels

that are all distributed to different processors, while in part b) the net contains two voxels distributed to the same processor p_j, with $p_j \neq p_i$.

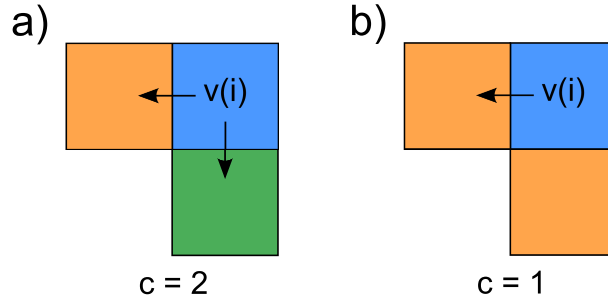


Figure 1.3: All outgoing communication for a voxel v_i on a net n_i in the case that the net contains only voxels distributed to different processors (a) and in the case that several voxels on the net are distributed to the same processor (b).

1.2 Load Balance

The partitioning methods discussed in this paper distribute the filled voxels on the domain D over p processors. The sets of filled voxels for the processors are $S_0, S_1, \ldots, S_{p-1}$, where S_i denotes the set of filled voxels distributed to processor i. Because Section 1.1 directly implies that the optimal strategy to minimise the total communication costs is to distribute all filled voxels to the same processor, the partitioning methods are also restricted by an important constraint on the *load balance* of the resulting partitioning, i.e. approximately the same number of filled voxels should be distributed to each processor. Given that each filled voxel needs to be distributed to precisely one of the processors, this goal can be formulated formally as minimizing the value of equation (1.2), where $w(S_i)$ denotes the number of filled voxels distributed to set S_i, the *weight* of the set,

$$\max_{0 \leq i < p} w(S_i). \tag{1.2}$$

In the best possible situation, the weight of each set in the partition would equal $w(D)/p$, where $w(D)$ denotes the weight of the domain, i.e. the number of filled voxels in the domain. The partitioning algorithms are however not expected to achieve this optimal situation, as it leaves little freedom for the heuristics. Although it is not feasible to expect the partitioning algorithms to achieve the optimal load balance, it does provide a good measure for the quality of the load balance of a given partitioning. The restriction that is applied to all the partitioning algorithms discussed in this paper is shown in equation (1.3)

$$\max_{0 \leq i < p} w(S_i) \leq (1 + \epsilon) \frac{w(D)}{p}. \tag{1.3}$$

This restriction essentially provides a margin for the load balance to differ from the optimal situation; if the maximum of the weights satisfies the restriction, all of the weights satisfy the margin. The value of ϵ denotes the *imbalance* of the load corresponding to the partitioning, i.e. the percentage that the load

balance differs from the best possible load balance. Unless otherwise specified, the value of ϵ is chosen to be 3%. An example of an 8-way partitioning is shown in Figure 1.4; the difference between the minimum and maximum number of filled voxels per processor is relatively large, but the constraint in equation (1.3) is satisfied.

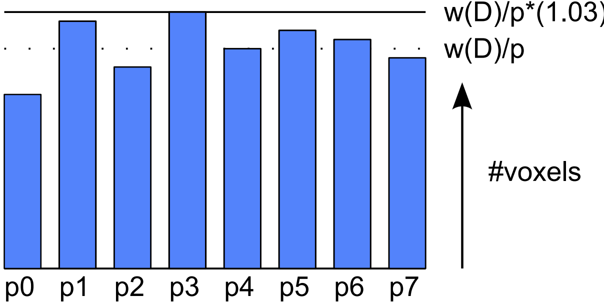


Figure 1.4: Illustration of an 8-way partitioning satisfying the load balance constraints in equation (1.3).

1.3 RCB

A straightforward algorithm applicable to the partitioning of domains embedded in a regular grid is *RCB*, or *recursive coordinate bisection*, as presented by Berger and Bokhari in [4]. In general, the RCB algorithm recursively divides the domain in a predetermined number of subdomains. The distribution heuristic used by RCB is constructed in such a way that filled voxels in the domain are evenly distributed over the subdomains, i.e. the RCB algorithm always achieves optimal load balance by design. For the purposes of this paper, each recursive call of RCB bipartitions its allocated subdomain.

As suggested by the name, the partitioning method used by RCB is solely based on the coordinates of the filled voxels in its domain. The domains handled in this paper are given to be embedded in a regular grid; this grid defines an order on the coordinates of the filled voxels. The distance between filled voxels in the domain can be determined by their coordinates alone. Each bipartitioning step performed by RCB *splits* its assigned domain in half relative to one of its coordinate directions d. In the first subdomain each filled voxel's d-coordinate is less than or equal to the *median* of all the domain's d-coordinates, in the second subdomain greater than or equal to the median for each filled voxel. This simple process is illustrated in Figure 1.5.

As illustrated in Figure 1.6, the coordinate direction over which RCB performs its split can be an important factor in the resulting communication costs. The method that RCB uses to select this direction is based on the *span* of each coordinate direction, i.e. the difference between the maximum and minimum coordinates of the dimensions in its allocated domain. In particular, in an attempt to minimize the number of adjacent filled voxels on opposite sides of the cut, the split performed by the RCB algorithm cuts over the coordinate direction with the largest span. The motivation behind this strategy is illustrated in the simple two-dimensional example in Figure 1.6, which shows a domain

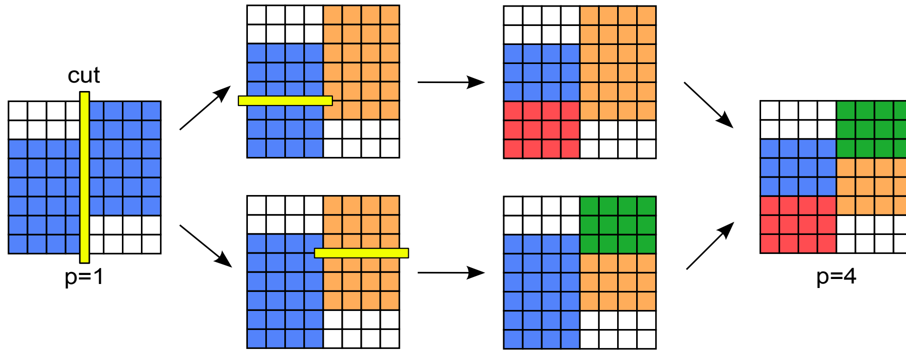


Figure 1.5: Illustration of the partitioning process of the RCB algorithm. The RCB algorithm bipartitions its allocated subdomain in each of the steps. In the first step the blue domain is split, after that the blue and orange domains both divide their respective subdomains, resulting in a 4-way partitioning.

embedded in a 2×4 grid and the two different ways to split this domain, over the x- and y-direction in Figure 1.6a) and 1.6b) respectively. This strategy automatically results in relatively *square* subdomains; one can instinctively see how the RCB method as described will minimize the total communication costs quite adequately, as neighbouring filled voxels are implicitly often distributed to the same subdomain.

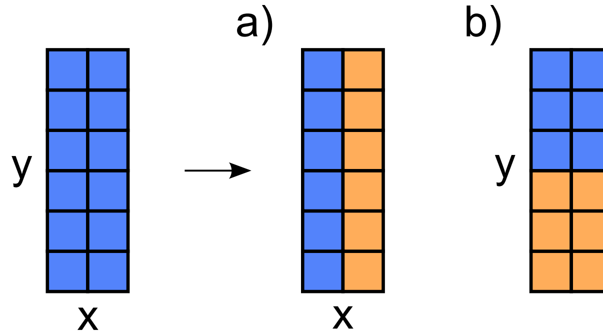


Figure 1.6: Illustration of the two different ways (with equivalent load balance) RCB could split a subdomain, either with respect to the x- or the y-axis in case a) and b) respectively. The different splits result in very different communication costs, 12 in case a) and 4 in case b). In an attempt to reduce the implied total communication costs, RCB always splits with respect to the dimension with the largest span, therefore case b) would be chosen in this example.

The span of each direction of a certain subdomain can simply be determined by finding the maximum and minimum coordinate over all filled voxels in the subdomain for each direction and subtracting them. After finding the direction with the largest span, the RCB algorithm uses the *quicksort* algorithm to sort all the filled voxels in the subdomain with respect to the chosen dimension. Since the subdomain is sorted, the median can be directly chosen as the filled

voxel in the middle of the sorted list, this median will then serve as the *pivot* for the cut operation. Pseudo-code for this RCB implementation can be found in Figure 1.3.

Since all operations in RCB are very computationally efficient, the total computation time of partitioning a domain with RCB will be very low. Because each RCB split evenly divides the filled voxels in that particular subdomain, the load balance will be optimal by definition. The communication costs, however, are only minimized in the sense that often the cutting operation implicitly distributes neighbouring filled voxels to the same subdomain.

```
input:          x : vector of x-coordinates of length n, numbered from 0 to n − 1,
                where n is the number of filled voxels in the set.
                y : vector of y-coordinates, length n.
                z : vector of z-coordinates, length n.
                p : vector of assigned processors of length n, initialized at 0.
                c : counter for the number of parts in the current partition.
                g : goal for number of parts in the final partition,
                note that g equals a power of two for this implementation.
output:         p : vector of assigned processors of length n, i.e: the final partition.
function call:  rcb(x, y, z, p, c, g).

/* The number of current parts has to be smaller than the target number */
if c < g then
    /* Calculate the span of each dimension */
    Δx = max(x) - min(x)
    Δy = max(y) - min(y)
    Δz = max(z) - min(z)
    Quicksort(x, y, z, p) w.r.t. the dimension with the largest span.

    /* Define the pivot and assign new processor values */
    pivot = ⌊(n+1)/2⌋
    for i := pivot to n − 1 do
        p(i) = p(i) + c
    endfor

    /* Recursion calls */
    rcb(x(0, pivot − 1), y(0, pivot − 1), z(0, pivot − 1), p(0, pivot − 1), 2c, g)
    rcb(x(pivot, n − 1), y(pivot, n − 1), z(pivot, n − 1), p(pivot, n − 1), 2c, g)
endif
```

Figure 1.7: Pseudo-code for the rcb implementation.

1.4 Mondriaan

A sophisticated partitioning of a domain embedded in a regular grid can be obtained with the Mondriaan distribution, as presented by Vastenhouw and Bisseling in [13]. Mondriaan uses the underlying structure of the sparsity pattern of the input matrix corresponding to the grid to generate a good partitioning.

The first process in the Mondriaan algorithm is to translate the domain partitioning problem to a hypergraph bipartitioning problem; in order to do this, the $n \times n \times n$ input grid with k filed voxels is translated to a $k^3 \times k^3$ Laplacian matrix, with each column storing information from the adjacency list of each filled voxel. Each filled voxel is now represented by a column in the Laplacian matrix, or vertex in the hypergraph bipartitioning problem.

After translating the problem to a hypergraph bipartitioning problem, Mondriaan uses the *multilevel* method, a method specifically designed for (hyper)graph partitioning problems. The multilevel method consists of the following three phases: the *coarsening* phase, in which the problem is reduced by merging similar vertices, the *initial partitioning*, in which the reduced problem is partitioned, and finally the *uncoarsening*, in which the initial partitioning is translated back into a partitioning for the original problem and refined in the process.

The similarity between sparsity patterns of the vertices can be analysed in the coarsening phase with help from the input matrix; a measure for this similarity is obtained by choosing all nonzero entries in the matrix to equal 1 and calculating the inner product of the columns of two vertices. The coarsening phase traverses a certain number of merging cycles in which vertices are merged in order to reduce the size of the domain. Each merging cycle halves the amount of vertices in the domain by pairwise merging the most similar vertices, where vertices with a larger inner product between their columns are more similar. The coarsening phase is automatically stopped once a certain empirically chosen number of vertices remain, this threshold is usually chosen around a couple of hundred vertices.

The uncoarsening phase translates the partitioning obtained from the coarsening stepwise back into a partitioning of the original domain. After undoing one of the merge cycles, the uncoarsening process also attempts to refine the obtained partitioning for the larger problem with an iterative algorithm known as the Kernighan-Lin heuristic [11], using the implementation by Fiduccia and Mattheyses suggested in [7]. Each cycle of this heuristic repeatedly chooses a vertex to move to the other set in the respective bipartition, where no vertex is allowed to be moved more than once. The beauty of the Kernighan-Lin heuristic however lies in that the chosen vertex is always the vertex that provides the highest *gain* in the current state. In other words, the vertex that once switched in the current state would reduce the communication costs of the entire domain the most; note that this reduction does not need to be positive, allowing the Kernighan-Lin heuristic to escape local minima. The best partitioning state observed in a Kernighan-Lin cycle is stored and used as initial state for the next cycle.

The operations used to obtain a Mondriaan distribution are quite computationally expensive, therefore, the required computation time to obtain a Mondriaan distribution will be significantly higher than the required computation time for, e.g., RCB. However, the resulting partitioning will presumably imply significantly less communication costs when compared to RCB.

Since the Mondriaan package is readily available at [6], the implementation details are not of great interest for the purposes of this paper. Further details on the Mondriaan distribution can be found in [5].

1.5 Goals

In order for us to compare the RCB algorithm and the established Mondriaan algorithm[1], the performance of both algorithms is tested for a few test grids, namely: cube064, a $64 \times 64 \times 64$ grid (or 64^3) with 17919 filled voxels that is shown in Figure 1.1, cube083 (83^3, 60482 filled voxels) and cube128 (128^3, 242694 filled voxels); these test cases were all provided by Peter Arbenz[2]. Note that both the computational time required to obtain the partitioning and the communication costs accompanied by that partitioning are shown.

Table 1.1: Results of both RCB and Mondriaan for the cube064 test grid.

# proc. (p)	RCB time (in s)	Mond. time (in s)	RCB comm.	Mond. comm.
2	0.010	0.627	654	111
4	0.013	1.272	1243	267
8	0.016	1.915	1781	562
16	0.018	2.646	2891	1031
32	0.021	3.453	4449	1618
64	0.038	4.385	6658	2712

Table 1.2: Results of both RCB and Mondriaan for the cube083 test grid.

# proc. (p)	RCB time (in s)	Mond. time (in s)	RCB comm.	Mond. comm.
2	0.035	1.897	970	376
4	0.047	3.722	1873	797
8	0.059	5.318	3303	1646
16	0.067	6.964	5940	2264
32	0.077	9.263	11748	3931
64	0.087	11.187	16023	6293

Table 1.3: Results of both RCB and Mondriaan for the cube128 test grid.

# proc. (p)	RCB time (in s)	Mond. time (in s)	RCB comm.	Mond. comm.
2	0.148	7.170	5407	961
4	0.210	15.581	11525	2001
8	0.263	23.354	15296	3825
16	0.306	31.731	22291	5325
32	0.350	39.607	32276	9306
64	0.388	51.461	45944	14076

As the results were expected to show, Tables 1.1, 1.2 and 1.3 indicate that the RCB algorithm is significantly faster than the Mondriaan algorithm. More importantly however, the partitions resulting from the RCB algorithm imply significantly more communication costs.

The goal of this thesis is to combine aspects of RCB and Mondriaan to develop a partitioning algorithm that has a computational complexity that is

[1]Note that Mondriaan was executed in its $1D$-mode, i.e. every column of the matrix is assigned to a processor

[2]Computer Science Department, ETH Zurich

relatively close to RCB, but whose heuristic also results in implied communication costs comparable to those of Mondriaan. This appears to be an achievable goal, because RCB is not concerned with lowering communication costs in the slightest, but still implicitly minimizes the communication costs reasonably well. Also, it must be noted again that the Mondriaan algorithm is purposed for general hypergraph partitioning problems and is not able to use the knowledge that the domain is embedded in a regular grid, knowledge that our designed algorithm will be able to use to simplify several aspects of the heuristic.

2 Methods

Inspired by the Mondriaan algorithm [13], the partitioning algorithm presented in this thesis uses the multilevel approach. However, unlike the Mondriaan algorithm, the algorithm presented here will be able to use the assumption that the domain is embedded in a regular grid, which, for example, allows for a more specialized coarsening method for equivalent computational costs. By recursively bipartitioning the domain using the multilevel method, a partitioning into 2^q parts is obtained for a certain desired $q \in \mathbb{N}$.

Each multilevel cycle starts with the coarsening of the domain as described in Section 2.1; the coarsening step lowers both the resolution of the bipartitioning problem and the partition's expected implied communication costs. Next, the coarsened structures resulting from the coarsening are bipartitioned as described in Section 2.2 to form the initial partitioning. Before the uncoarsening phase starts, the KLFM algorithm (Kernighan–Lin algorithm as implemented by Fiduccia and Mattheyses), as described in Section 2.3, is applied a certain number of times to the initial partitioning. The KLFM algorithm is also applied in the different steps of the uncoarsening process to iteratively improve the quality of the bipartitioning.

The recursive bipartitioning algorithm using the multilevel method is illustrated in Figure 2.1, where a 4-way partition for a given grid domain is obtained by traversing the multilevel V three times. Note that where the first multilevel cycle considers the whole grid and bipartitions the whole grid, the second (blue) and third (orange) multilevel cycles only consider their respective parts and only bipartition these respective parts. So even though the number of multilevel cycles required to go from a bipartition to a 4-way partition is double the amount of cycles required to obtain the bipartition, each of these multilevel cycles considers an approximately halved domain.

2.1 Coarsening

Where the Mondriaan algorithm evaluates similarity in sparsity patterns to merge nodes in the input domain, i.e. columns in the input matrix, the algorithm presented in this paper uses the underlying grid of the input domain to more efficiently merge nodes in the input domain, i.e. filled voxels on the grid.

The input for the domain in an embedded grid problem can be seen as an array with binary entries; a nonzero entry in the input array indicates that the filled voxel stored in that entry is active. Therefore, the partitioning problem can be viewed as a partitioning of a certain number of filled voxels. Goals of the partitioning of a domain embedded in a regular grid could now be formulated as: optimizing the load balance and minimizing the communication costs implied by the partitioning, where communication costs arise when two adjacent filled voxels are in different parts of the partition.

2.1.1 Definition of cluster structure

In order to lower the resolution of the grid partitioning problem, filled voxels can be merged into *clusters*. Intuitively, clusters can be viewed as a collection of adjacent filled voxels; the precise definition of the data structure is listed in Table 2.1. All data stored per cluster can be efficiently deduced from the

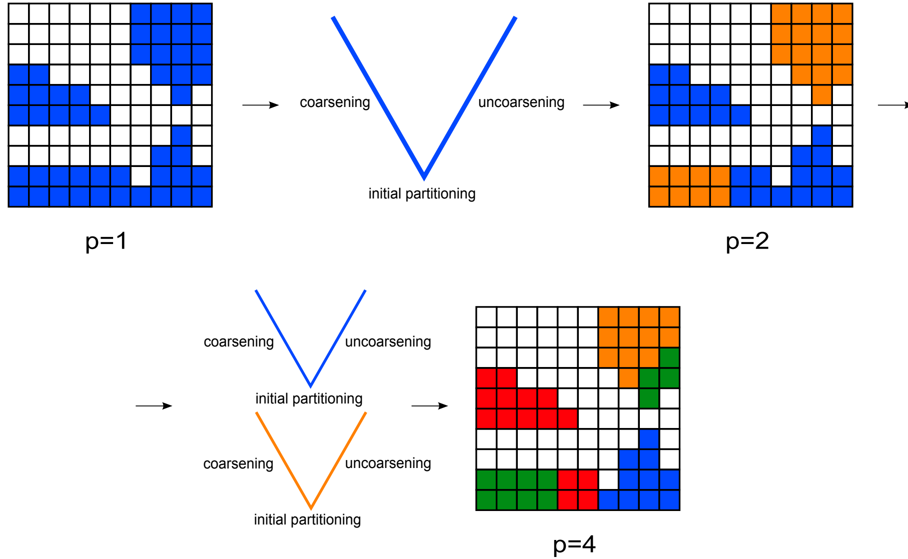


Figure 2.1: Illustration of an application of the recursive multi-level method, resulting in a 4-way partition of the domain grid. In the initial phase, all filled voxels are distributed to the blue set. The blue set traverses the multilevel method and the grid is bipartitioned to a blue and orange set, both of which call the multilevel method.

underlying grid of the domain. The coarsening step of the multilevel algorithm can now be described as multiple so called *merge cycles*; in each merge cycle, clusters are pairwise merged until no more merge operations can be performed, under the restriction that clusters can only be merged once per cycle. In the ideal situation, each merge cycle halves the number of clusters in the domain; even though this reduction is far from a realistic goal, each merge cycle will significantly lower the resolution of our domain.

In preparation for the merge cycles, the clusters need to be initialized. The initial state describes the state where there is a cluster for each filled voxel. The initialization of the clusters is very cost-efficient because of the underlying properties of the grid; to identify each adjacent filled voxel, precisely six sets of coordinates need to be considered.

2.1.2 Merge operation

The aforementioned pairwise merge operations are based on the implied communication costs between the pairs of clusters. More precisely, the implied communication costs should the two clusters be distributed to different parts in the partitioning. As described in Section 2.1, communication costs arise when adjacent filled voxels are not distributed to the same part in the partition. Note however that a filled voxel does not need to send its data to the same part more than once, so a filled voxel sends its data once to all adjacent parts. This subtlety is illustrated in Figure 2.2c), where the blue filled voxel is adjacent to two orange filled voxels that are member of the same cluster. For the purposes

of the coarsening phase, clusters are considered to be distributed as a whole, so the two orange filled voxels adjacent to the blue filled voxel will be distributed to the same part and the blue filled voxel only sends its data once.

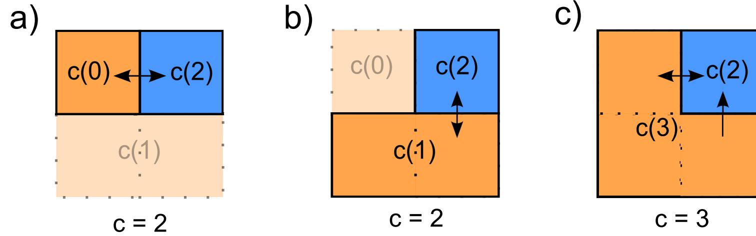


Figure 2.2: Illustration of the communication costs between different pairs of adjacent clusters should they be distributed to different parts. a) shows the relation between cluster 0 and 2, b) shows the relation between cluster 1 and 2 and c) shows the result of merging clusters 0 and 1 to obtain the merged cluster 3. Note how merging these two clusters results in a reduction of communication costs, because cluster 0 and cluster 1 share an adjacent filled voxel in cluster 2. So by merging the two clusters, 2 only has to send its data once.

Even though the exact implied communication costs of a partitioning can only be calculated when the entire partitioning is known, the pairwise costs are efficiently calculated from the cluster initialization described in Section 2.1.1. After the initialization, each cluster contains only one filled voxel. The pairwise costs between adjacent clusters always equal 2 after initialization, as adjacency between clusters in this phase is equivalent to adjacency between filled voxels and adjacent filled voxels that are not distributed to the same part have 2 mutual communications, as illustrated in Figure 2.2a). Note that adjacency between clusters in general is satisfied when at least one filled voxel from one cluster is adjacent to a filled voxel in the other cluster.

When analysing the mutual communication costs between adjacent clusters, one also has to consider the *halo points* of the clusters. Once at least one multilevel cycle has been performed and individual parts start performing multilevel cycles on the clusters that they have been assigned, these halo points no longer are a negligible factor in computing the communication costs. Halo points are defined as filled voxels that are not distributed to the part performing the multilevel cycle but *are* adjacent to at least one filled voxel that is distributed to the part performing the multilevel cycle. Figure 2.3 illustrates the influence that halo points may have on the implied communication costs of a partitioning; Figure 2.3 shows two clusters and an adjacent halo point and the two possible following scenario's, either the clusters are merged or they are not merged. From Figure 2.2b) one would expect that the mutual communication costs between cluster 0 and 1 would equal 2, but since the halo point is adjacent to both clusters of the blue part the communication can be further reduced than that by merging cluster 0 and 1 and the mutual communication is actually 3. This statement is confirmed by comparing Figure 2.3a) and 2.3b), where a difference of 3 in communication is clear.

When merging two clusters c_0 and c_1, all data as listed in Table 2.1 need to be calculated for the new merged cluster c_m. The various ID lists, e.g. the member

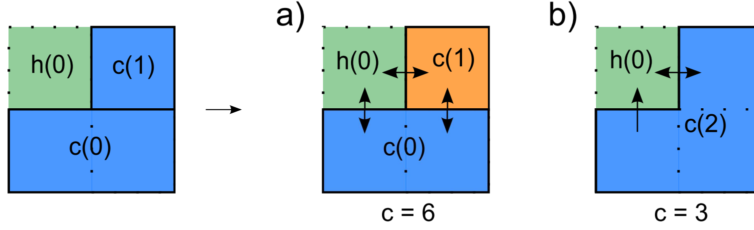


Figure 2.3: Illustration of the influence of halo points on the total communication costs. The blue part performs the coarsening and arrives at the situation with two clusters remaining and a halo point on the green part. a) shows the case where cluster 0 and 1 are not merged, b) shows the case where they have been merged; not only have the communications between cluster 0 and cluster 1 disappeared, but the halo point no longer has to send its data twice, resulting in a reduction of communication costs of 3 instead of 2.

filled voxels and the adjacent clusters, can all be constructed by determining the overlap of those lists as stored in the structures of c_0 and c_1. As each of these arrays lists various kinds of IDs, e.g. cluster IDs, filled voxel IDs or halo point IDs, each of these lists has an upper bound on their entries, i.e. the number of clusters, filled voxels or halo points respectively; because of this upper bound the overlap of the lists from c_0 and c_1 can be found extremely efficiently. By iterating the desired list of c_0 and storing a true boolean for each ID on the list in a *checklist* array one can evaluate whether a given ID in c_1's list is also present in c_0's list by checking the boolean stored at the respective ID in the checklist array. Note that constructing and using such a checklist array is only possible because the values in c_0's and c_1's list are bounded.

The most complicated field of the cluster structure that needs to be computed for the new cluster c_m is the list with the communication costs with respect to each adjacent cluster c_{adj}. If the respective adjacent cluster is only adjacent to c_0, the communication costs between c_m and c_{adj} are given by

$$\text{costs}(c_m, c_{adj}) = \text{costs}(c_0, c_{adj}) + \#(\text{halopoints}(c_1) \cap \text{halopoints}(c_{adj})), \quad (2.1)$$

and analogously if the respective adjacent cluster is only adjacent to c_1. If the adjacent cluster is adjacent to both c_0 and c_1 then the communication costs with respect to c_m are given by

$$\begin{aligned}\text{costs}(c_m, c_{adj}) = \text{costs}(c_0, c_{adj}) + \text{costs}(c_1, c_{adj}) - \\ \#(\text{adjfilledvoxels}(c_0, c_{adj}) \cap \text{adjfilledvoxels}(c_1, c_{adj})). \quad (2.2)\end{aligned}$$

A merge procedure on a small grid that describes both of these equations is shown in Figure 2.4. The course of action is elaborated upon below:

- This merge procedure is initialized on a 3×2 grid consisting of 5 filled voxels and 1 halo point, shown in Figure 2.4a); note that $\text{costs}(c_a, c_b) = 2$ for each combination (c_a, c_b).

Table 2.1: Definition of the *cluster* data structure.

Type	Name	Description
int	*flag*	Flag to keep track of whether this cluster has already been merged in the current merge cycle
int	*nfilledvoxels*	The number of filled voxels that are joined in this cluster structure
int	*nadjclusters*	The number of clusters adjacent to this cluster
int	*nadjhalopoints*	The number of halo points adjacent to this cluster
int[ ]	*nadjfilledvoxels*	The number of adjacent filled voxels per adjacent cluster
int[ ]	*filledvoxels*	The ID of each filled voxel that is a member of this cluster
int[ ]	*adjclusterids*	The ID of each cluster adjacent to this cluster
int[ ]	*adjhalopointids*	The ID of each halo point adjacent to this cluster
int[ ]	*costs*	For each adjacent cluster: the communication costs implied were this cluster and the adjacent cluster not to be merged
int[ ][ ]	*adjfilledvoxelids*	The ID of each filled voxel adjacent to this cluster per adjacent cluster

- Figure 2.4b) shows the situation after c_0 and c_1 have been merged into c'_0. Because of the merge operation the values of $\text{costs}(c'_0, c_2)$ and $\text{costs}(c'_0, c_3)$ need to be calculated. Equation (2.1) can be used for both, as neither of c_2 and c_3 are adjacent to both c_0 and c_1. In order for us to calculate $\text{costs}(c'_0, c_3)$, we must note that c_0 and c_3 share precisely one halo point; using equation (2.1) we get $\text{costs}(c'_0, c_3) = 3$, exactly as illustrated in Figure 2.3. Calculating $\text{costs}(c'_0, c_2) = 2$ can be done a lot more straightforward because c_0 and c_2 have no common adjacent halo points.
- Figure 2.4c) shows the situation after c_2 and c_4 have been merged into a new c'_2. Because of this merge operation, the values of $\text{costs}(c'_2, c_3)$ and $\text{costs}(c'_2, c'_0)$ need to be calculated. Because neither of c'_0 and c_3 is adjacent to both c_2 and c_4, equation (2.1) needs to be applied for both of these calculations. Since c'_2 shares no adjacent halo points with c_3, this results straightforwardly in $\text{costs}(c'_2, c_3) = 2$ and $\text{costs}(c'_2, c'_0) = 2$, which mirrors the example in 2.2b)
- Figure 2.4d) shows the final situation after c'_0 and c'_2 have been merged into c''_0- after which only $\text{costs}(c''_0, c_3)$ needs to be updated. Because c_3 is adjacent to both c'_0 and c'_2, equation (2.2) needs to be applied for this calculation, which results in $\text{costs}(c''_0, c_3) = \text{costs}(c'_2, c_3) + \text{costs}(c'_0, c_3) - \#(\text{adjfilledvoxels}(c'_0, c_3) \cap \text{adjfilledvoxels}(c'_2, c_3)) = 2 + 3 - 1 = 4$. Note how equations (2.1) and (2.2) have ensured that the potential gain in communication caused by h_0 is considered just one time, i.e. the first time it becomes a factor in step b).

2.1.3 Merge cycle

A merge cycle iterates over all clusters and merges each cluster c with its adjacent cluster with the highest costs relative to c, given that cluster c has adjacent clusters and at least one of those was not yet merged in that merge cycle. The coarsening phase ends when either of two conditions has been met: a certain

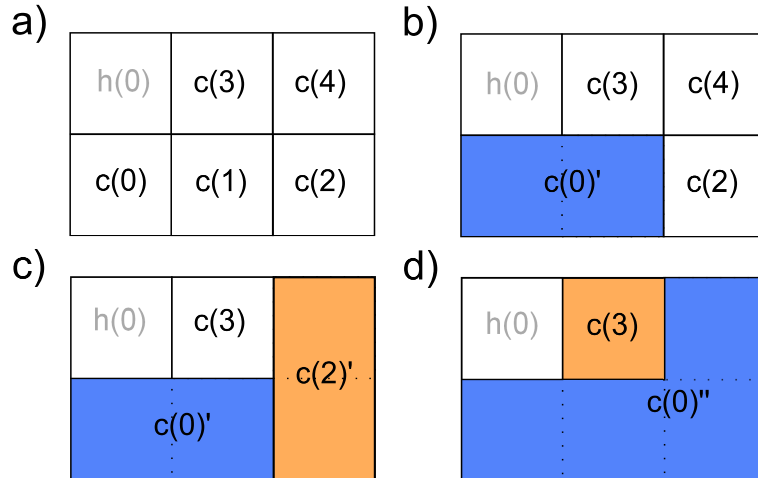


Figure 2.4: Illustration of a merge procedure between 5 filled voxels on a 3×2 grid with 1 halo point.

preset number of merge cycles has been performed or a sufficiently low number of clusters is remaining in the domain.

The merging process is designed to implicitly minimize the partitioning's implied communication costs as clusters will be greedily merged with the neighbouring cluster that at that point in the process appears to be most effective in reducing the total communication costs.

2.2 Initial partitioning

After completing the coarsening phase, the multilevel method starts the partitioning process by obtaining an initial partitioning. Where the coarsening attempts to minimize the communication costs of the partitioning, the initial partitioning focuses on optimizing the initial load balance. Three strategies for the initial partitioning are to be compared, the first being solely focused on optimizing the initial load balance, the second trying to use simple data resulting from the coarsening to minimize the communication costs of the initial partitioning to some extent and the third trying to use the net structure constructed by Kernighan–Lin to significantly lower the communication costs.

Since the first two initial partitioning algorithms are not concerned much with minimizing the communication costs, similar to Mondriaan's initial partitioning method, no guarantees regarding the quality of the resulting partitioning can be given. However, initial partitioning algorithms are designed to be computationally quick, so building in a certain randomness allows each part traversing the initial partitioning phase to generate several different versions of the initial partitioning cheaply and choose the best one to enter the uncoarsening phase. The quality of the partitionings can be compared relatively cheaply, since the only filled voxels necessary to compare the quality are the filled voxels distributed to the part performing this multilevel cycle and their halo points, similar to the description in Section 2.1.2. The way this is implemented will later be explained in detail in Section 2.3.2; essentially, the net structure of

the coarsened grid defining the communication costs as described in Section 1.1 needs to be constructed in its entirety for the Kernighan–Lin cycles over the initial partitioning. Fortunately, this structure can be created ahead of time; by using the net structure of the coarsened grid, one can cheaply determine the relative quality of each provided initial partitioning (we say cheaply because this net structure is later used for the Kernighan–Lin cycles, so very few unnecessary calculations are performed).

2.2.1 Karmarkar–Karp

Omitting concerns regarding the communication costs allows for the use of a fast and simple bipartitioning method known as the differencing method, introduced by Karmarkar and Karp [10].

The differencing method repeatedly removes the two clusters with largest weights w_0 and w_1 (with $w_0 \geq w_1$) from the set and replaces them with a dummy cluster with weight $w_0 - w_1$. Once only one dummy cluster remains, the differencing method traverses back over the sequence of differencing operations to obtain the actual bipartition. This back traversal is based on repeatedly applying the operations in (2.3), where a 2-way partition (A', B') is translated to a 2-way partition (A, B) of equal weight difference, meaning that the weight of the dummy cluster starting the back traversal equals the total difference in weight in the final bipartitioning,

$$A = A' - \{|a - b|\} + \{a\}, \quad B = B' + \{b\}, \quad \text{if } a > b \text{ and } (a - b) \in A'. \tag{2.3}$$

In the current implementation, the clusters are stored in a singly linked list, storing each cluster's weight and parent clusters, two parents if it is a dummy cluster, zero otherwise. By using quicksort to initialize the list as a list sorted in descending order, the two clusters with the largest weight can repeatedly be chosen in linear time as the front two clusters. Since a singly linked list has no efficient random access, inserting the dummy node so that the list maintains its sorting is relatively slow, i.e. each insert operation has complexity $O(n_c)$, with n_c being the number of clusters. Once a dummy node with a weight that is already present in the list is inserted back into the list, the default (deterministic) strategy is to add the respective dummy node before other nodes with equal weight; a random element can be added by giving the dummy node a certain probability r_k to skip each of the nodes with equal weight (starting at the first), which over multiple runs likely results in different partitionings with equal load balance. Since the algorithm is guaranteed to generate $n_c - 1$ dummy clusters, it has to perform $n_c - 1$ insert operations, meaning that the current implementation of the algorithm is $O(n_c^2)$. An example of the Karmarkar–Karp algorithm for a simple problem is given in Figure 2.5.

For further improvements to the Karmarkar–Karp algorithm, a heap implementation can be attempted. This should bring the algorithm's complexity down to $O(n_c \log n_c)$.

2.2.2 Expanding neighbours

Unlike the Karmarkar–Karp algorithm, the *expanding neighbours* algorithm attempts to implicitly reduce the implied communication costs of the resulting

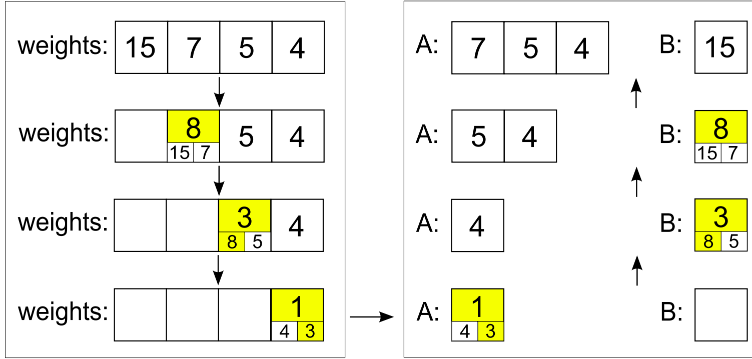


Figure 2.5: Illustration of the Karmarkar–Karp algorithm. On the left side the two largest clusters are repeatedly replaced by a dummy node of their difference. Dummy nodes are denoted by yellow nodes, which store their b and a parent clusters in order for the algorithm to easily traverse back by repeatedly applying equation (2.3) on the right side.

initial partitioning. To accomplish this, it uses the results from the coarsening, as the coarsening phase calculates all data from Table 2.1 for each cluster and Karmarkar–Karp uses none of this information. The expanding neighbours algorithm performs cheap operations by using the number of adjacent clusters as an indication of which clusters to distribute to the same part in the partitioning.

The expanding neighbours algorithm starts with all its clusters partitioned to the same part. Next, it finds the cluster with the fewest adjacent clusters, switches this cluster to the other part and starts expanding from there. The algorithm does not allow clusters to be switched more than once and it maintains a candidate list of adjacent clusters (of already switched clusters) that have not yet been switched. After switching the first cluster, the expanding neighbours algorithm operates according to a simple priority list: if the candidate list is not empty, choose the candidate with the fewest neighbours; if the candidate list is empty, choose the remaining cluster with the fewest adjacent clusters on the domain. Once the balance constraints are satisfied, the algorithm continues switching candidates until the load balance no longer improves; it then chooses the best encountered load balance and stops the partitioning algorithm. Note that an ultimate load balance is not guaranteed, but the load balance will be sufficient to start the uncoarsening phase.

By continuously adding clusters with few adjacent clusters to the other part, the expanding neighbours algorithm attempts to minimize the number of adjacent clusters not distributed to the same part after finishing the initial partitioning phase. When encountering several clusters with the same minimal number of adjacent clusters, a choice is made with a random variable r_e, similar to the strategy in the previous section. Thus, several runs of the expanding neighbours algorithm will likely result in several different initial partitionings and the best can be chosen to start the uncoarsening phase.

2.2.3 Net split

Given that both the Karmarkar–Karp and the expanding neighbours algorithm use at most a rough measure for minimizing the communication costs, let us now define an initial partitioning method that is more focused on minimizing the communication costs implied by the bipartitioning. Given that communication costs are defined in terms of nets (Section 1.1), this method is called the *net split* algorithm. Net split starts with all clusters from the coarsened grid distributed to the same processor, similar to the expanding neighbours algorithm (Section 2.2.2). Next, it switches clusters to the other processor until the balance constraints are satisfied.

Inspired by Mondriaan's coarsening method, which evaluates the sparsity pattern of the filled voxels, the net structure of the coarsened grid is used to list the total number of times each pair of clusters is present on the same net. The net split method then sorts the neighbour list of each cluster w.r.t. the number of paired occurrences on the nets using the quicksort algorithm. While constructing these paired occurrences lists, the pair of clusters with the most paired occurrences is stored and switched to initialize the algorithm. After listing all neighbours of these two initial clusters as possible candidates, the remaining calculations can be described in the form of a simple priority list like in the expanding neighbours algorithm: if any candidates remain, switch the candidate cluster with the highest total number of paired occurrences with all previously added clusters. If no candidate remains, choose the pair of unadded clusters with the most paired occurrences and switch them both. If no pair can be added because of the balance constraints, switch a single cluster if and only if it improves the load balance. Add the unadded neighbours of the newly added cluster(s) to the candidate list and repeat until the balance constraints are satisfied.

Note that, given that the coarsening ensures very few clusters, these calculations are very cheap. Because this initial partitioning is likely to be very efficient, we have opted to not build any randomness into it; instead, the net split algorithm is executed several times, each time initialized with another *unique* pair of clusters occurring on the nets ($\text{netsplit}(c_0, c_1) = \text{netsplit}(c_1, c_0)$ is guaranteed for any two clusters c_0, c_1, because the net split algorithm is deterministic).

2.3 KLFM and uncoarsening

2.3.1 Kernighan–Lin

The communication costs of the partitioning are initially lowered in the coarsening phase; the initial partitioning phase, however, is not specialized in lowering the communication costs, but rather in finding a good load balance. After obtaining the initial partitioning, the multilevel method attempts to improve the communication costs both before and during the uncoarsening phase. By translating the original problem to a graph partitioning problem, the Kernighan–Lin heuristic [11] can be applied to iteratively improve the initial partitioning.

The Kernighan–Lin heuristic is an iterative algorithm for improving graph partitions. Let us define the gain of a cluster in a bipartition as the (possibly negative) gain in total communication costs should this cluster be assigned to the other part in the bipartition. The Kernighan–Lin heuristic performs a certain number of cycles over the clusters, where in each cycle it iteratively

moves the cluster with the highest gain, prevents that cluster from being moved again that cycle and stores the bipartitioning with the lowest communication costs encountered during that cycle. The move of a cluster is only allowed if it satisfies all balance constraints, otherwise the optimal communication costs would obviously be obtained by moving all clusters to one part. Note that the Kernighan–Lin heuristic chooses the best possible move at each step in the cycle, but also allows for this gain to be negative, thus enabling the algorithm to escape local minima.

Determining the optimal move at each step is not a trivial task however; nor are the required computations occurring after a cluster has been successfully moved and the gains for the affected clusters need to be updated. An incredibly delicate and efficient implementation was introduced by Fiduccia and Mattheyses [7]; this implementation manages to perform each cycle in linear time.

2.3.2 Fiduccia-Mattheyses

The Kernighan–Lin implementation by Fiduccia-Mattheyses (KLFM) is a famous iterative mincut heuristic for partitioning networks that can be readily applied to the graph partitioning problem described in Section 2.3.1. As proved in [7], the KLFM implementation is linear in the size of the graph, i.e. the number of nets; let us discuss how to apply the KLFM implementation to the graph partitioning problem.

Let the set of cells described in KLFM be defined as the clusters obtained from the coarsening. Each cell then defines a net as the collection of that cell and its directly neighbouring cells. For each cell, a doubly linked list is created storing the nets it appears on (the CELL lists) and for each net a doubly linked list is created storing each cell that appears on it (the NET lists).

The main idea of the KLFM implementation is that by defining the graph bipartition problem in terms of cells and nets, both calculating and updating the cell gains can be done by incrementing and decrementing the cell gains a small number of times. The increment and decrement operations are determined by iterating over the aforementioned CELL lists and checking simple conditions as described in great detail in [7]. A major efficiency trick is to maintain and store the cell gains in a bucket array from index $-p_{\max}$ to $p_{\max}$ ($p_{\max}$ denoting the maximum gain possible), storing for each index a doubly linked list of each cell with cell gain equal to that index; two BUCKET lists are stored, one for each of the parts in the bipartition.

The process of selecting the next cell to move, the *base cell*, can be described as follows: first the candidates from both of the parts in the bipartition are selected, i.e. the cells with the highest gain. Then the algorithm checks whether moving the candidates would violate the balance constraints; if no move satisfies the balance constraints, the current cycle will terminate and no further moves will be attempted. If both moves satisfy the balance constraints, the move resulting in the best balance is chosen. Since the bucket arrays maintain the index of the highest non-empty bucket, the selection of each part's candidate costs only $O(1)$ operations, as a cell in the highest non-empty bucket for one of the parts is by definition a cell with maximum gain for that part.

Once a cell has been moved, it is removed from its bucket and placed on a free cell list, which is used to reinitialize the BUCKET lists when a new cycle starts. While updating the cell gains, the algorithm iterates over all nets and

selects the cells on these nets that have not yet been moved, i.e. cells that are free.

After the KLFM run chooses which cell to move, this cell must be locked, as it is not allowed to move again for the remainder of this KLFM run. There is no explicit description of how to allow the KLFM algorithm to efficiently maintain which cells on the nets are free in [7]; in our implementation, each cell stores pointers to each of its occurrences on the NET lists as depicted in Figure 2.6. Once a cell has been moved, the pointers to all the cell's occurrences on the NET lists are used to move the occurrence of this cell to the end of that NET list and to mark them as locked. By moving each locked cell to the end of the NET list, we allow the net iteration to stop once the first locked cell has been considered, as all following cells will also be locked. Since pointers to the cell's occurrences and pointers to the end of the NET lists are given, each move costs only $O(1)$ operations.

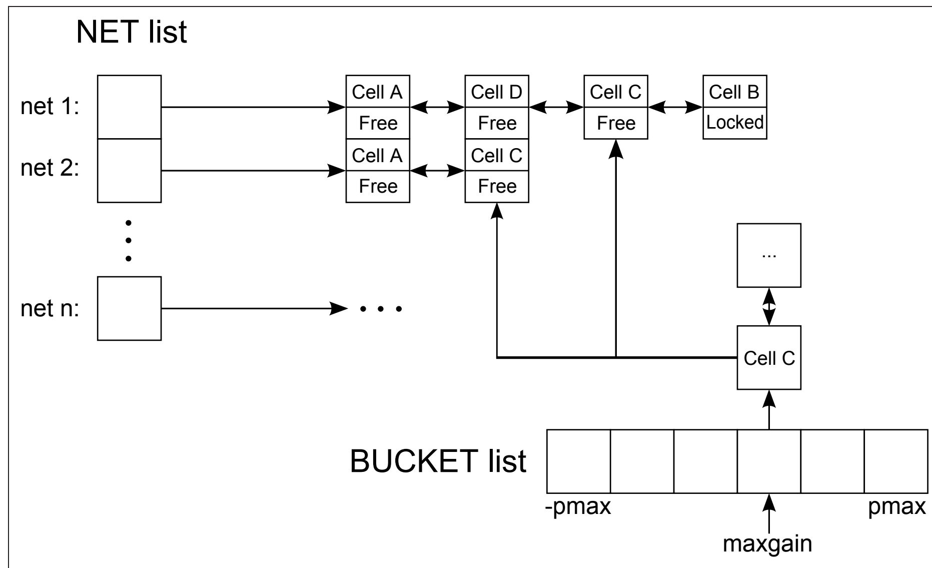


Figure 2.6: Illustration of the structure of the BUCKET and NET doubly linked lists, where each cell on the BUCKET lists has pointers (each arrow represents a pointer) to all of that cell's occurrences on the NET lists.

After moving a cell, the gain of each neighbouring free cell needs to be updated, meaning that cells might need to be moved to other buckets. Similar to the pointers to each cell's occurrence on the NET lists, in our implementation of the KLFM algorithm each cell also stores a pointer to its position on either the free cell list or on one of the bucket lists as illustrated in Figure 2.7. Now, for each cell gain update a pointer to the cell on the bucket lists is given and that cell can be moved to the front of the destination bucket, so each cell move between buckets again costs only $O(1)$ operations.

These data structures may appear to be overly complicated, but they allow all of the steps within a Kernighan–Lin cycle to have complexity $O(1)$.

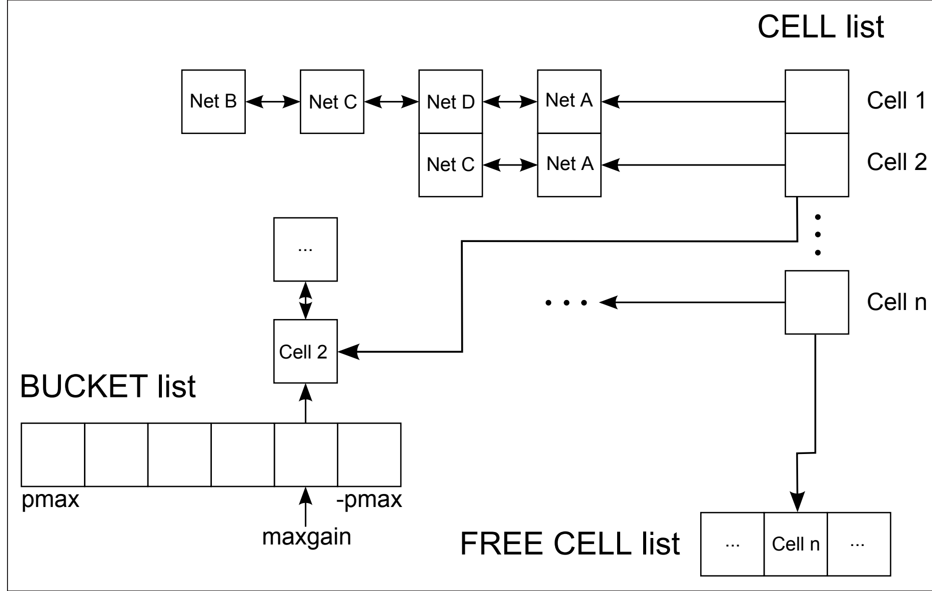


Figure 2.7: Illustration of the structure of the BUCKET, FREECELL and CELL doubly linked lists, where each cell on the CELL list has a pointer (each arrow represents a pointer) to either that cell's occurrence on the FREECELL list or its occurrence on the BUCKET list.

2.3.3 Uncoarsening

After the initial partitioning is chosen, the multilevel approach will be continued by starting the uncoarsening phase; the uncoarsening phase essentially does the exact opposite of the coarsening phase, i.e. the partitioning of the lowest resolution grid (the largest clusters) will iteratively be translated to a distribution of the highest resolution grid. Each time a distribution of the clusters of merge cycle m is directly translated to a distribution of the clusters of merge cycle $m-1$, the KLFM algorithm is applied once to the higher resolution grid, to enhance the obtained partitioning further.

Since the uncoarsening phase travels back along the steps from the coarsening phase, the KLFM algorithm should be able to parse all resolutions of the grid that occurred during the coarsening phase. While it might be possible to initialize the KLFM algorithm for all resolutions that passed during the coarsening phase using the corresponding cluster structure from that merge cycle and the adjacency lists from the clusters, for the purposes of this thesis, the cells will be equal to the clusters, so that *moving a cell* will equal *moving a cluster*, but the adjacencies will be determined by the filled voxel adjacencies on the highest resolution. This could be accomplished by saving copies of all cluster structures over all the merge cycles, but this would be very costly. A more efficient way would be to save a copy of the initial cluster structure, before the merge cycles started, and to store a hash list for every merge cycle, storing per filled voxel what cluster it was a member of in that merge cycle. Thus, the net structures can be initialized by the initial cluster structure and only need to be initialized once. Then, the uncoarsening step corresponding to a certain coarsening step

can use a copy of this net structure, use the hash list to filter all duplicate values from the NET lists, i.e. two separate filled voxels that are member of the same cluster will become equivalent entries in the NET lists. The initialized NET lists can then be used to construct the CELL lists. An example of the difference between CELL and NET lists for different resolution levels is given in Figure 2.8, wherein a simple 2×2 grid is distributed to one part and the CELL and NET lists are initialized for 0 and 1 performed merge cycles respectively.

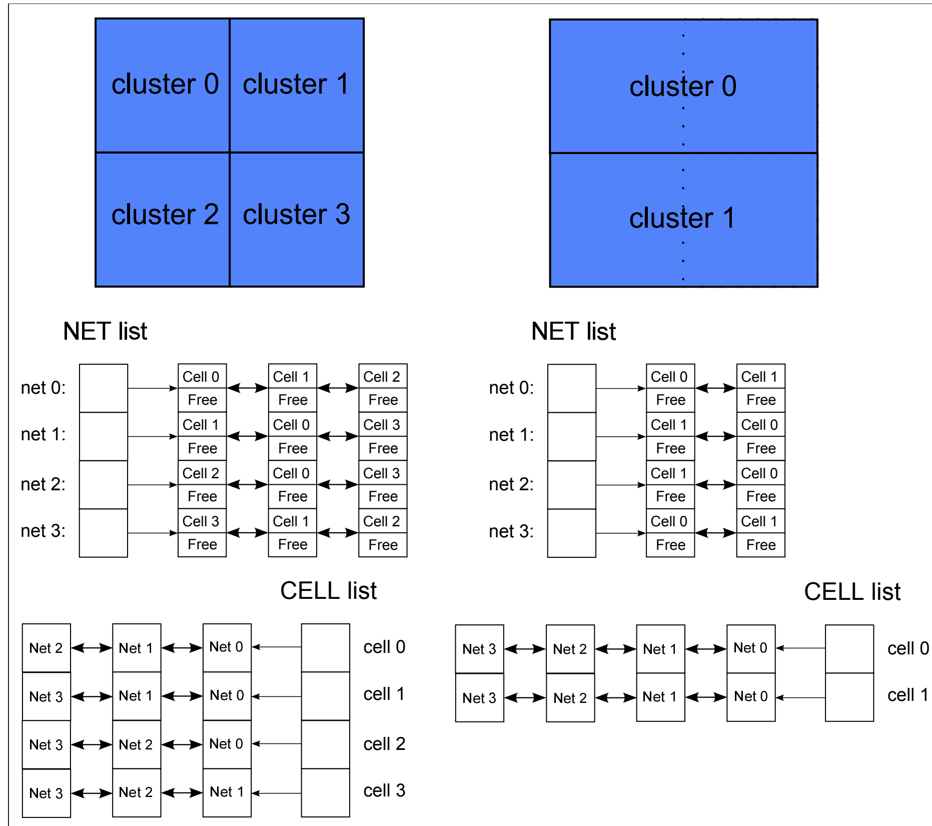


Figure 2.8: Illustration of the initialization of the KLFM algorithm for two different resolutions of the grid, the left part of the figure represents the highest resolution of the grid, the right part of the grid is clustered after 1 merge cycle.

This approach has as consequence that the NET lists only need to be initialized once for an entire uncoarsening phase; also, the average length of the NET lists will be significantly shorter because filled voxels are now clustered, the amount of NET lists, however, will not decrease when compared to the highest resolution representation of the grid. The amount of CELL lists will also decrease significantly, the length of each CELL list, however, will increase, because cells now represent multiple filled voxels and are therefore part of more nets. Overall, the clustering will still result in a significantly more efficient KLFM run.

3 Results

3.1 Coarsening

In order to reduce the grid resolution (the problem size), the suggested multilevel bipartitioning algorithm traverses the coarsening phase; in the coarsening phase the filled voxels are merged to form cluster structures. These merge operations are based on the implied communication costs between the clusters, which are deduced from knowledge of the finest grid level as described in detail in Section 2.1.2. Lowering the grid resolution results in a significantly faster initial partitioning and uncoarsening phase, as both of their complexities depend on the number of clusters on the coarsened grid. By taking pairwise communication costs between clusters into account in the merge operations, the coarsening also aims to implicitly minimize the communication costs of the resulting bipartitioning.

3.1.1 Threshold for terminating the merge process

An important parameter determining the effectiveness of the coarsening phase is the number of clusters at which to stop the merging process. Obviously, would the entire grid be connected, the process could continue until only one cluster remains.

Reducing the number of clusters to near single digits poses several problems however. The number of adjacent clusters per cluster decreases significantly near the end of the merge process; this has as effect that clusters might be picked as merge candidate because of a lack of options, not because of their significant contribution to the quality of the partitioning. An effective way to prevent this from happening is to simply stop the merge process if the number of clusters does not decrease significantly over an entire cycle; this strategy is also used in the Mondriaan algorithm. The coarsening in Mondriaan is terminated if the number of clusters does not decrease more than 5% over the course of an entire merge cycle. In our experiments we have seen equivalent quality by choosing 10% as this *reduction ratio*; however, taking this ratio results in an overall speed-up of up to 8%. For all of the tests discussed in this section we therefore take 10% as the minimal reduction ratio.

When realizing that the merge operation essentially approximates a gain in communication costs by merging clusters, it becomes clear that one of the risks of reducing the cluster number too much is that performing merge cycles results in a loss of information about the highest resolution grid. As mentioned before, the number of clusters contributes to the success of the later phases in the multilevel method; finding an initial partitioning that satisfies the balance constraints might even prove to be impossible for an extremely low number of clusters. Finally, even though Kernighan–Lin's run time will benefit greatly from a smaller number of clusters, its ability to incrementally improve the quality of the partitioning will likely be hindered by the rough estimations at lower grid resolutions.

Notable histogram samples from the first entire coarsening phase of a single bipartitioning of test matrix cube064 are shown in Figures 3.1 through 3.5. In Figure 3.1, one can see how the clusters are initialized, i.e. each filled voxel is its own cluster. Figure 3.2 indicates that indeed almost all of the clusters

have quadrupled their size by the time they could have merged twice, with only a few falling behind. Figure 3.3 is the point at which Mondriaan would stop, around 200 remaining clusters; one can see how the clusters are spread out very unevenly over the different cluster sizes, with most clusters hovering around the 125 mark. Figures 3.4 and 3.5 show what happens with the clusters when the coarsening cycle continues, showing 50 and 3 remaining clusters respectively. These two histograms indicate that clusters are much more evenly spread over the different cluster sizes once the coarsening phase nears its end.

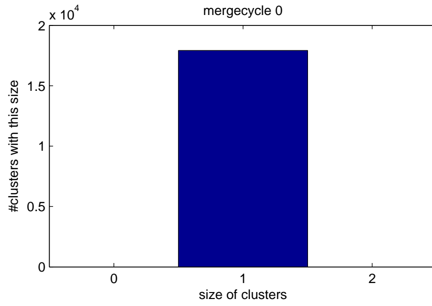


Figure 3.1: The cluster sizes before performing any merge cycles in the coarsening phase of the first bipartitioning performed for the test grid cube064, 17919 clusters.

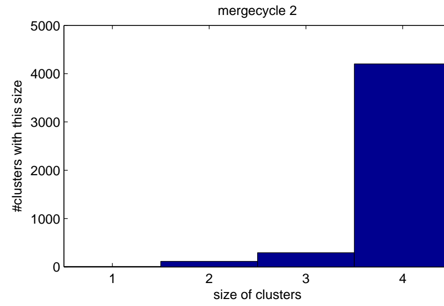


Figure 3.2: The varying cluster sizes after two merge cycles in the coarsening phase of the first bipartitioning performed for the test grid cube064, 4616 clusters.

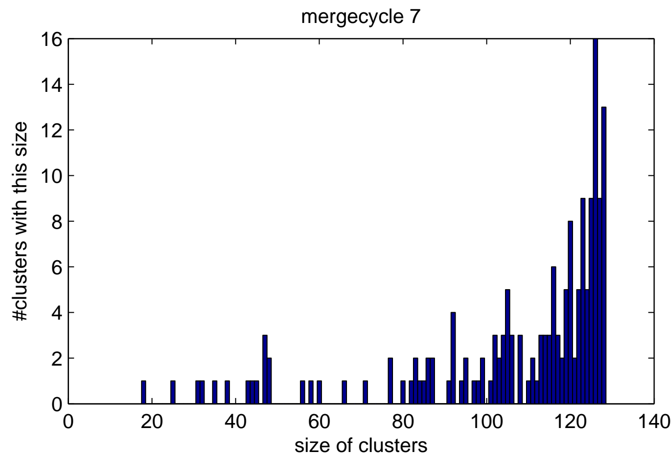


Figure 3.3: The varying cluster sizes after 7 merge cycles in the coarsening phase of the first bipartitioning performed for the test grid cube064, 168 clusters.

From our experiments, the coarsening appears to provide better overall results for the partitioning when it is terminated after reaching 50 clusters, as opposed to Mondriaan where the coarsening is stopped after reaching 200 clusters. A sample of these experiments are shown in Table 3.1 and Table 3.2 where the run time and implied communication costs are shown for all the test matrices using the threshold of 200 and 50 respectively. These tests were run with

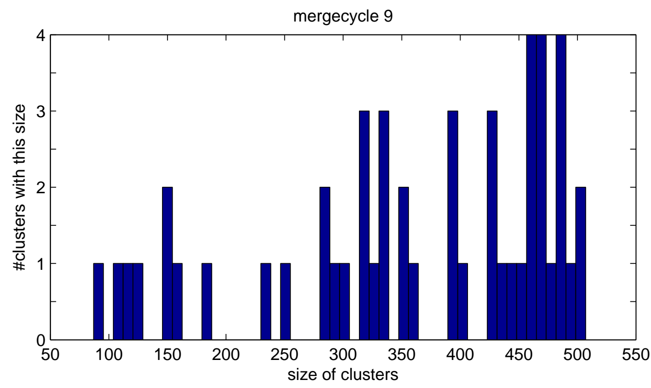


Figure 3.4: The varying cluster sizes after 9 merge cycles in the coarsening phase of the first bipartitioning performed for the test grid cube064, 50 clusters

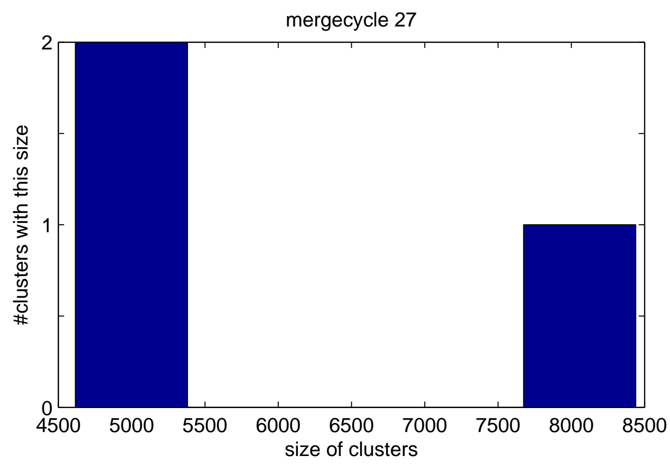


Figure 3.5: The varying cluster sizes after 27 merge cycles in the coarsening phase of the first bipartitioning performed for the test grid cube064, 3 clusters

the same set of parameters[3] and appear to indicate that for our test matrices the threshold of 50 provides an overall speedup of up to 15%, with improved scaling for larger grids and greater number of processors. The costs implied by the resulting partitionings in the tables also suggest that choosing the threshold to equal 50 provides better communication costs.

	time (in s)			communication		
# proc. (p)	cube064	cube083	cube128	cube064	cube083	cube128
8	1.135	5.077	24.365	543	1573	4310
64	2.949	11.674	79.49	3521	8355	14697

Table 3.1: Quality of the partitioning when the coarsening is stopped after reaching 200 clusters.

	time (in s)			communication		
# proc. (p)	cube064	cube083	cube128	cube064	cube083	cube128
8	1.042	4.751	23.041	511	1446	3438
64	3.103	11.266	67.43	3323	7367	14337

Table 3.2: Quality of the partitioning when the coarsening is stopped after reaching 50 clusters.

Concluding, the coarsening phase is terminated when either of the equations in (3.1) and (3.2) are satisfied,

$$\#(\text{clusters}(m)) \leq 50 \tag{3.1}$$

$$\#(\text{clusters}(m)) > \#(\text{clusters}(m-1)) \cdot 0.9, \tag{3.2}$$

where m defines the number of the merge cycle.

3.2 Initial Partitioning

As described in Section 2.2, once the merge operations in the coarsening phase have constructed the cluster structures, the initial partitioning phase provides a bipartitioning of these clusters. The initial partitioning serves as a starting point for the uncoarsening phase, in which the merge operations are back traversed to obtain a partitioning for the considered filled voxels. While the uncoarsening process iteratively improves the initial partitioning, a better quality of the initial partitioning generally translates to a better bipartitioning resulting from the uncoarsening phase. Let us therefore compare the implied communication costs of each of the initial partitioning strategies discussed in 2.2 and the computational time required to obtain these partitionings.

Just as in Section 1.5, the three test grids will be denoted by cube064, cube083 and cube128, in ascending order of grid dimensions. By evaluating the partitioning quality right after performing the individual initial partitioning methods, the quality of the raw initial partitioning methods can be compared

[3] Merge reduction ratio of 0.9, initial partitioning generated with netsplit, Kernighan–Lin on the initial partitioning until no more improvements are found, 1 Kernighan–Lin cycle in each step of the uncoarsening, $\epsilon = 0.03$.

quite well. While we should not expect a high-quality partitioning, disabling all Kernighan–Lin cycles, in both the initial partitioning and the uncoarsening phase, allows us to discern which method is most beneficial to our algorithm.

Each of the initial partitioning methods will be initialized with the same coarsening phase, i.e. with the parameter values as defined in Section 3.1.1. The test results for the Karmarkar–Karp initial partitioning algorithm, as described in Section 2.2.1, are shown in Table 3.3.

	time (in s)			communication		
# proc. (p)	cube064	cube083	cube128	cube064	cube083	cube128
2	0.009	0.025	0.093	1196	4476	14592
4	0.019	0.053	0.202	2873	9300	23648
8	0.033	0.083	0.263	5032	13059	32347
16	0.049	0.117	0.394	7958	19707	44932
32	0.072	0.156	0.592	12511	27764	65159
64	0.111	0.210	0.636	18213	39248	88113

Table 3.3: Run time and implied communication costs of the Karmarkar–Karp initial partitioning algorithm. Note that the coarsening phase is excluded from the run times and no Kernighan–Lin refinement is performed.

In order to give the Karmarkar–Karp algorithm a higher chance of success, the algorithm is called 50 times with $r_k = 50\%$, as described in Section 2.2.1; this variable is chosen relatively high as the number of times that (dummy) clusters on the stack have equal weight will not be very high. After performing these 50 runs, the best initial partitioning observed is chosen as the definitive initial partitioning. While the algorithm is extremely fast, when taking into account that these times include 50 individual runs for each bipartitioning, the resulting partitionings are quite poor. Even the RCB runs listed in Tables 1.1 through 1.3 provide significantly better partitionings in comparable times. Even though this method was designed with speed in mind, in an attempt to let the coarsening and uncoarsening minimize the communication costs, Kernighan–Lin can not be expected to make up for a factor of up to 3 between the implied communication costs of Karmarkar–Karp and RCB.

	time (in s)			communication		
# proc. (p)	cube064	cube083	cube128	cube064	cube083	cube128
2	0.009	0.026	0.089	1343	3956	12614
4	0.018	0.053	0.182	2629	8244	20510
8	0.029	0.084	0.348	4034	10666	30662
16	0.047	0.114	0.384	6341	16566	37620
32	0.074	0.154	0.506	9450	23684	56485
64	0.118	0.205	0.632	13780	32941	75245

Table 3.4: Run time and implied communication costs of the expanding neighbours initial partitioning algorithm. Note that the coarsening phase is excluded from the run times and no Kernighan–Lin refinement is performed.

Next, the test results for the expanding neighbour algorithm, as described in Section 2.2.2, are shown in Table 3.4. Similar to Karmarkar–Karp, in an attempt to boost the chances of success for the expanding neighbours algorithm,

the algorithm is applied 50 times. The random variable r_e (Section 2.2.2) for the expanding neighbours is chosen lower than Karmarkar–Karp's random variable, at 30%. This value is chosen to account for the fact that the situation where clusters with equal amount of neighbours are considered will occur quite often. By comparing Tables 3.3 and 3.4 one can easily observe that the Karmarkar–Karp and the expanding neighbours algorithm have equivalent run times. But, whereas Karmarkar–Karp essentially bipartitions randomly w.r.t. the communication costs, the expanding neighbour algorithm uses a fairly effective measure for the communication costs, which allows it to minimize the communication costs somewhat. Table 3.4 appears to confirm this statement, because the expanding neighbours method is up to 20% more effective in minimizing communication costs than Karmarkar–Karp.

	time (in s)			communication		
# proc. (p)	cube064	cube083	cube128	cube064	cube083	cube128
2	0.006	0.015	0.056	385	1003	2809
4	0.013	0.032	0.114	730	2598	7617
8	0.023	0.047	0.178	1245	4461	10981
16	0.035	0.062	0.316	2230	7825	18542
32	0.050	0.082	0.339	3288	12083	28645
64	0.074	0.107	0.387	5402	19163	41099

Table 3.5: Run time and implied communication costs of the net split initial partitioning algorithm. Note that the coarsening phase is excluded from the run times and no Kernighan–Lin refinement is performed.

Finally, the test results for the net split algorithm, as described in Section 2.2.3, are shown in Table 3.5. Whereas the Karmarkar–Karp and expanding neighbours algorithm are performed randomly for a certain random variable, the net split algorithm is deterministic; however, the net split method is performed with each unique cluster as its starting cluster and the best resulting partitioning is picked as the final initial partitioning. The fact that our threshold for stopping the coarsening phase lies at a relatively low number of clusters, significantly improves the run time of this algorithm, as is shown in Table 3.5. Compared to the other two initial partitioning algorithms, the net split algorithm is clearly the fastest, registering times that are up to twice as fast. More importantly, the communication costs resulting from the net split algorithm are also significantly better than those of the other two algorithms. The communication costs resulting from the net split method are up to a factor 4 lower, a factor that does not appear to scale very well. While the improvement factor for low numbers of processors can climb upwards of 4, the factor for $p = 32$ and $p = 64$ lies closer to 2. This can most likely be explained by the fact that the net split operations, much like those in Kernighan–Lin, are aimed at optimizing the communication costs between the two processors involved in the current partitioning. By increasing the number of processors, more levels of optimization are introduced before reaching the lowest level, where final parts are distributed between processors. This increasing gap between the first and last bipartitionings means that the first few optimization rounds are much less relevant to the final partitioning.

Nevertheless, the net split is faster than the Karmarkar–Karp and expand-

ing neighbours algorithms and generates initial partitionings with significantly better implied communication costs. All further tests will therefore make use of the net split initial partitioning method, which synergizes extremely well with the choice made in Section 3.1.1 to aim for a relatively low number of clusters going into the initial partitioning.

3.3 Uncoarsening and the algorithm as a whole

Let us now evaluate the influence of Kernighan–Lin on our final partitioning. The partitioning results for the coarsening phase plus the initial partitioning algorithms, not including the Kernighan–Lin run, were already shown in Section 3.2; as these describe our entire algorithm without Kernighan–Lin, they will provide a good comparison. The test results for the entire algorithm, including Kernighan–Lin, are listed in Table 3.6. The cases use mainly the same set of parameters[4], the main difference being that one Kernighan–Lin cycle is performed at each uncoarsening step for cube083 and cube128, where two cycles are done for cube064, because this improves the quality of the partitioning moderately, while adding no significant computation time.

	time (in s)			communication		
# proc. (p)	cube064	cube083	cube128	cube064	cube083	cube128
2	0.382	1.532	6.982	158	360	859
4	0.828	2.980	13.914	257	991	1862
8	1.315	4.037	22.595	500	1446	3438
16	1.610	6.218	30.619	935	2747	5582
32	1.986	7.632	42.393	1576	4474	9611
64	3.141	11.232	58.780	3197	7367	14337

Table 3.6: Run times and resulting communication costs of our complete suggested algorithm for all test grids.

Table 3.5 together with Table 3.6 illustrate the importance of Kernighan–Lin nicely. The uncoarsening phase and initial partitioning refinement improve the quality of the partitioning by a factor ranging from 2 to 3, even spiking at 4. The refinement does not come without a price, however, as the run times of the algorithm become significantly longer.

Figure 3.6 illustrates the effect that Kernighan–Lin has on the run time of the algorithm, executed with the same set of variables for each run[5]. The coarsening and uncoarsening phases make up the bulk of the run time. However, the coarsening phase, as shown here, is severely faster than Mondriaan's coarsening phase, which determines at least half of its run time for large grids. While Figure 3.6 indicates that each phase scales rather well for an increased processor number, it also appears to indicate that the uncoarsening phase scales rather badly when the grid size is increased; the uncoarsening phase accounts

[4]The cluster threshold equals 50, net split is used as initial partitioning algorithm, Kernighan–Lin is applied to the initial partitioning until no further improvement is found (2-3 cycles on average), $\epsilon = 0.03$.

[5]The cluster threshold equals 50, net split is used as initial partitioning algorithm, Kernighan–Lin is applied to the initial partitioning until no further improvement is found (2-3 cycles on average), 1 cycle of Kernighan–Lin in the uncoarsening phase for cube083 and cube 128, 2 cycles for cube064, $\epsilon = 0.03$.

for approximately 55% of the run time for cube064, around 71% for cube083 and 80% for cube 128. Note that this very clearly illustrates the negative effect of using the net structure of the highest resolution grid for the initialization of Kernighan–Lin, as Mondriaan's uncoarsening phase scales much better; in Mondriaan the coarsening phase takes up the bulk of the computational time for large grids.

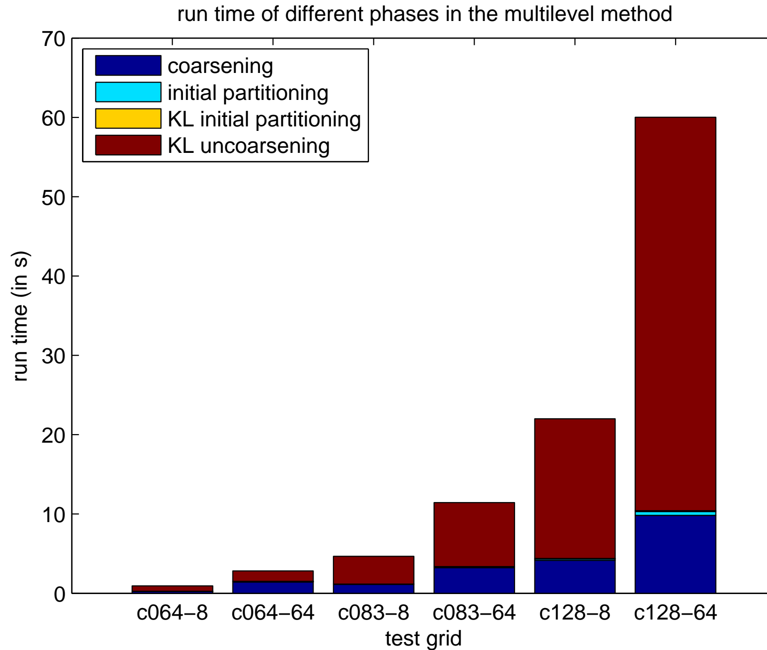


Figure 3.6: The run time of a sample of tests on our test grids cube064, cube083 and cube128 divided up into the different parts of the algorithm.

Let us now compare the scaling of our algorithm to the scaling properties of Mondriaan. The run times of each of these algorithms for each of the test grids are shown in the graph in Figure 3.7. While Tables 1.1 through 1.3 clearly indicate that RCB scales sublinearly in the number of processors, Figure 3.7 seems to indicate that both our algorithm and Mondriaan scale sublinearly as well. For cube064, our algorithm seems to scale a lot better than Mondriaan, but upon increasing the grid size our algorithm appears to scale only marginally better than Mondriaan, it even appears to spike near the end of our processor range, a spike that gets more distinct for larger grid sizes.

Unfortunately, Figure 3.7 and Table 3.6 together with the tables for RCB and Mondriaan, i.e. Table 1.1, 1.2 and 1.3, suggest that our suggested algorithm is not a significant improvement upon the Mondriaan algorithm. Somewhat significant improvements were made for the test grid cube064, where the run-time was approximately halved and the implied communication costs were also improved, albeit less significantly. The run time is still of the order of that of Mondriaan, however, and not comparable to RCB's run times, as hoped. For the test grids cube083 and cube128, our algorithm seems to be somewhat equivalent

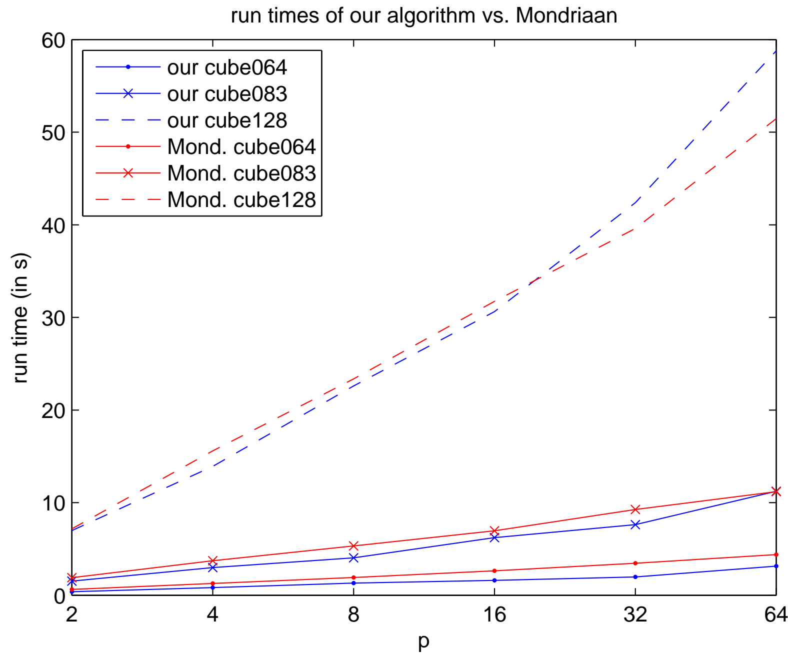


Figure 3.7: The run times for our algorithm compared to those of Mondriaan for each of the testgrids.

to Mondriaan and marginally better, respectively; this assessment is especially true for relatively low number of processors, where our algorithm improves in both run time and communication costs. However, for a relatively high number of processors ($p = 64$) our algorithm appears to be inferior to Mondriaan.

Finally, to illustrate the results of our suggested algorithm, let us take the graphic representation of cube064 as shown in Figure 1.1 and partition the grid into 8 parts with our algorithm; the result of this partitioning is visualized in Figure 3.8. This figure is quite interesting, as even from this simple illustration one can make out the workings of our suggested algorithm and how it manages to minimize the communication costs: all thin pieces of bone are used to transition from one processor to another and most major chunks of bone are distributed in their entirety to the same processor.

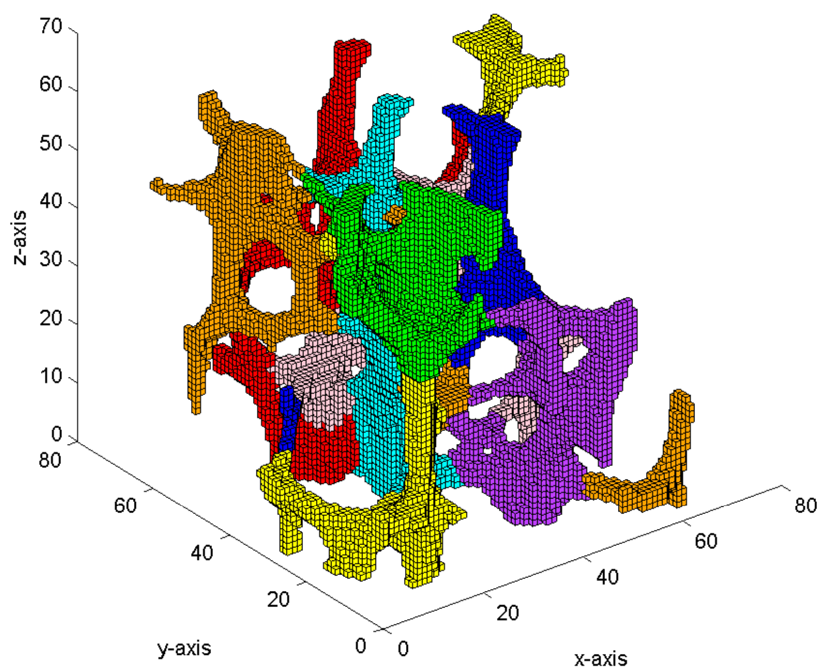


Figure 3.8: Voxel representation of an 8-way partitioning of the test grid cube064 using our suggested algorithm.

4 Conclusion

In this thesis we presented a designed partitioning method for domains embedded in a regular grid; these partitioning problems have applications in the field of osteology. The main condition of these partitionings is that each processor should be assigned approximately the same number of elements on the domain, with a small permitted imbalance. Each partitioning also implies a certain communication cost, based on neighbouring elements on the domain (in the case of an embedding grid, these can be seen as filled voxels on the grid) that are not distributed to the same processor. The goal of such a partitioning method is to minimize the communication cost resulting from its partitioning.

Two established partitioning algorithms are found in RCB, which generates its partitioning very quickly, but minimizes its communication only to a certain degree, and Mondriaan, which takes a relatively long time generating its partitioning, but minimizes its communication costs very well. Taking cues from partitioning algorithms such as RCB and Mondriaan, the goal of the new partitioning method presented in this thesis was to use the added knowledge, that the grid provides, to improve the Mondriaan algorithm in such a way that the run time would be of the order of that of RCB, while also providing implied communication costs that were comparable to Mondriaan's.

Our main strategy was to use the same philosophy as Mondriaan, i.e. to use the multi-level method, a method specifically designed for (hyper)graph partitioning problems. The multi-level method consists of three phases: the coarsening phase, in which the given domain size is reduced, the initial partitioning, in which the reduced domain is bipartitioned, and finally the uncoarsening phase, in which the coarsening phase is back traversed to translate the initial partitioning iteratively into a partitioning of the original domain.

The strategy we used to optimize the coarsening phase was to merge filled voxels into so called cluster structures based on the communication costs involved should they not be distributed to the same processor. These cluster structures maintain very specific data about their neighbours to ensure that merging these cluster structures can be executed efficiently. Initializing these cluster structures can be done extremely efficiently as the grid guarantees that each filled voxel has at most six neighbouring filled voxels.

For the initial partitioning phase we tested three different algorithms, each with a different speed-quality trade-off. One of these initial partitioning algorithms is known as Karmarkar–Karp, the other two were designed by us, the expanding neighbours and net split algorithms, with an increasing effort in minimizing communication costs. From the test results obtained by each of these initial partitioning methods, we could conclude that the net split method is superior in every way. By letting the coarsening phase continue until only about 50 clusters remain, net split was able to profit greatly from the low cluster count, once called; net split attempts each of its possible resolutions and chooses the best initial partitioning.

For the uncoarsening phase we used a similar, but slightly optimized, method as Mondriaan, i.e. applying the iterative refinement algorithm known as Kernighan–Lin as implemented by Fiduccia–Mattheyses (KLFM) at each step in the uncoarsening. The uncoarsening constructs the data structures necessary to execute KLFM by using the net structure of the highest resolution grid and the clusters of the grid in the latest uncoarsening step.

Our implemented coarsening method improves significantly upon Mondriaan's coarsening in computational time; where Mondriaan's coarsening phase determines at least half of the total computational time for large grids, our coarsening phase only accounts for around a sixth of the total computational time.

The net split algorithm suggested in this thesis can also be seen as a significant improvement on Mondriaan's initial partitioning method, which almost exclusively considers load balance, because net split severely reduces the communication costs of the initial partitioning with almost no added overhead.

The communication costs resulting from our partitionings are relatively good, improving upon Mondriaan in most cases, but failing to improve Mondriaan's costs for the highest tested number of processor.

Unfortunately, the proposed partitioning algorithm as a whole does not improve Mondriaan as significantly as we aimed for. While for cube064 (a 64^3 grid), our suggested algorithm improved Mondriaan up to a factor of approximately 2 in run time, for the larger grid sizes (cube083 and cube128, being 83^3 and 128^3 grids, respectively) our algorithm was only marginally faster than Mondriaan, if at all. For cube064 our suggested algorithm's run time scales significantly better than Mondriaan, but for cube083 and cube128 our suggested algorithm's run time scales only marginally better than Mondriaan. These results can most likely be attributed to the fact that our implementation of the uncoarsening has to use the net structure of the highest resolution grid and thus scales worse than Mondriaan's uncoarsening implementation, which uses the net structure of the lowest resolution grid.

4.1 Future Work

A possible reason that our algorithm is unable to improve Mondriaan more significantly is that we essentially tried to rebuild Mondriaan in such a way that it used the added knowledge, that our embedding grid provides, to try and improve Mondriaan; we could instead focus more on entirely different (uncoarsening) strategies in the future. I would however be mostly interested in attempting to implement KLFM in such a way that the net structure of the lowest resolution grid level is used instead of the net structure of the highest resolution grid level. This should provide a very significant speed up to our algorithm.

References

[1] P. Arbenz, S.D. Margenov, and Y. Vutov. Parallel MIC(0) preconditioning of 3D elliptic problems discretized by Rannacher-Turek finite elements. *Computers and Mathematics with Applications*, 55(10):2197–2211, 2008.

[2] R. Barrett et al. *Templates for the solution of linear systems: building blocks for iterative methods*. Society for Industrial Mathematics, 1987. No. 43.

[3] C. Bekas, A. Curioni, P. Arbenz, C. Flaig, G.H. van Lenthe, R. Muller, and A.J. Wirth. Massively parallel graph partitioning: A case in human bone simulations. In *Combinatorial Scientific Computing*, pages 407–425. Chapman & Hall/CRC Press, 2012.

[4] M.J. Berger and S.H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 100:570–580, 1987.

[5] R.H. Bisseling. *Parallel Scientific Computation, A structured approach using BSP and MPI*. Oxford University Press, 2004.

[6] R.H. Bisseling. Mondriaan package 3.11: http://www.staff.science.uu.nl/ bisse101/mondriaan/mondriaan_v3.11.tar.gz, June 2012.

[7] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th IEEE Design Automation Conference*, pages 175–181. IEEE Press, Los Alamitos, CA, 1982.

[8] G.H. Golub and C. F. van Loan. *Matrix Computations*. The John Hopkins University Press, 4th edition, 2013.

[9] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[10] N. Karmarkar and R.M. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. *23rd Annual Symposium on Foundations of Computer Science*, pages 312–320, 1982.

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

[12] L.J. Melton et al. Contribution of in vivo structural measurements and load/strength ratios to the determination of forearm fracture risk in postmenopausal women. *Journal of Bone and Minderal Research*, 22(9):1442–1448, 2007.

[13] B. Vastenhouw and R. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM review*, 47:67–95, 2005.